Python Arcade Library

Release 3.0.0.dev26

Paul Vincent Craven

Feb 18, 2024

GET STARTED

CHAPTER

ONE

WHAT IS ARCADE?

Arcade is an easy-to-learn Python library for creating 2D video games. It is ideal for people learning to program, or developers that want to code a 2D game without learning a complex framework.

CHAPTER TWO

START HERE

2.1 Installation

Arcade can be installed like any other Python Package. Arcade needs support for OpenGL 3.3+. If you are familiar with Python package management you can just "pip install" Arcade. For more detailed instructions see *Installation*.

2.2 Getting Help

If you get stuck, you can always ask for help! See the page on *How to Get Help* for more information.

2.3 Tutorials

If you are already familiar with basic Python programming, follow the *Simple Platformer* as a quick way to get up and running. If you are just learning how to program, see the Learn Arcade book.

2.4 Arcade Skill Tree

- Basic Drawing Commands - See How to Draw with Your Computer, drawing_primitives
  - ShapeElementLists - Batch together thousands of drawing commands into one using a `arcade.ShapeElementList`. See examples in *Faster Drawing with ShapeElementLists*.
- Sprites - Almost everything in Arcade is done with the `arcade.Sprite` class.
  - Basic Sprites and Collisions
  - Individually place sprites
  - Place sprites with a loop
  - Place sprites with a list
- Moving player sprites
  - Mouse - sprite_collect_coins
  - Keyboard - sprite_move_keyboard

    - Keyboard, slightly more complex but handles multiple key presses better: sprite_move_keyboard_better
    - Keyboard with acceleration, de-acceleration: sprite_move_keyboard_accel
    - Keyboard, rotate and move forward/back like a space ship: sprite_move_angle
  - Game Controller - sprite_move_controller
    - Game controller buttons - *Supported, but documentation needed.*
- Sprite collision detection
  - Basic detection - Learn arcade book on collisions, sprite_collect_coins
  - Understanding collision detection and spatial hashing: *Collision detection performance*
  - Sprite Hit boxes
    - Detail amount - *`arcade.Sprite`*
    - Changing -*`arcade.Sprite.hit_box`*
    - Drawing - `arcade.Sprite.draw_hit_box`
  - Avoid placing items on walls - sprite_no_coins_on_walls
  - Sprite drag-and-drop - See the *Solitaire*.
- Drawing sprites in layers
- Sprite animation
  - Change texture on sprite when hit - sprite_change_coins
- Moving non-player sprites
  - Bouncing - sprite_bouncing_coins
  - Moving towards player - sprite_follow_simple
  - Moving towards player, but with a delay - sprite_follow_simple_2
  - Space-invaders style - slime_invaders
  - Can a sprite see the player? - line_of_sight
  - A-star pathfinding - astar_pathfinding
- Shooting
  - Player shoots straight up - sprite_bullets
  - Enemy shoots every x frames - sprite_bullets_periodic
  - Enemy randomly shoots x frames - sprite_bullets_random
  - Player aims - sprite_bullets_aimed
  - Enemy aims - sprite_bullets_enemy_aims
- Physics Engines
  - SimplePhysicsEngine - Platformer tutorial *Step 3 - Many Sprites with SpriteList*, Learn Arcade Book Simple Physics Engine, Example sprite_move_walls
  - PlatformerPhysicsEngine - From the platformer tutorial: *Step 4 - Add User Control*,
    - sprite_moving_platforms
    - Ladders - Platformer tutorial *Step 10 - Adding a Score*

- Using the physics engine on multiple sprites - *Supported, but documentation needed.*
- Pymunk top-down - *Supported, needs docs*
- Pymunk physics engine for a platformer - *Pymunk Platformer*

• View management

- Minimal example of using views - view_screens_minimal
- Using views to add a pause screen - view_pause_screen
- Using views to add an instruction and game over screen - view_instructions_and_game_over

• Window management

- Scrolling - sprite_move_scrolling
- Add full screen support - full_screen_example
- Allow user to resize the window - resizable_window

• Map Creation

- Programmatic creation
  - Individually place sprites
  - Place sprites with a loop
  - Place sprites with a list
- Procedural Generation
  - maze_depth_first
  - maze_recursive
  - procedural_caves_bsp
  - procedural_caves_cellular
- TMX map creation - Platformer tutorial: *Step 8 - Collecting Coins*
  - Layers - Platformer tutorial: *Step 8 - Collecting Coins*
  - Multiple Levels - sprite_tiled_map_with_levels
  - Object Layer - *Supported, but documentation needed.*
  - Hit-boxes - *Supported, but documentation needed.*
  - Animated Tiles - *Supported, but documentation needed.*

• Sound - Learn Arcade book sound chapter

- music_control_demo
- Spatial sound sound_demo

• Particles - particle_systems

• GUI

- Concepts - *GUI Concepts*
- Examples - *GUI Concepts*

• OpenGL

- Read more about using OpenGL in Arcade with *OpenGL*.

    - Lights - light_demo
    - Writing shaders using “ShaderToy”
        - *Shader Toy - Glow*
        - *Shader Toy - Particles*
        - Learn how to ray-cast shadows in the *Ray-casting Shadows*.
        - Make your screen look like an 80s monitor in *CRT Filter*.
        - Study the Asteroids Example Code.
    - Rendering onto a sprite to create a mini-map - minimap
    - Learn to do a compute shader in *Compute Shader*.
- *Logging*

CHAPTER

THREE

INSTALLATION

Arcade runs on Windows, Mac OS X, and Linux.

Arcade requires Python 3.7 or newer. It does not run on Python 2.x.

Select the instructions for your platform:

3.1 Installation on Windows

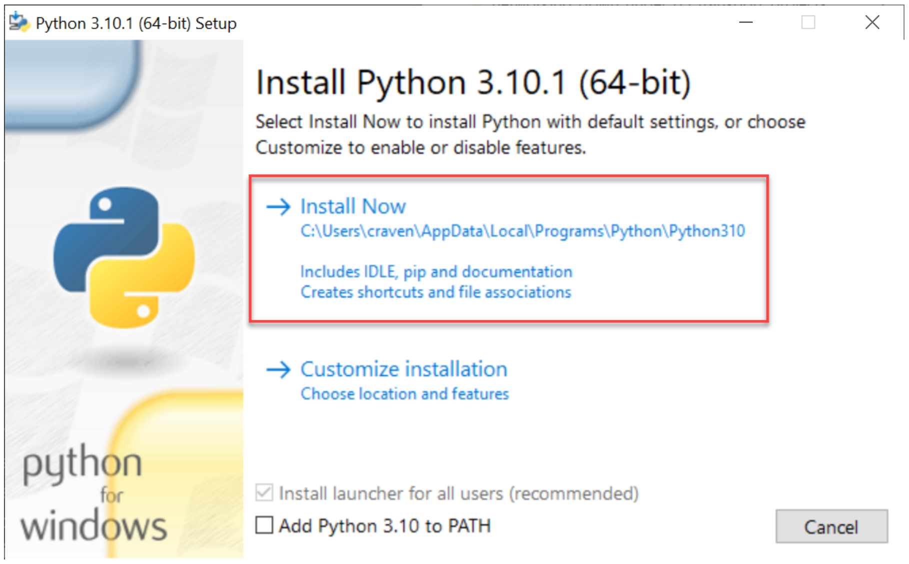
To develop with the Arcade library, we need to install Python, then install Arcade.

3.1.1 Step 1: Install Python

Install Python from the official Python website:

https://www.python.org/downloads/

Run the downloader. From there, you can just click 'install'. If you aren't using an IDE like PyCharm or Visual Studio, you might want to also mark the checkbox and add Python to the path.



Python 3.10.1 (64-bit) Setup

Install Python 3.10.1 (64-bit)

Select Install Now to install Python with default settings, or choose Customize to enable or disable features.

→ Install Now
C:\Users\craven\AppData\Local\Programs\Python\Python310

Includes IDLE, pip and documentation
Creates shortcuts and file associations

→ Customize installation
Choose location and features

☑ Install launcher for all users (recommended)
☐ Add Python 3.10 to PATH

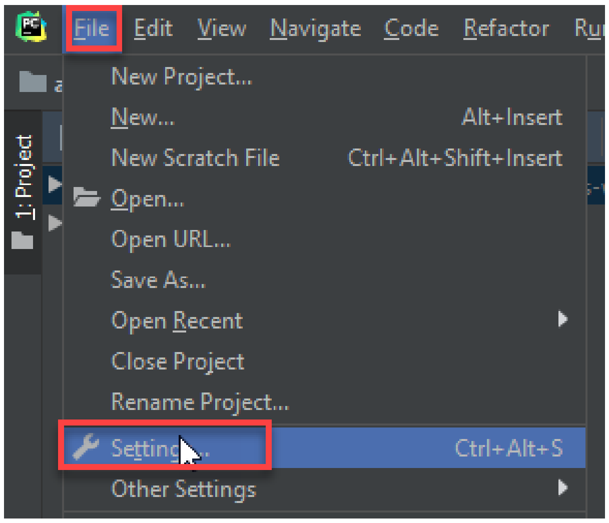
Once installed, you can just close the dialog. There's no need to increase the path length, although it doesn't hurt anything if you do.

3.1.2 Step 2: Install The Arcade Library

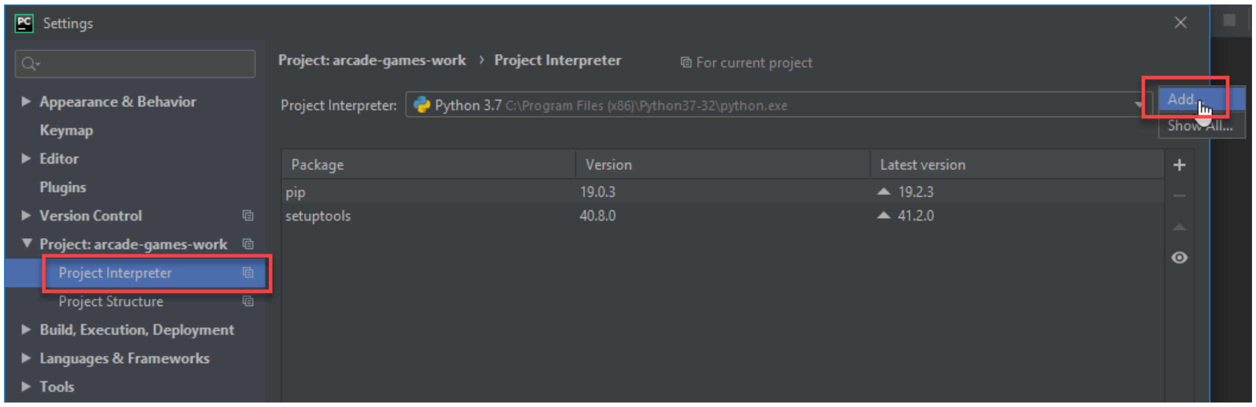
If you install Arcade as a pre-built library, there are two options on how to do it. The best way is to use a "virtual environment." This is a collection of Python libraries that only apply to your particular project. You don't have to worry about libraries for other projects conflicting with your project. You also don't need "administrator" level privileges to install libraries. Instructions for doing this with the PyCharm IDE are below:

Install Arcade with PyCharm and a Virtual Environment

If you are using PyCharm, (the community edition works great and is free) setting up a virtual environment is easy. Once you've created your project, open up the settings:



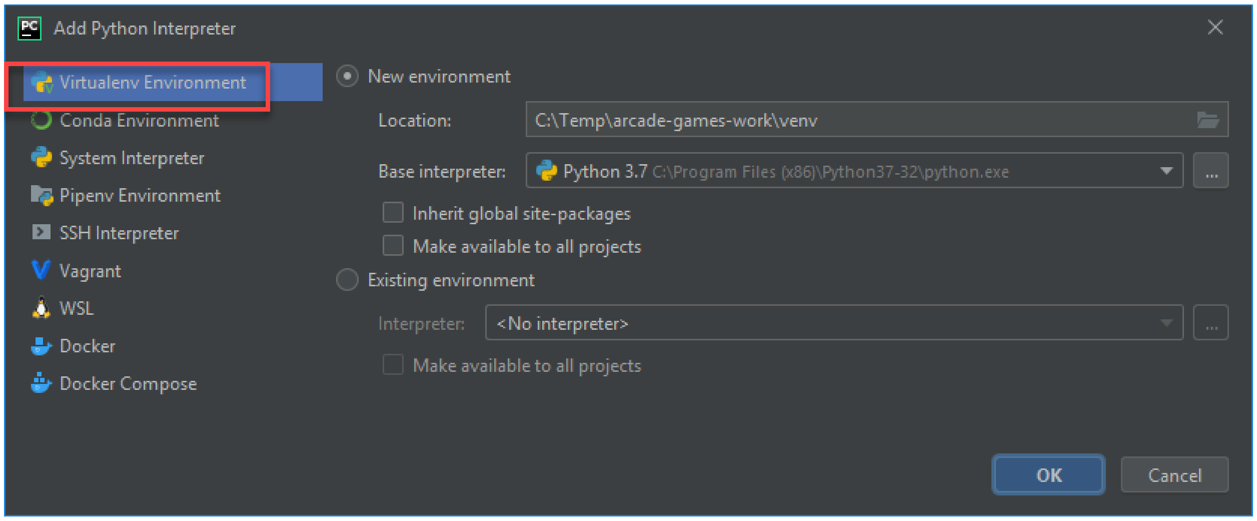
Select project interpreter:



Project Interpreter: Python 3.7 C:\Program Files (x86)\Python37-32\python.exe

Package	Version	Latest version
pip	19.0.3	▲ 19.2.3
setuptools	40.8.0	▲ 41.2.0

Create a new virtual environment. Make sure the venv is inside your project folder.


Add Python Interpreter

- Virtualenv Environment
- Conda Environment
- System Interpreter
- Pipenv Environment
- SSH Interpreter
- Vagrant
- WSL
- Docker
- Docker Compose

(●) New environment

Location: C:\Temp\arcade-games-work\venv

Base interpreter: Python 3.7 C:\Program Files (x86)\Python37-32\python.exe

[ ] Inherit global site-packages

[ ] Make available to all projects

( ) Existing environment

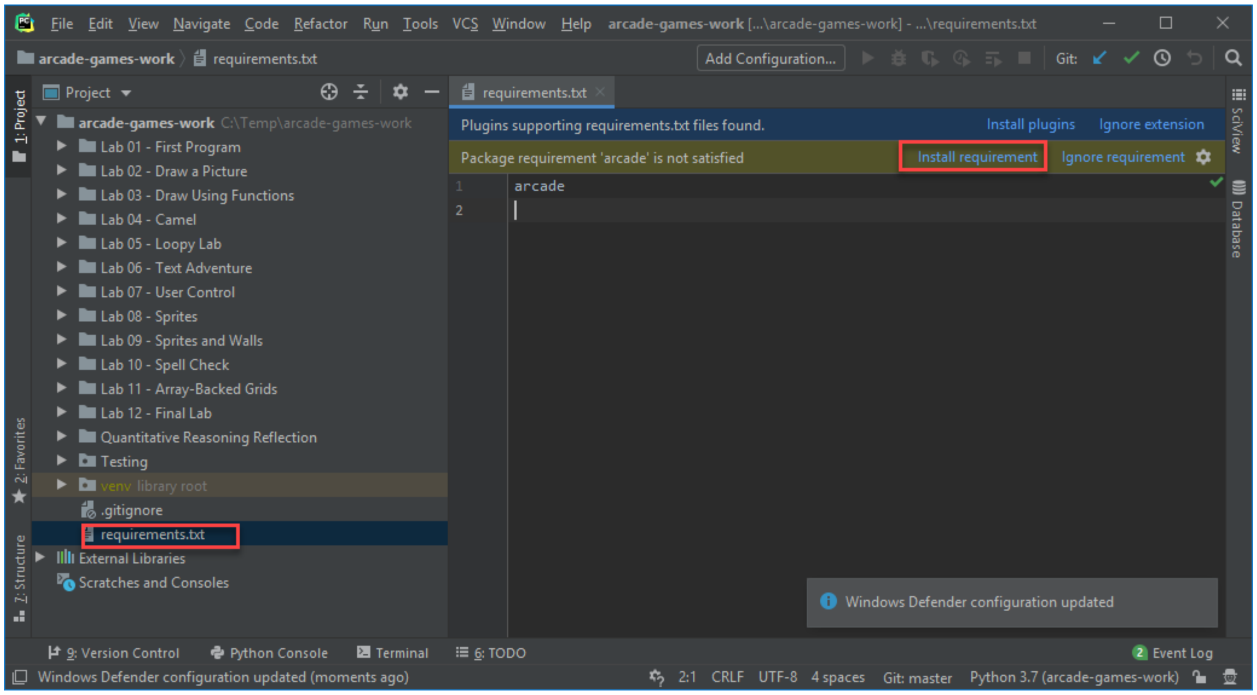
Interpreter: <No interpreter>

[ ] Make available to all projects

OK Cancel

Now you can install libraries. You can search for "Arcade" and install it.

Another way to do it is create a file called `requirements.txt` and just type `arcade` in that file. PyCharm will automatically ask any libraries in that file. It is a common way to list dependencies for Python projects.


File Edit View Navigate Code Refactor Run Tools VCS Window Help arcade-games-work [...\arcade-games-work] - ...\requirements.txt
arcade-games-work requirements.txt
Add Configuration...
Git:
Project
requirements.txt
1: Project
2: Favorites
Z: Structure
SciView
Database
9: Version Control
Python Console
Terminal
6: TODO
Event Log

Plugins supporting requirements.txt files found. Install plugins Ignore extension

Package requirement 'arcade' is not satisfied Install requirement Ignore requirement

```
arcade
```

- arcade-games-work C:\Temp\arcade-games-work
  - Lab 01 - First Program
  - Lab 02 - Draw a Picture
  - Lab 03 - Draw Using Functions
  - Lab 04 - Camel
  - Lab 05 - Loopy Lab
  - Lab 06 - Text Adventure
  - Lab 07 - User Control
  - Lab 08 - Sprites
  - Lab 09 - Sprites and Walls
  - Lab 10 - Spell Check
  - Lab 11 - Array-Backed Grids
  - Lab 12 - Final Lab
  - Quantitative Reasoning Reflection
  - Testing
  - venv library root
  - .gitignore
  - requirements.txt
- External Libraries
- Scratches and Consoles

Windows Defender configuration updated

Windows Defender configuration updated (moments ago) 2:1 CRLF UTF-8 4 spaces Git: master Python 3.7 (arcade-games-work)

Install Arcade using the command line interface

If you prefer to use the command line interface (CLI), then you can install arcade directly using pip:

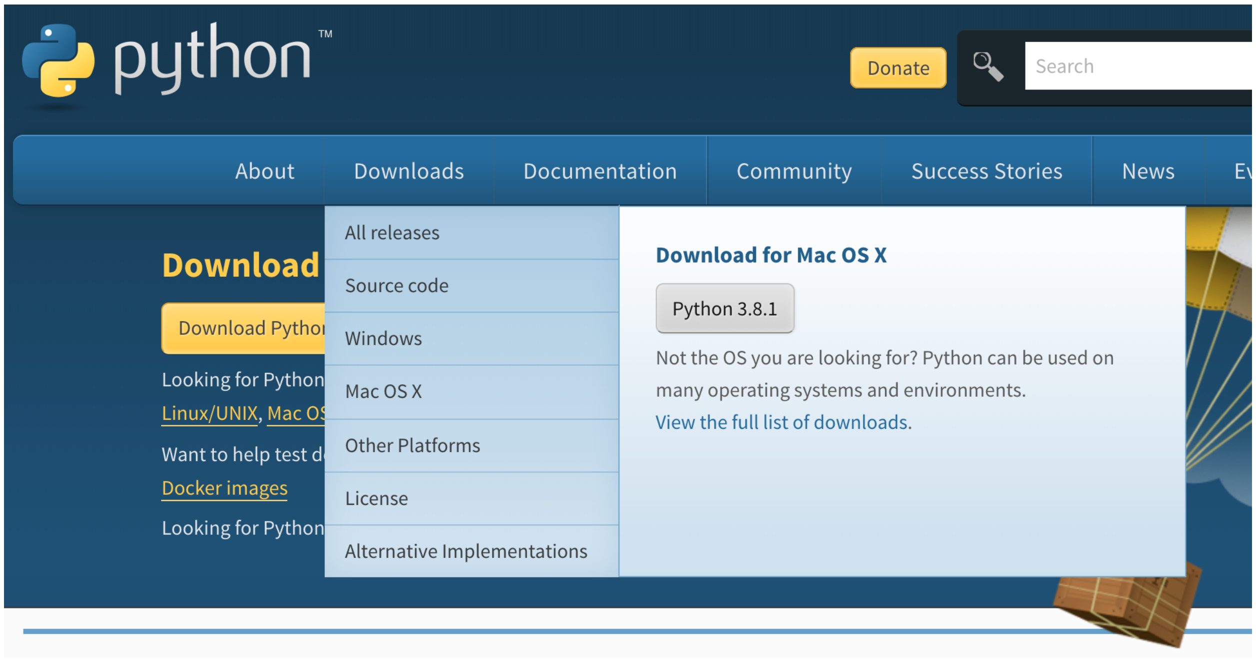
```
pip3 install arcade
```

If you happen to be using pipenv, then the appropriate command is:

```
python3 -m pipenv install arcade
```

3.2 Installation on Mac

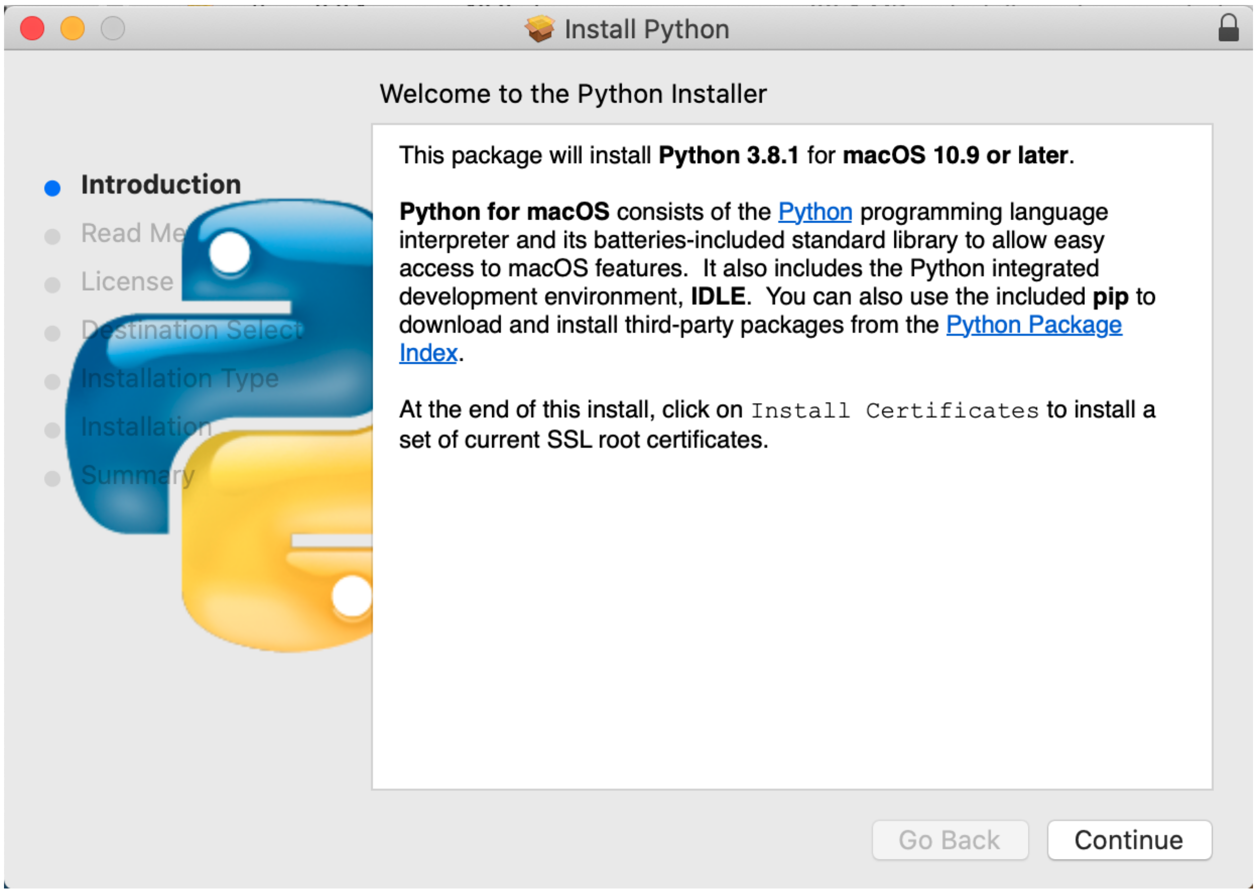
Go to the Python website and download Python.



Looking for Python
Linux/UNIX, Mac OS
Want to help test d
Docker images
Looking for Python

Not the OS you are looking for? Python can be used on many operating systems and environments.
View the full list of downloads.

Then install it:



Welcome to the Python Installer

This package will install **Python 3.8.1** for **macOS 10.9 or later**.

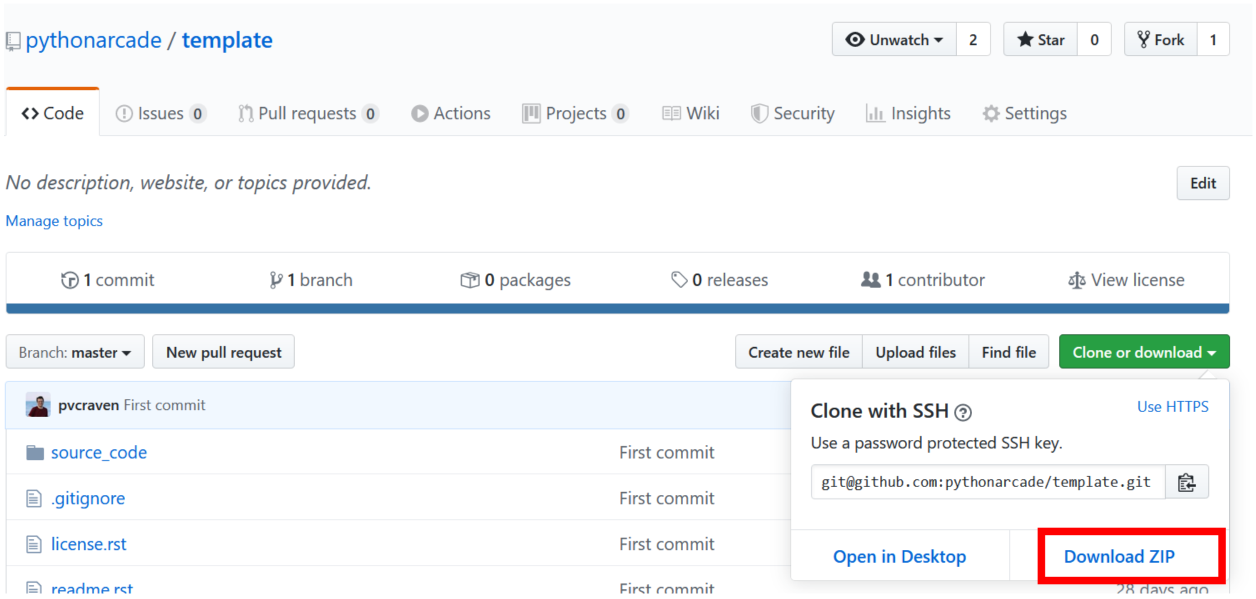
Python for macOS consists of the Python programming language interpreter and its batteries-included standard library to allow easy access to macOS features. It also includes the Python integrated development environment, **IDLE**. You can also use the included **pip** to download and install third-party packages from the Python Package Index.

At the end of this install, click on `Install Certificates` to install a set of current SSL root certificates.

Download and install PyCharm. The community edition is free, and WAY better than IDLE.

Download the zip file (or use git) for the Arcade template file.

https://github.com/pythonarcade/template



No description, website, or topics provided.

pvcraven First commit

source_code	First commit
.gitignore	First commit
license.rst	First commit
readme.rst	First commit

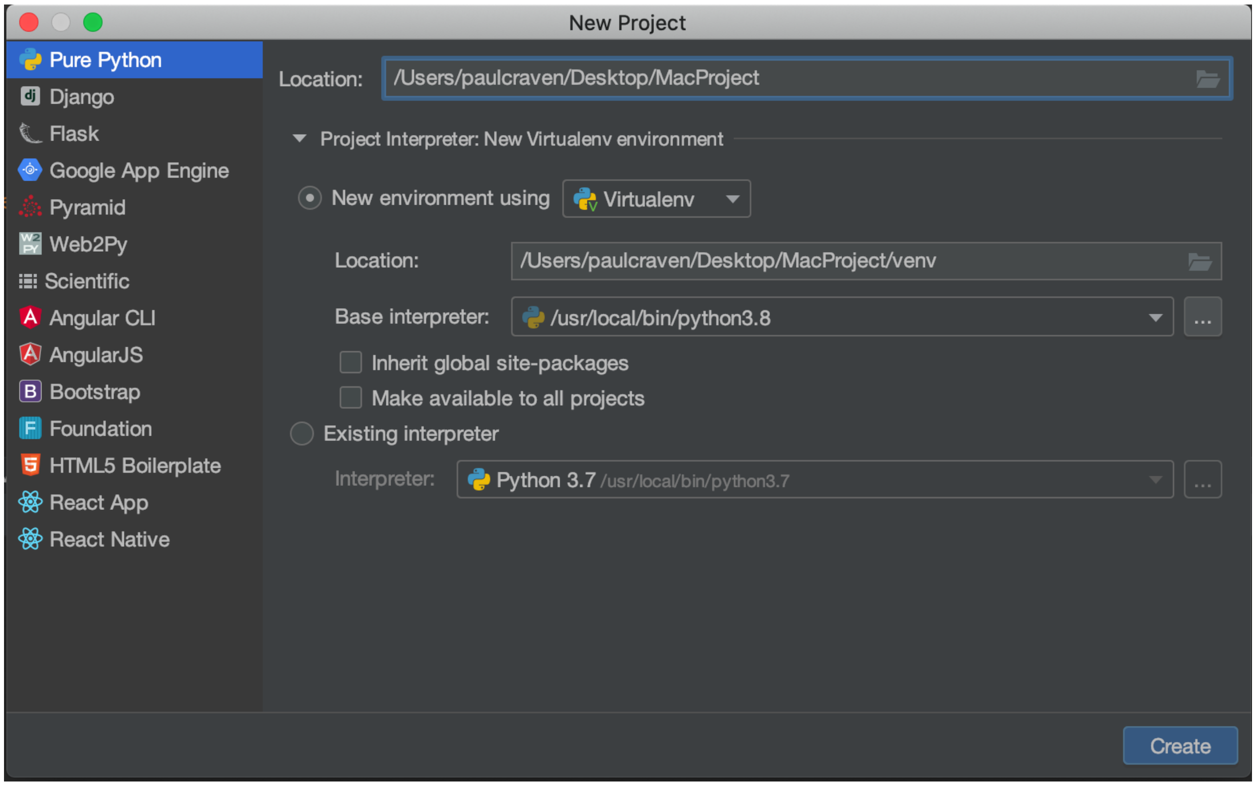
Use a password protected SSH key.

`git@github.com:pythonarcade/template.git`

After you've downloaded it, open up the zip file, and pull out the template folder to your desktop or wherever you'd like to save it. Then rename it to your project name.

Start PyCharm, and select File…Open and select the folder you just created.

When creating opening the new project, create a virtual environment like so:


New Project

Pure Python
Django
Flask
Google App Engine
Pyramid
Web2Py
Scientific
Angular CLI
AngularJS
Bootstrap
Foundation
HTML5 Boilerplate
React App
React Native

Location: /Users/paulcraven/Desktop/MacProject

Project Interpreter: New Virtualenv environment

(●) New environment using Virtualenv

Location: /Users/paulcraven/Desktop/MacProject/venv

Base interpreter: /usr/local/bin/python3.8

☐ Inherit global site-packages

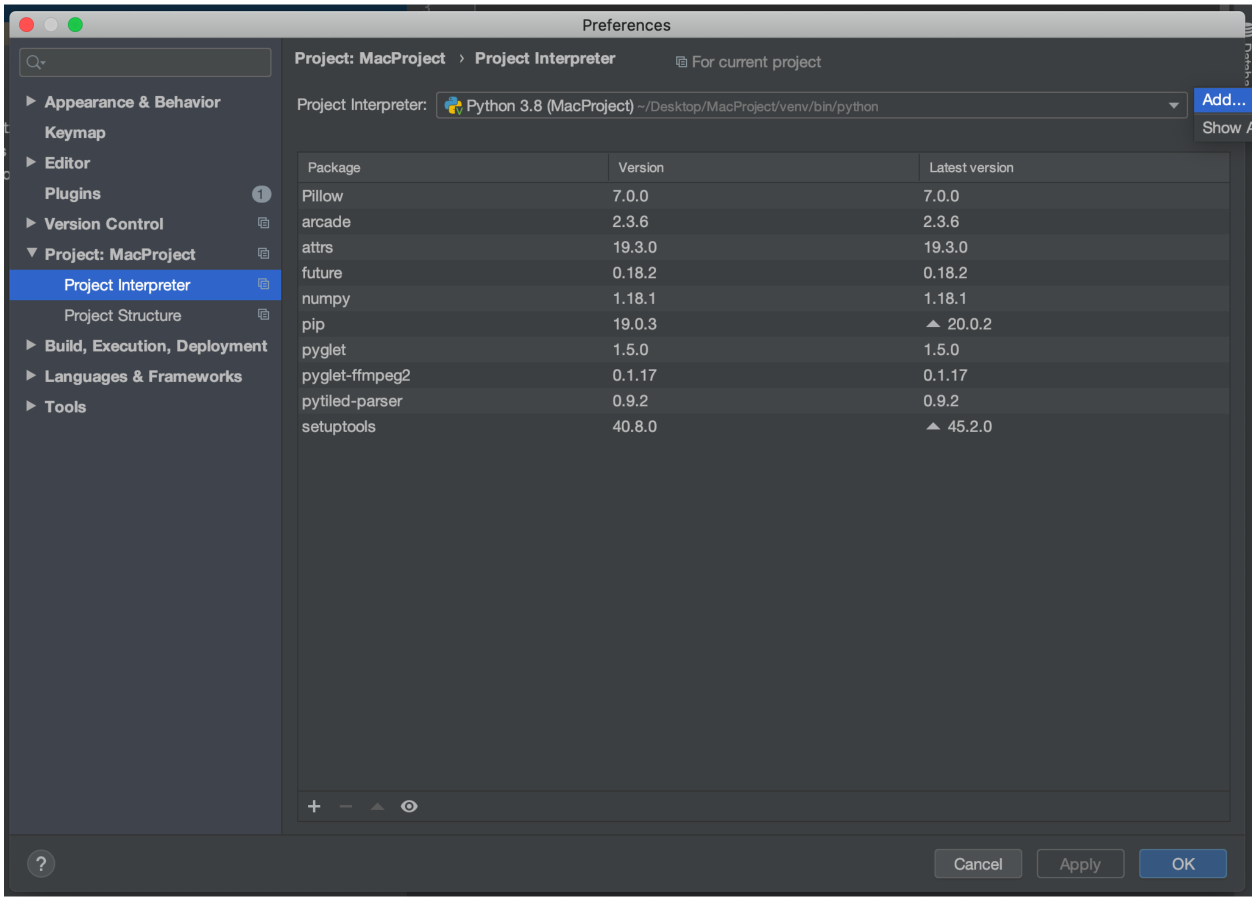
☐ Make available to all projects

( ) Existing interpreter

Interpreter: Python 3.7 /usr/local/bin/python3.7

Create

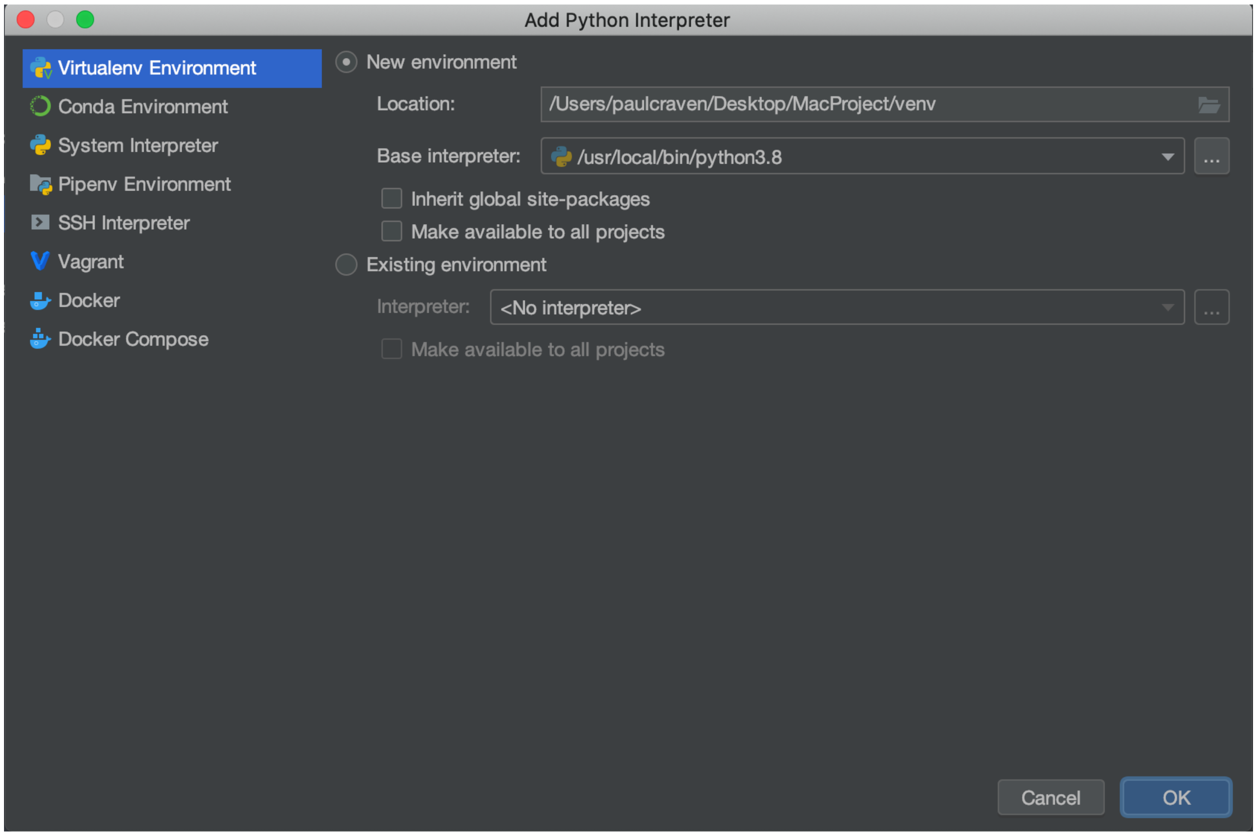
If that doesn't work, (sometimes PyCharm seems to ignore that, or maybe that step got skipped) go into PyCharm…settings, then "Project interpreter" on the right side, click the easy-to-miss gear icon and "Add"



Project Interpreter: Python 3.8 (MacProject) ~/Desktop/MacProject/venv/bin/python

Package	Version	Latest version
Pillow	7.0.0	7.0.0
arcade	2.3.6	2.3.6
attrs	19.3.0	19.3.0
future	0.18.2	0.18.2
numpy	1.18.1	1.18.1
pip	19.0.3	▲ 20.0.2
pyglet	1.5.0	1.5.0
pyglet-ffmpeg2	0.1.17	0.1.17
pytiled-parser	0.9.2	0.9.2
setuptools	40.8.0	▲ 45.2.0

…Then set it like so:



● New environment

Location: /Users/paulcraven/Desktop/MacProject/venv

Base interpreter: /usr/local/bin/python3.8

☐ Inherit global site-packages

☐ Make available to all projects

○ Existing environment

Interpreter: <No interpreter>

☐ Make available to all projects

You should get a warning at the top of the screen that 'arcade' is not installed. Go ahead and install it. Then try running the starting template.

3.2.1 Sound Support

Support for `.ogg` Ogg Vorbis files and `mp3` files can be added via HomeBrew with:

```
brew install ffmpeg
```

3.3 Installation on Linux

The Arcade library is Python 3.7+ only. First check your version of Python to ensure you have 3.7 or higher:

```
python -V
```

If your version shows Python 2.X then try running with:

```
python3 -V
```

If that works and shows you Python 3.7+, then anytime you see the `python` command, replace it with `python3`.

If you do not have Python 3.7+, please lookup how to install it for your specific distro of Linux. For Ubuntu/Debian this would be with the below command, if you did have Python 3.7, you can skip this step:

```
sudo apt install python3 python3-pip libjpeg-dev zlib1g-dev
```

Next you'll need to setup a Virtual Environment. Arcade should always be installed with a virtual environment. Installing outside of a virtual environment can lead to unintended consequences and bugs with your system. You can read more about Virtual Environments at this page: https://docs.python.org/3/tutorial/venv.html

```
python -m venv my_venv
```

This creates a new folder called `my_venv` which contains your Python virtual environment. You can now activate it with:

```
source my_venv/bin/activate
```

And deactivate it with:

```
deactivate
```

Once your venv is activated, you can install Arcade with:

```
pip install arcade
```

3.3.1 Raspberry Pi Instructions

Arcade required OpenGL graphics 3.3 or higher. Unfortunately the Raspberry Pi does not support this, Arcade can not run on the Raspberry Pi.

3.4 Installation From Source

First step is to clone the repository:

```
git clone https://github.com/pythonarcade/arcade.git
```

Or download from:

https://github.com/pythonarcade/arcade/archive/development.zip

Next, we'll create a linked install. This will allow you to change files in the arcade directory, and is great if you want to modify the Arcade library code. From the root directory of arcade type:

```
pip install -e .
```

To install additional documentation and development requirements:

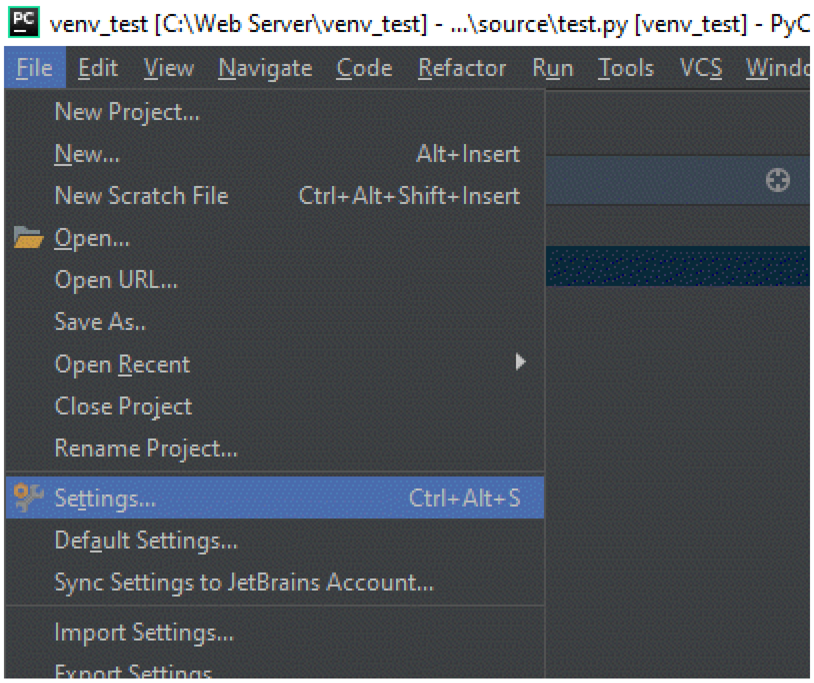
```
pip install -e .[dev]
```

3.5 Setting Up a Virtual Environment In PyCharm

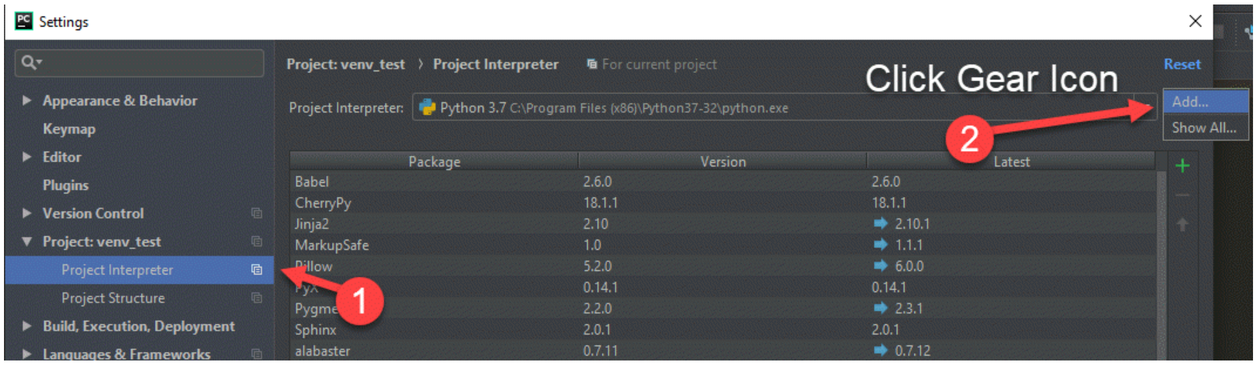
A Python virtual environment (venv) allows libraries to be installed for just a single project, rather than shared across everyone using the computer. It also does not require administrator privilages to install.

Assuming you already have a project, follow these steps to create a venv:

Step 1: Select File…Settings



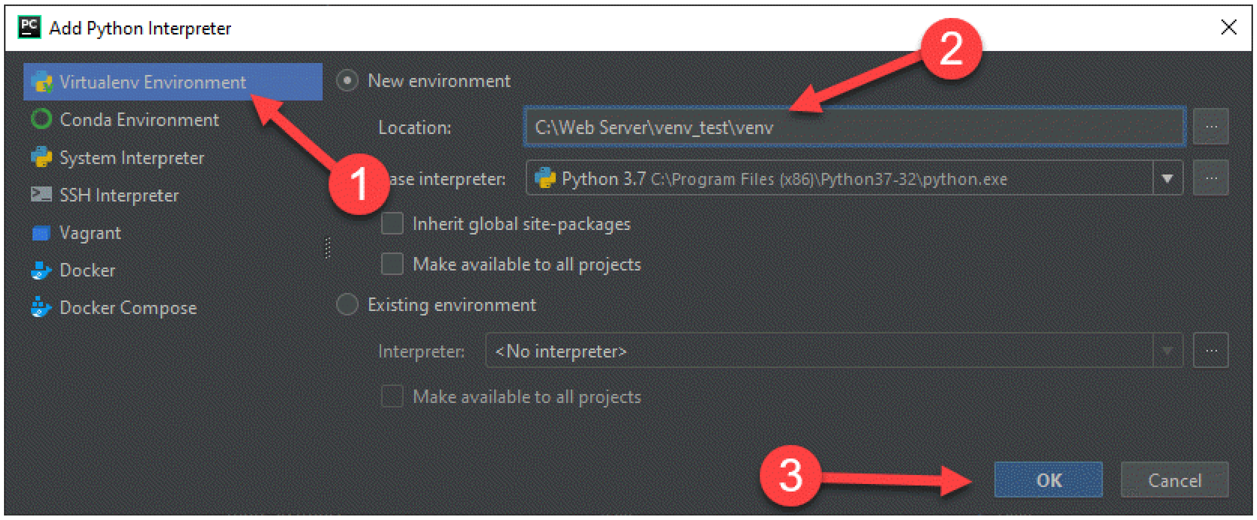
Step 2: Click "Project Interpreter". Then find the gear icon in the upper right. click on it and select "Add"



Project Interpreter: Python 3.7 C:\Program Files (x86)\Python37-32\python.exe

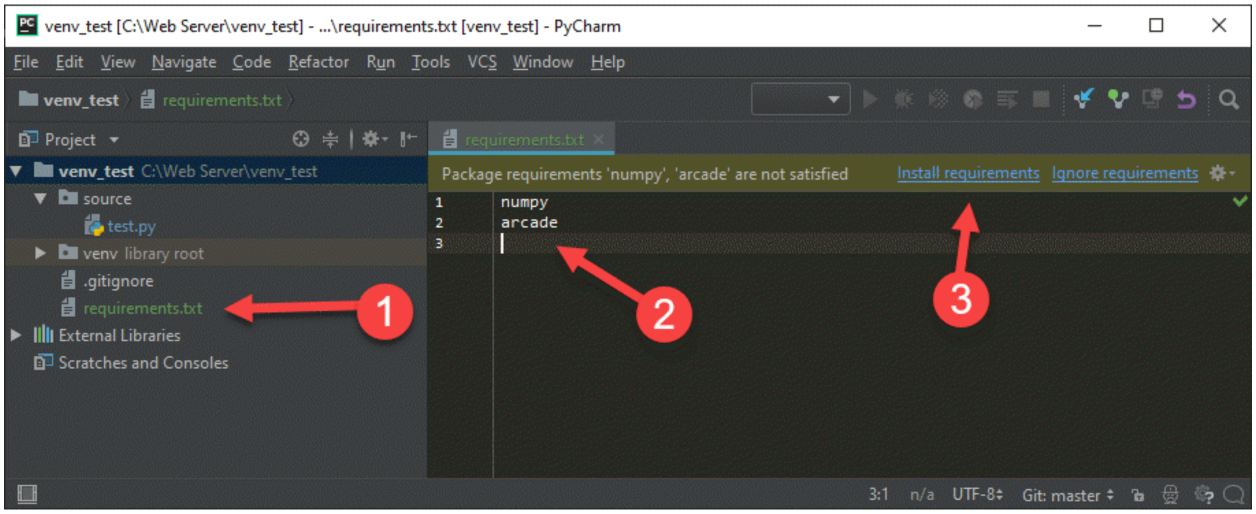
Package	Version	Latest
Babel	2.6.0	2.6.0
CherryPy	18.1.1	18.1.1
Jinja2	2.10	➡ 2.10.1
MarkupSafe	1.0	➡ 1.1.1
Pillow	5.2.0	➡ 6.0.0
PyX	0.14.1	0.14.1
Pygame	2.2.0	➡ 2.3.1
Sphinx	2.0.1	2.0.1
alabaster	0.7.11	➡ 0.7.12

Step 3: Select Virtualenv Environment from the left. Then create a new environment. Usually it should be in a folder called `venv` in your main project. PyCharm does not always select the correct location by default, so carefully look at the path to make sure it is correct, then select "Ok".


Add Python Interpreter
Virtualenv Environment
Conda Environment
System Interpreter
SSH Interpreter
Vagrant
Docker
Docker Compose
New environment
Location: C:\Web Server\venv_test\venv
Base interpreter: Python 3.7 C:\Program Files (x86)\Python37-32\python.exe
Inherit global site-packages
Make available to all projects
Existing environment
Interpreter: <No interpreter>
Make available to all projects
OK
Cancel
1
2
3

Now a virtual environment has been set up. The standard in Python projects is to create a file called `requirements.txt` and list the packages you want in there.

PyCharm will automatically ask if you want to install those packages as soon as you type them in. Go ahead and let it.


venv_test [C:\Web Server\venv_test] - ...\requirements.txt [venv_test] - PyCharm
File Edit View Navigate Code Refactor Run Tools VCS Window Help
venv_test requirements.txt
Project
venv_test C:\Web Server\venv_test
source
test.py
venv library root
.gitignore
requirements.txt
External Libraries
Scratches and Consoles
Package requirements 'numpy', 'arcade' are not satisfied
Install requirements
Ignore requirements
3:1 n/a UTF-8 Git: master
1
2
3

```
numpy
arcade
```

3.6 Installation for Obsolete Python Versions

Arcade aims to support the same Python versions currently supported by the PSF.

You are strongly encouraged to upgrade to one of the versions listed at the link above, with the exception of 3.11 or later. Some of arcade's dependencies have not yet been ported for those versions.

If you absolutely cannot upgrade to Python 3.7 or later, you can try using an older and unsupported version of Arcade.

Please remember the following:

1. Bugs will not be fixed, unless they are also present in current versions
2. The features and API may be very different from current versions
3. You will need use documentation for the version of Arcade you run

The pairings suggested below might not work. They are based on briefly skimming git history. You may have to use trial and error to look for a version that works, and it's possible that you won't find one! Here be dragons!

Obsolete Python Version	Suggested Arcade Version	Git Commit Hash
3.6	2.6.7	6e0a9af
3.5	1.2.2	078f5be

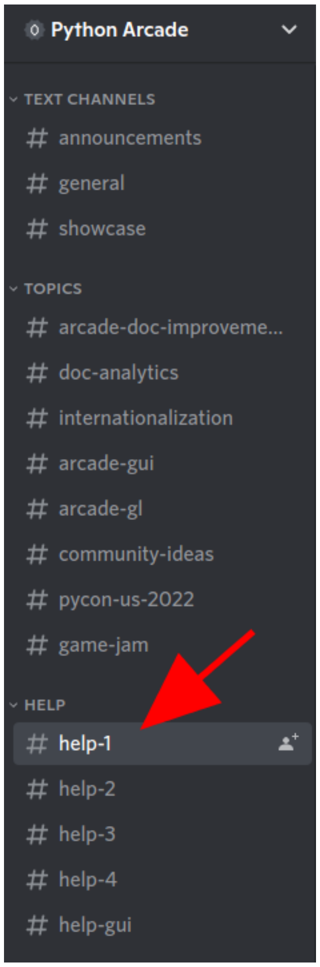
You can attempt to install these versions via the command line through pip, or by installing from source from github. Check the tags on Arcade's github page for additional commit IDs.

CHAPTER

FOUR

HOW TO GET HELP

The best places to get help are the help channels on the the Discord server. They are located in the 3rd category from the top in the channel list:



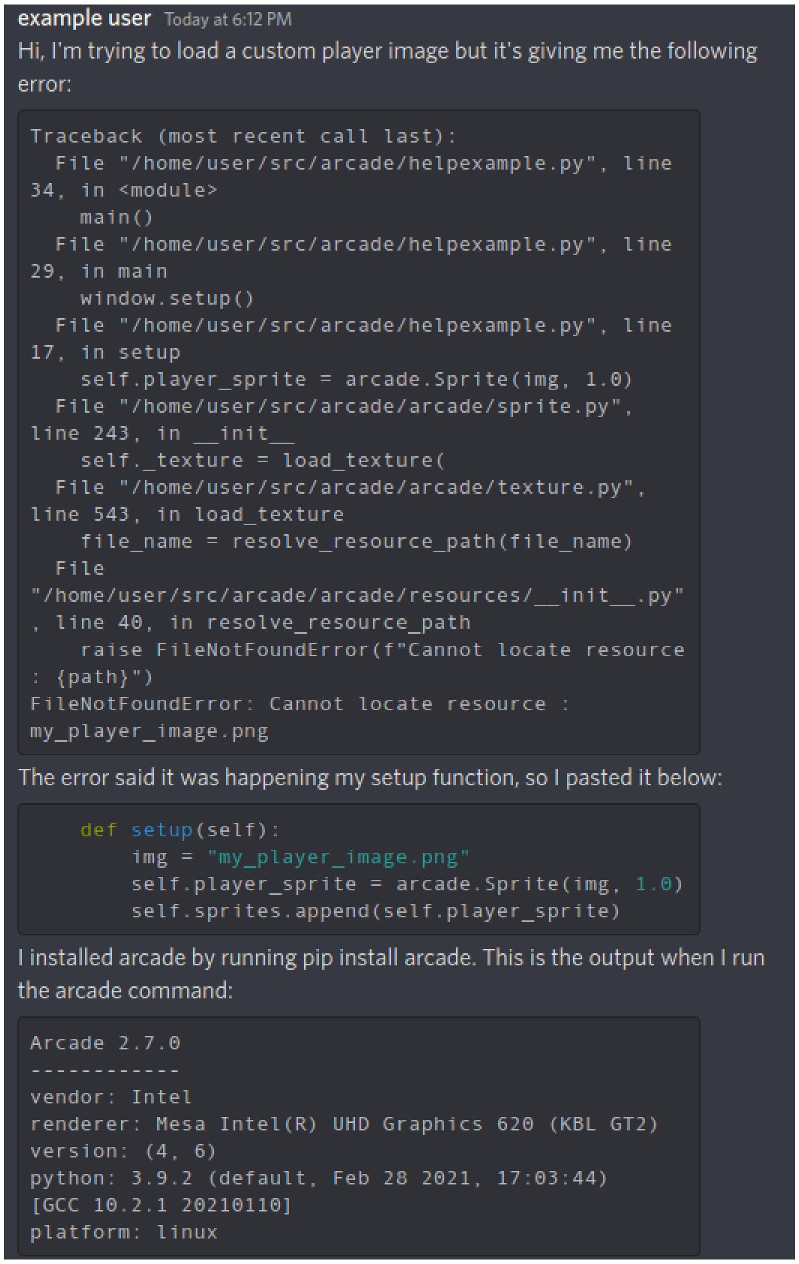
To get help, start by choosing an inactive help channel. Inactive means that the last message was sent a day or more ago. If all the help channels have been active in that time, choose the one in with the earliest last message.

Once you have chosen a channel, do your best to provide the following information:

1. A very short explanation of what you're trying to do
2. The problem you're having, with any *error output formatted properly*

3. Your code, with *proper formatting*
4. Which *version of arcade* you're using and how you installed it

Here's an example as a series of Discord messages (click or tap to enlarge):



example user Today at 6:12 PM

Hi, I'm trying to load a custom player image but it's giving me the following error:

```
Traceback (most recent call last):
  File "/home/user/src/arcade/helpexample.py", line
34, in <module>
    main()
  File "/home/user/src/arcade/helpexample.py", line
29, in main
    window.setup()
  File "/home/user/src/arcade/helpexample.py", line
17, in setup
    self.player_sprite = arcade.Sprite(img, 1.0)
  File "/home/user/src/arcade/arcade/sprite.py",
line 243, in __init__
    self._texture = load_texture(
  File "/home/user/src/arcade/arcade/texture.py",
line 543, in load_texture
    file_name = resolve_resource_path(file_name)
  File
"/home/user/src/arcade/arcade/resources/__init__.py"
, line 40, in resolve_resource_path
    raise FileNotFoundError(f"Cannot locate resource
: {path}")
FileNotFoundError: Cannot locate resource :
my_player_image.png
```

The error said it was happening my setup function, so I pasted it below:

```
    def setup(self):
        img = "my_player_image.png"
        self.player_sprite = arcade.Sprite(img, 1.0)
        self.sprites.append(self.player_sprite)
```

I installed arcade by running pip install arcade. This is the output when I run the arcade command:

```
Arcade 2.7.0
------------
vendor: Intel
renderer: Mesa Intel(R) UHD Graphics 620 (KBL GT2)
version: (4, 6)
python: 3.9.2 (default, Feb 28 2021, 17:03:44)
[GCC 10.2.1 20210110]
platform: linux
```

The rest of this page will explain how to format your messages like the example above.

4.1 Sharing & Formatting Your Code

Other people need to be able to see your code to help you. There are two preferred ways of showing it to them:

1. *Pasting into Discord* for small amounts of code
2. *Using a code hosting service* for 1 or more files

4.1.1 Formatting for Discord & Github Issues

It is important to format code and terminal output when posting it. Formatting helps other people understand what you've pasted.

Both Discord & GitHub issues use the same 3 steps below.

Step 1 : Find your Backtick Key

The ` characters below are not single quotes or apostrophes. They're called backticks.

On standard US keyboards, the backtick key is the same one used to type a tilda (~). You can find it to the left of the 1 key.

For other keyboard layouts, please see this StackExchange answer.

Step 2: Format & Paste

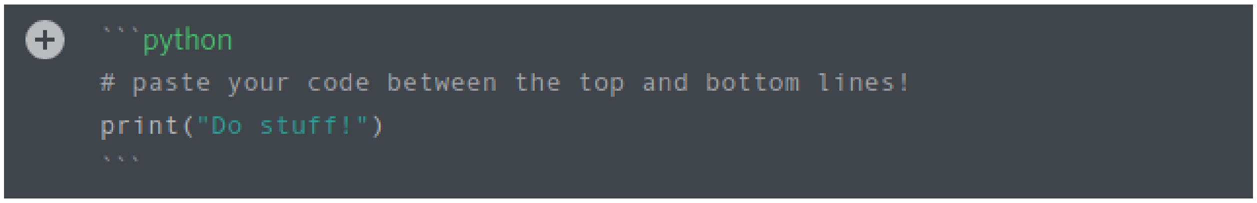
Formatting Python code is nearly identical to formatting terminal output.

Formatting Code

Once you have found your backtick key, you can format your code like this:

~~~
```python
# paste your code between the top and bottom lines!
print("Do stuff!")
```
~~~

If you cannot type a backtick on your keyboard, you can copy the example above to your clipboard. For convenience, clicking the icon at the top right of the example box will copy it for you. You can paste it into Discord's message box as shown below:



~~~
```python
# paste your code between the top and bottom lines!
print("Do stuff!")
```
~~~

Formatting Terminal Output

Terminal output, such as error traceback, can be formatted in almost the exact same way. The difference is that you don't type `python` after the three backticks on the first line:

~~~
```
Traceback (most recent call last):
  File "/home/user/src/arcade/helpexample.py", line 34, in <module>
    main()
  File "/home/user/src/arcade/helpexample.py", line 29, in main
    window.setup()
  File "/home/user/src/arcade/helpexample.py", line 17, in setup
    self.player_sprite = arcade.Sprite(img, 1.0)
  File "/home/user/src/arcade/arcade/sprite.py", line 243, in __init__
    self._texture = load_texture(
  File "/home/user/src/arcade/arcade/texture.py", line 543, in load_texture
    file_name = resolve(file_name)
  File "/home/user/src/arcade/arcade/resources/__init__.py", line 40, in resolve
    raise FileNotFoundError(f"Cannot locate resource : {path}")
FileNotFoundError: Cannot locate resource : my_player_image.png
```
~~~

Step 3: Post it!

On Discord, you can now press enter to send your message like any other formatted text.

For reporting bugs on GitHub, the same general formatting principles apply, but with a few differences.

You will also have to click Submit new issue instead of pressing enter. Please see the following links for more information on reporting bugs, GitHub issues, and their supported markdown syntax:

- How to Report Bugs Effectively
- GitHub issue creation documentation
- GitHub general markdown guide
- GitHub's code formatting documentation

4.1.2 Code Hosting

Code hosting services provide a formatted web view of your code which you can share with a link.

To share code snippets or single files without a signup, you can use the code pasting service provided by the Python Discord. If you're ok with signing up for something, there are also GitHub Gists. Afterwards, you can paste a link in Discord or another chat application.

A more advanced way to share code is to use a git hosting service. It takes effort to learn how to use git, but it has many benefits. Some of them include:

- Easy backup & undo
- Easier collaboration with others
- Allow people to view your entire project's source to help you better

Popular Git hosting options include:

- GitHub
- GitLab

4.2 Arcade Version & Basic Environment Info

This section assumes you have *installed arcade* and activated your virtual environment.

To get basic information about your current arcade version and environment, run this from within your development environment:

```
arcade
```

The command is cross-platform, which means it should work the same way regardless of whether you're on Mac, Linux, or Windows.

The output should should look something like this:

```
Arcade 2.7.0
------------
vendor: Intel
renderer: Mesa Intel(R) UHD Graphics 620 (KBL GT2)
version: (4, 6)
python: 3.9.2 (default, Feb 28 2021, 17:03:44)
[GCC 10.2.1 20210110]
platform: linux
```

It's ok if the output looks different from the example above. The second half of each line may change to reflect your arcade version, hardware, and operating system.

You can copy and paste the output into Discord or GitHub using the markdown formatting for terminal output described earlier.

Output like the example below means that something is wrong:

```
bash: arcade: command not found
```

You should still include the output as part of a request for help.

If you want to try fixing the problem yourself before getting help, the likeliest explanations for the error message above are:

- Forgetting to activate your virtual environment
- Not *installing arcade* successfully

CHAPTER

FIVE

HOW-TO EXAMPLE CODE

5.1 Starting Templates

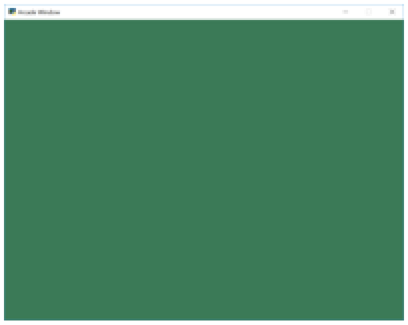


Fig. 1: starting_template

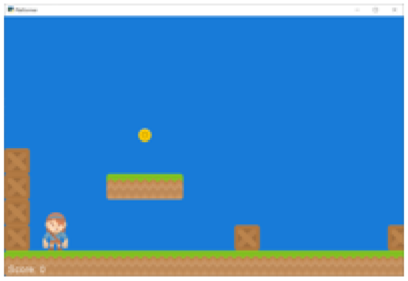


Fig. 2: template_platformer

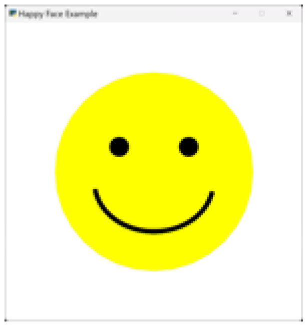


Fig. 3: happy_face

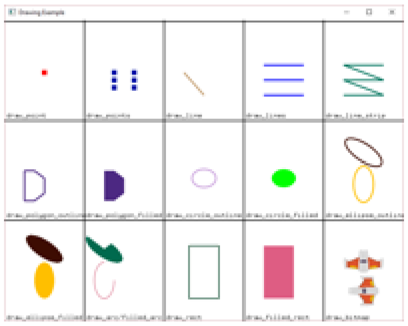
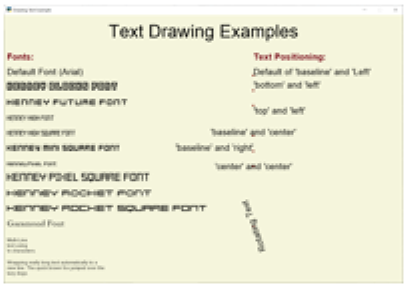


Fig. 4: drawing_primitives

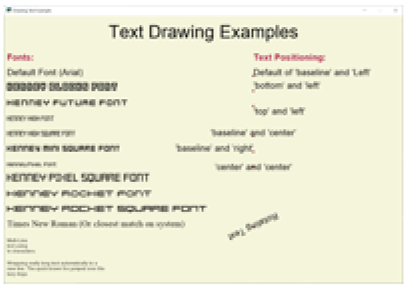


Text Drawing Examples

Fonts:
Default Font (Arial)
[illegible]
KENNEY FUTURE FONT
[illegible]
[illegible]
KENNEY MINI SQUARE FONT
[illegible]
KENNEY PIXEL SQUARE FONT
KENNEY ROCKET FONT
KENNEY ROCKET SQUARE FONT
Garamond Font
[illegible]

Text Positioning:
Default of 'baseline' and 'Left'
'bottom' and 'left'
'top' and 'left'
'baseline' and 'center'
'baseline' and 'right'
'center' and 'center'
Rotating Text

Fig. 5: drawing_text



Text Drawing Examples

Fonts:
Default Font (Arial)
[illegible]
KENNEY FUTURE FONT
[illegible]
[illegible]
KENNEY MINI SQUARE FONT
[illegible]
KENNEY PIXEL SQUARE FONT
KENNEY ROCKET FONT
KENNEY ROCKET SQUARE FONT
Times New Roman (Or closest match on system)
[illegible]

Text Positioning:
Default of 'baseline' and 'Left'
'bottom' and 'left'
'top' and 'left'
'baseline' and 'center'
'baseline' and 'right'
'center' and 'center'
Rotating Text

Fig. 6: drawing_text_objects

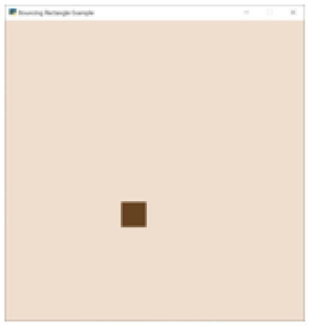


Fig. 7: bouncing_rectangle

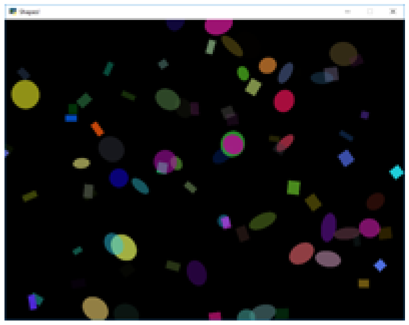


Fig. 8: shapes-slow

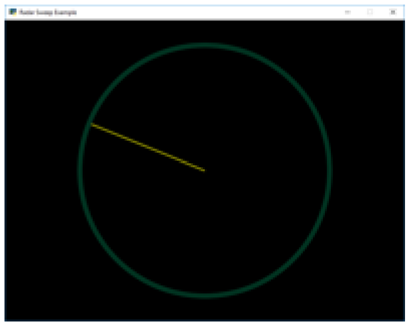


Fig. 9: radar_sweep

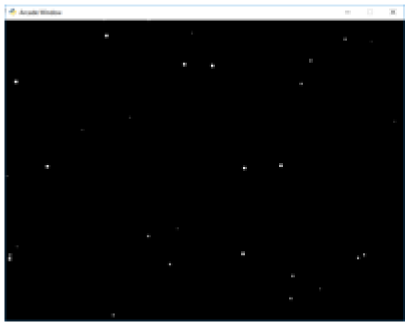


Fig. 10: snow

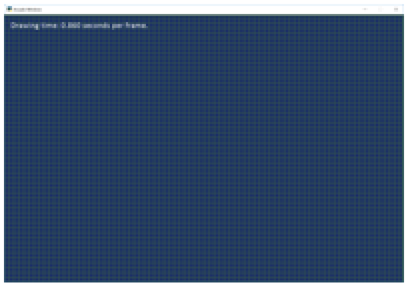


Fig. 11: shape_list_demo

Fig. 12: lines_buffered

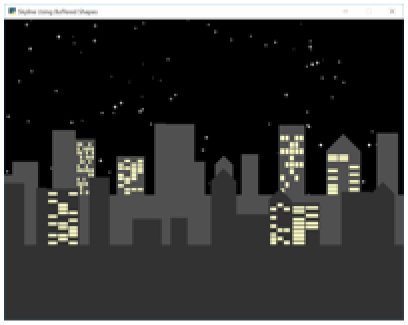


Fig. 13: shape_list_demo_skylines

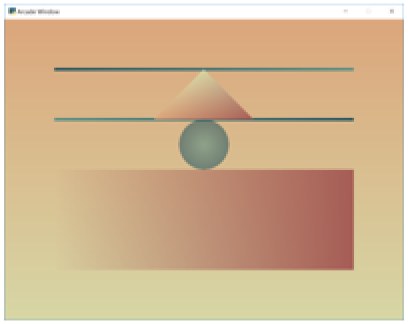


Fig. 14: gradients

5.2 Drawing

5.2.1 Drawing Primitives

5.2.2 Animating Drawing Primitives

5.2.3 Faster Drawing with ShapeElementLists

5.3 Sprites

5.3.1 Player Movement

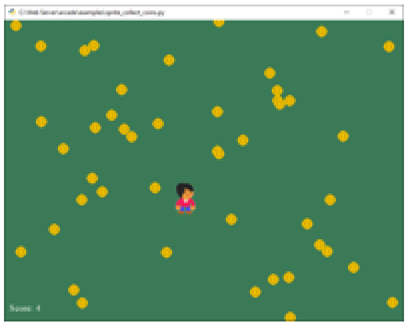


Fig. 15: sprite_collect_coins

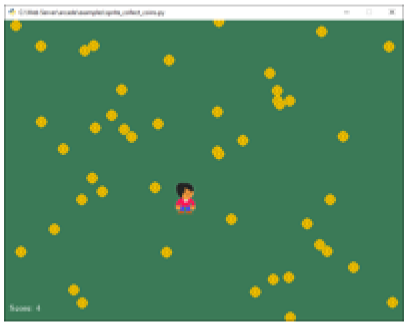


Fig. 16: sprite_move_keyboard

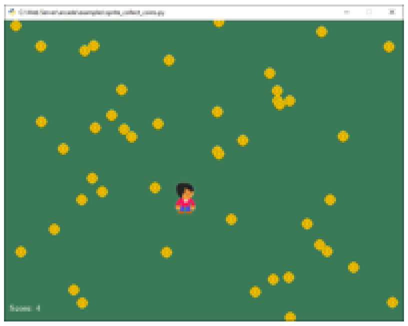


Fig. 17: sprite_move_keyboard_better

Fig. 18: sprite_move_keyboard_accel

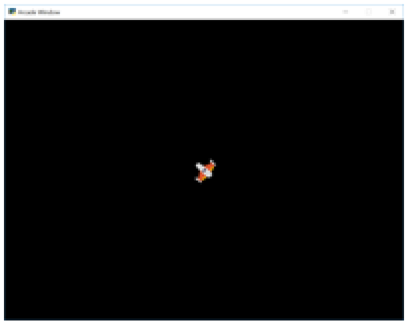


Fig. 19: sprite_move_angle

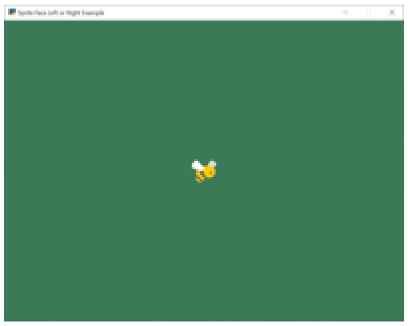


Fig. 20: sprite_face_left_or_right

Fig. 21: sprite_move_controller

Fig. 22: dual_stick_shooter

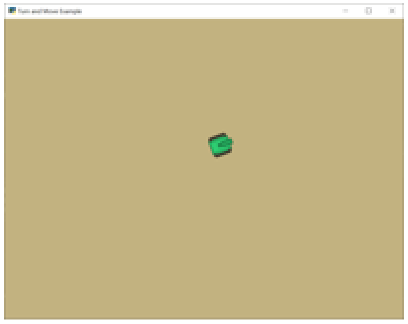
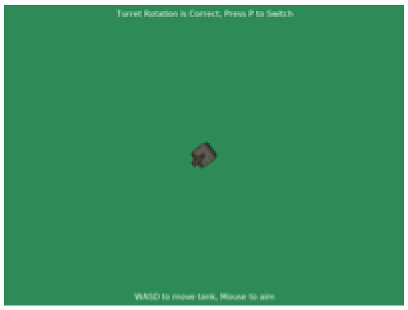


Fig. 23: turn_and_move



Turret Rotation is Correct, Press P to Switch

WASD to move tank, Mouse to aim

Fig. 24: sprite_rotate_around_tank

5.3.2 Non-Player Movement

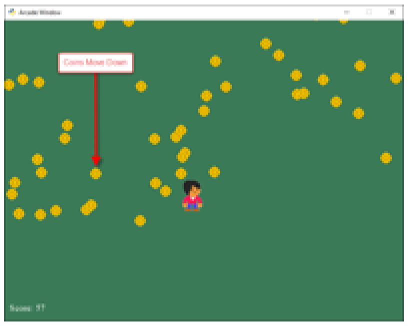


Fig. 25: sprite_collect_coins_move_down

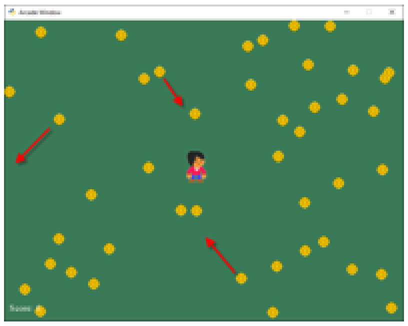


Fig. 26: sprite_collect_coins_move_bouncing

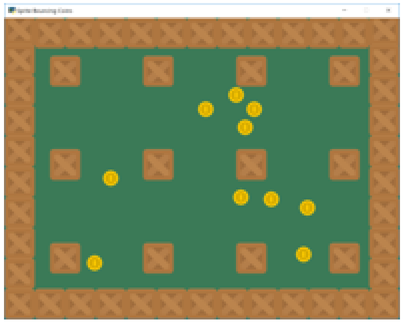


Fig. 27: sprite_bouncing_coins

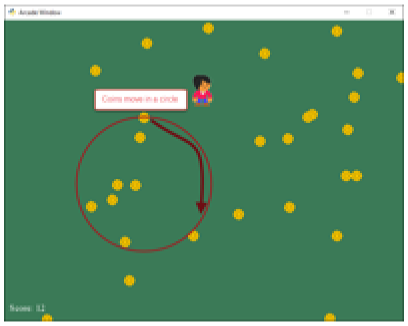


Fig. 28: sprite_collect_coins_move_circle

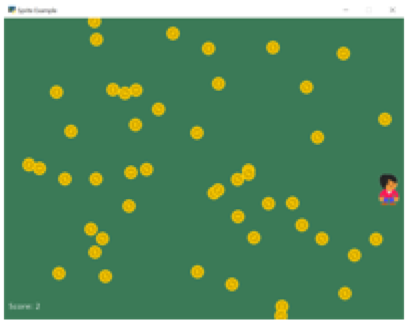


Fig. 29: sprite_collect_rotating

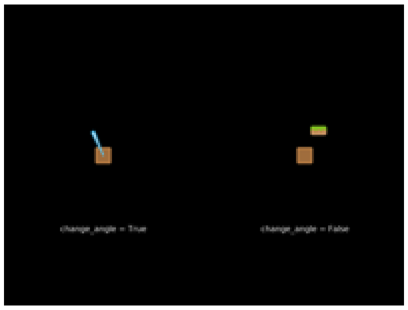


Fig. 30: sprite_rotate_around_point

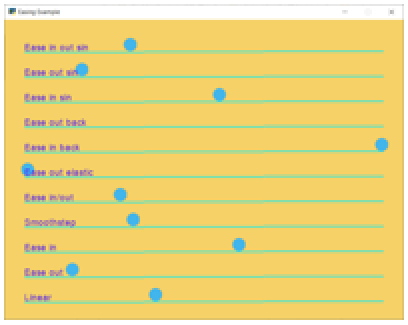


Fig. 31: easing_example_1

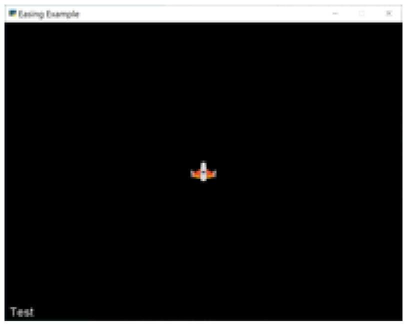


Fig. 32: easing_example_2

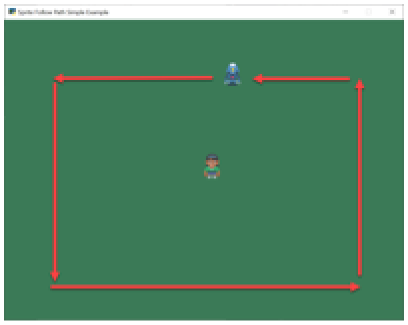


Fig. 33: follow_path

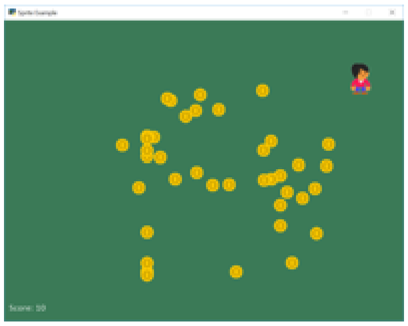


Fig. 34: sprite_follow_simple

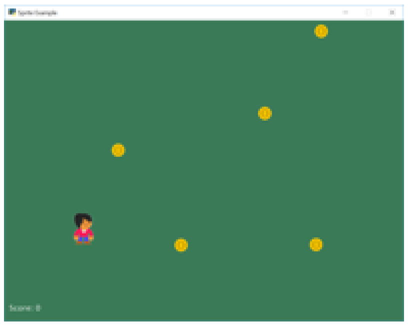


Fig. 35: sprite_follow_simple_2

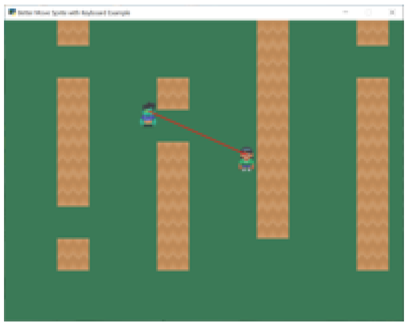


Fig. 36: line_of_sight

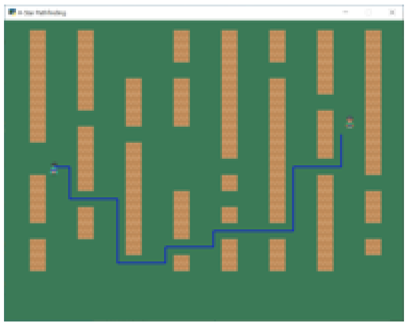


Fig. 37: astar_pathfinding

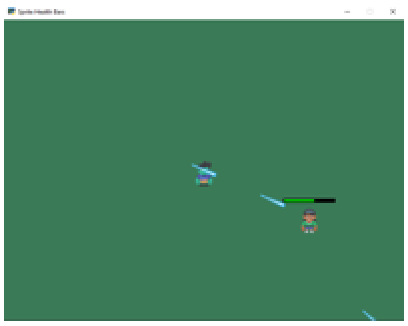
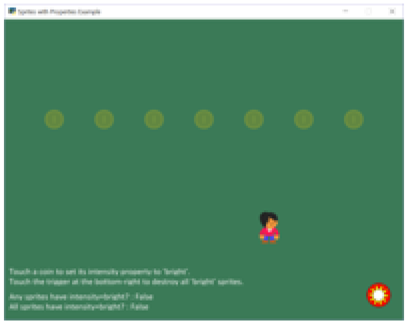


Fig. 38: sprite_health



Touch a coin to set its intensity property to 'bright'.
Touch the trigger at the bottom right to destroy all 'bright' sprites.

Any sprites have intensity=bright? : False
All sprites have intensity=bright? : False

Fig. 39: sprite_properties

Fig. 40: sprite_change_coins

Fig. 41: example-sprite-collect-coins-diff-levels

Fig. 42: sprite_rooms

Fig. 43: sprite_bullets

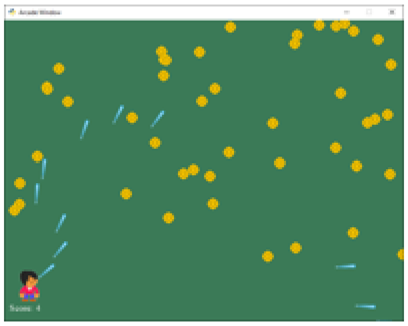


Fig. 44: sprite_bullets_aimed

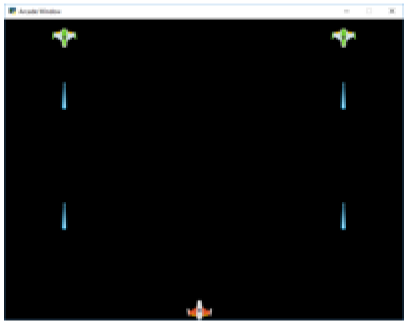


Fig. 45: sprite_bullets_periodic

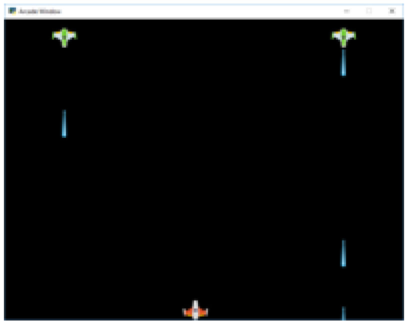


Fig. 46: sprite_bullets_random

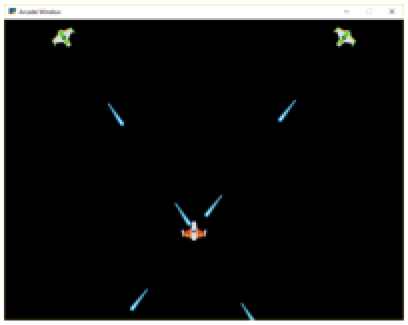


Fig. 47: sprite_bullets_enemy_aims

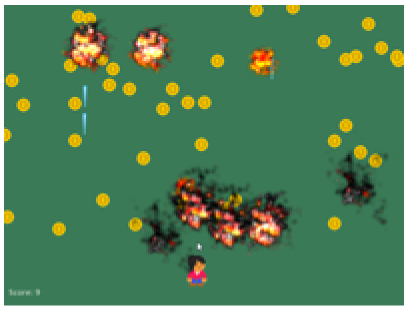


Fig. 48: sprite_explosion_bitmapped

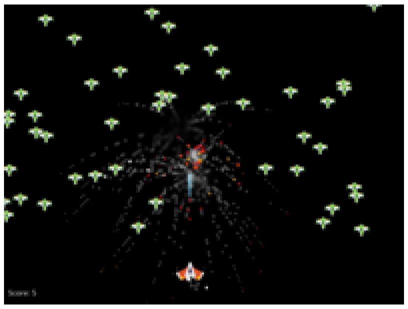


Fig. 49: sprite_explosion_particles

5.3.3 Easing

5.3.4 Calculating a Path

5.3.5 Sprite Properties

5.3.6 Games with Levels

5.3.7 Shooting with Sprites

5.4 Audio

5.4.1 Sound Effects

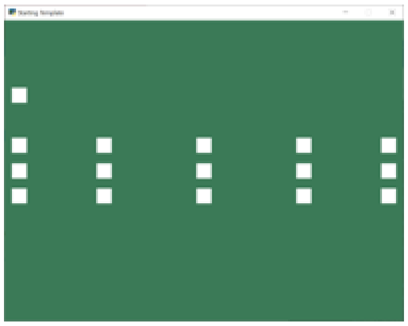


Fig. 50: sound_demo

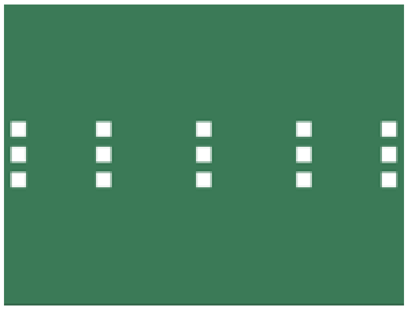


Fig. 51: sound_speed_demo

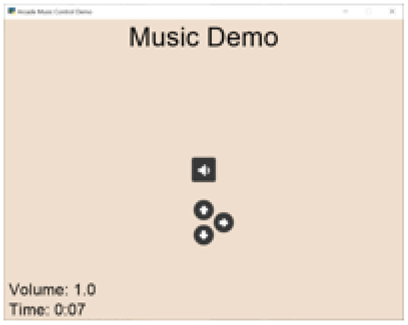


Fig. 52: music_control_demo

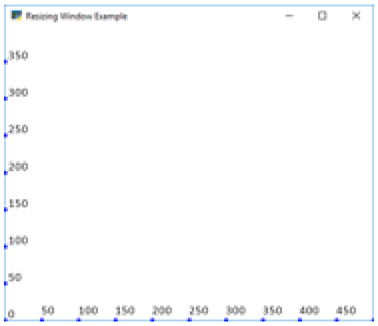


Fig. 53: resizable_window

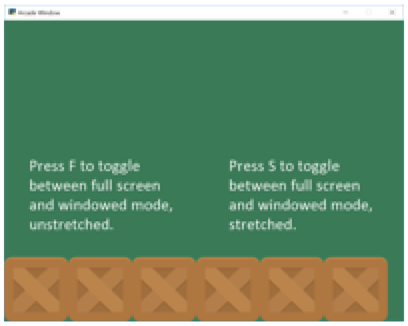

Press F to toggle between full screen and windowed mode, unstretched.
Press S to toggle between full screen and windowed mode, stretched.

Fig. 54: full_screen_example

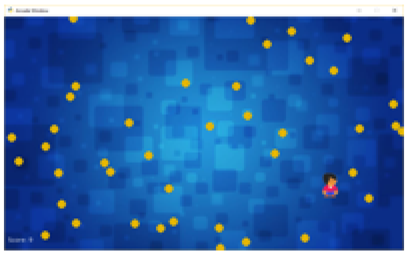


Fig. 55: sprite_collect_coins_background

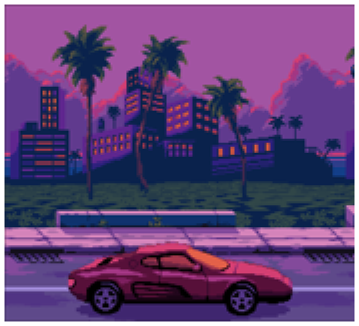


Fig. 56: background_parallax

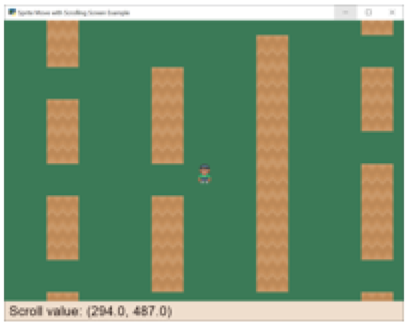
5.4.2 Music

5.5 Display Management

5.5.1 Resizable Windows

5.5.2 Backgrounds

5.5.3 Cameras



Scroll value: (294.0, 487.0)

Fig. 57: sprite_move_scrolling

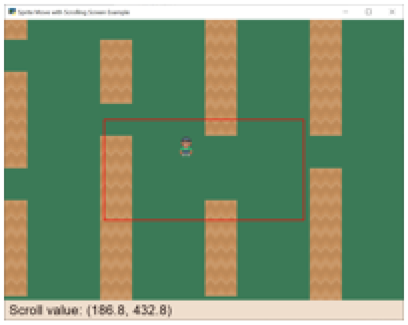


Fig. 58: sprite_move_scrolling_box

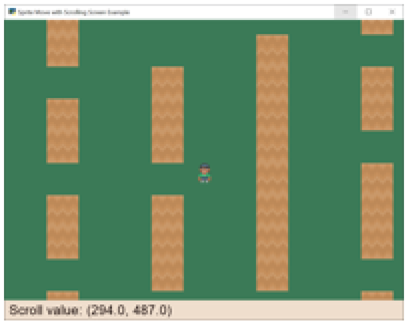


Fig. 59: sprite_move_scrolling_shake

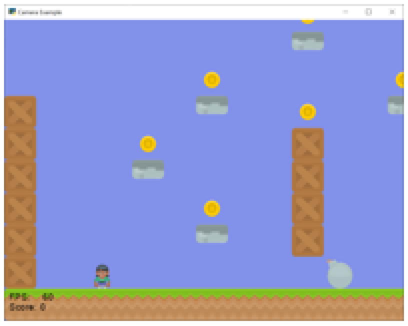
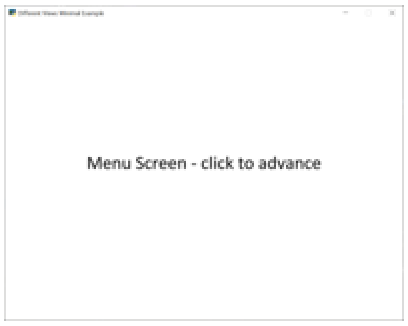


Fig. 60: camera_platform

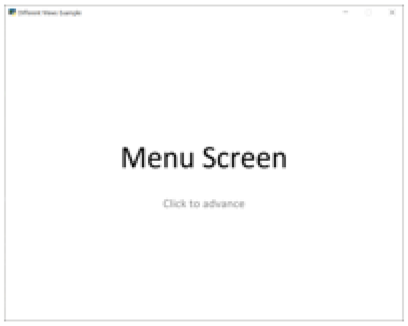
5.6 View Management

5.6.1 Instruction and Game Over Screens



Menu Screen - click to advance

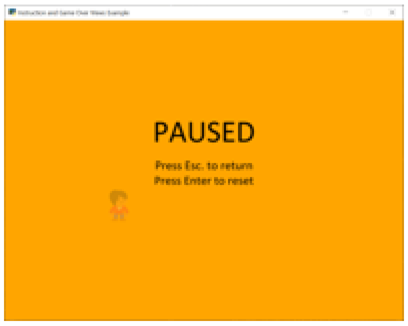
Fig. 61: view_screens_minimal



Menu Screen

Click to advance

Fig. 62: view_instructions_and_game_over



PAUSED

Press Esc. to return
Press Enter to reset

Fig. 63: view_pause_screen

Fig. 64: transitions

5.6.2 Sectioning a View

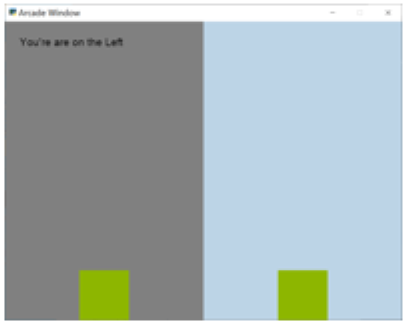


Fig. 65: sections_demo_1

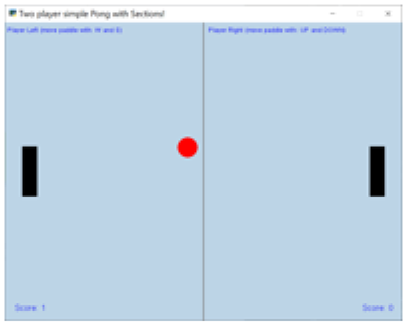


Fig. 66: sections_demo_2

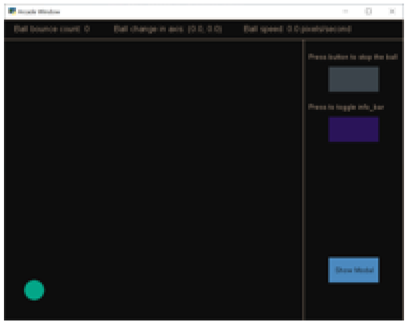


Fig. 67: sections_demo_3

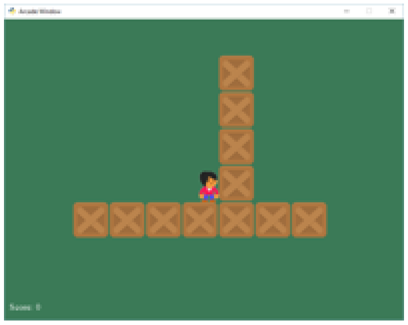


Fig. 68: sprite_move_walls

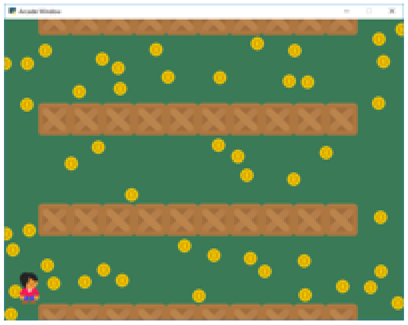


Fig. 69: sprite_no_coins_on_walls

Fig. 70: sprite_move_animation

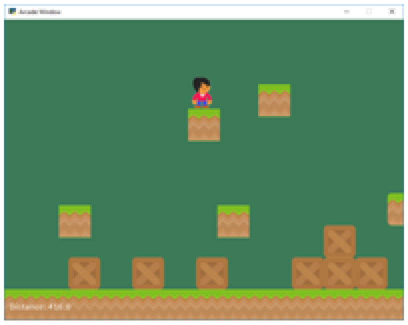


Fig. 71: sprite_moving_platforms

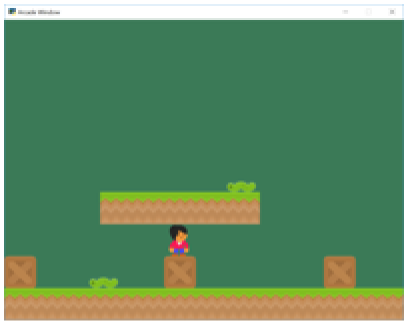


Fig. 72: sprite_enemies_in_platformer

Fig. 73: *Simple Platformer*

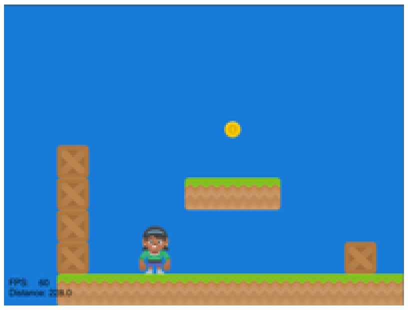


Fig. 74: sprite_tiled_map

Fig. 75: sprite_tiled_map_with_levels

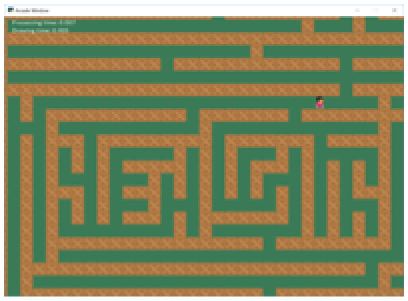


Fig. 76: maze_recursive

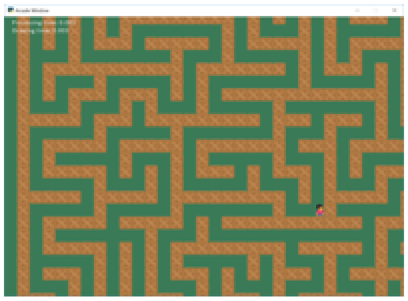


Fig. 77: maze_depth_first

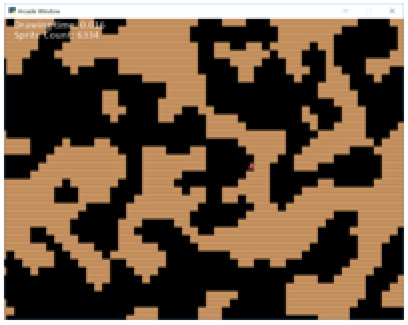


Fig. 78: procedural_caves_cellular

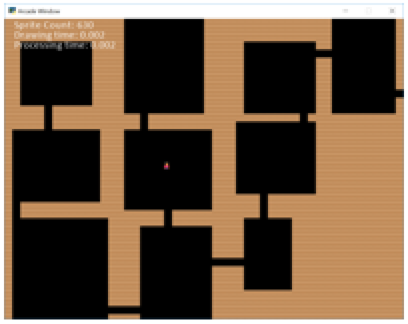


Fig. 79: procedural_caves_bsp

5.7 Platformers

5.7.1 Basic Platformers

5.7.2 Tiled Map Editor

5.7.3 Procedural Generation

5.8 Graphical User Interface

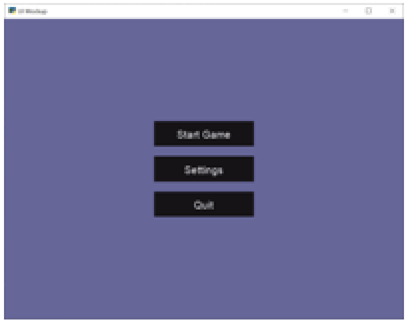


Fig. 80: gui_flat_button

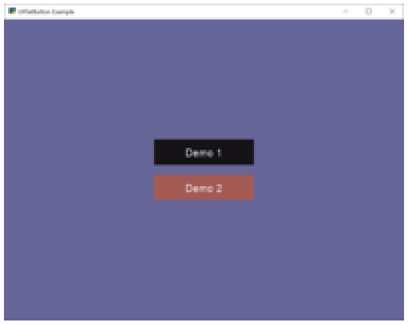
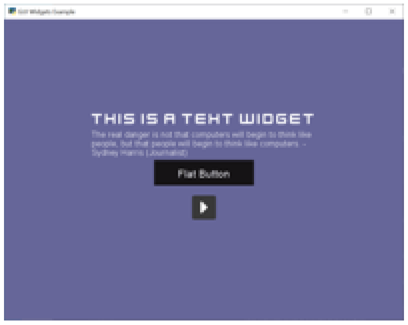
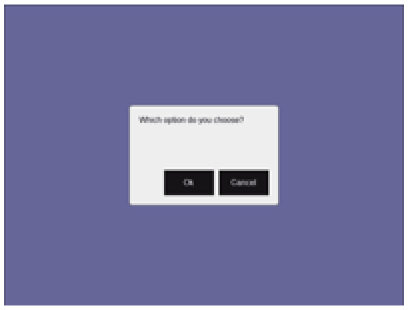


Fig. 81: gui_flat_button_styled



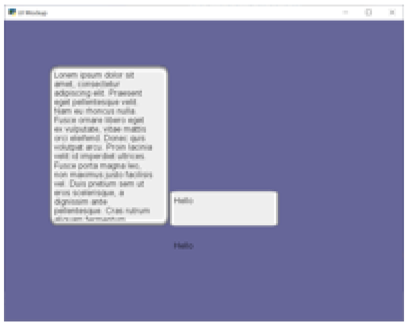
The real danger is not that computers will begin to think like people, but that people will begin to think like computers. – Sydney Harris (Journalist)

Fig. 82: gui_widgets



Which option do you choose?

Fig. 83: gui_ok_messagebox



Lorem ipsum dolor sit
amet, consectetur
adipiscing elit. Praesent
eget pellentesque velit.
Nam eu rhoncus nulla.
Fusce ornare libero eget
ex vulputate, vitae mattis
orci eleifend. Donec quis
volutpat arcu. Proin lacinia
velit id imperdiet ultrices.
Fusce porta magna leo,
non maximus justo facilisis
vel. Duis pretium sem ut
eros scelerisque, a
dignissim ante
pellentesque. Cras rutrum
aliquam fermentum.

Hello

Hello

Fig. 84: gui_scrollable_text

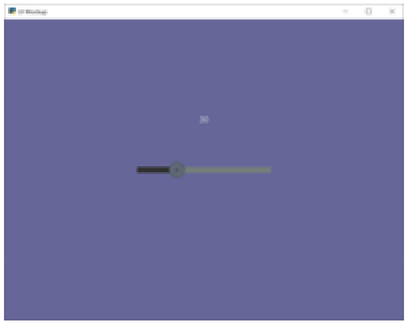


Fig. 85: gui_slider

5.9 Grid-Based Games

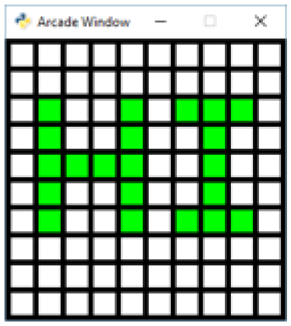


Fig. 86: array_backed_grid

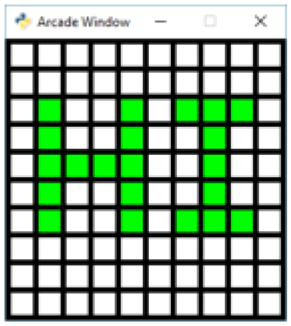


Fig. 87: array_backed_grid_buffered

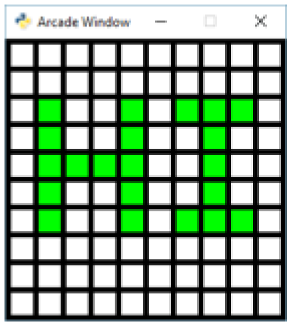


Fig. 88: array_backed_grid_sprites_1

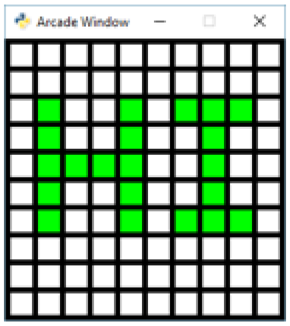


Fig. 89: array_backed_grid_sprites_2

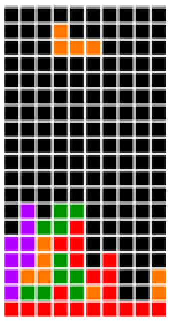


Fig. 90: tetris

Fig. 91: conway_alpha

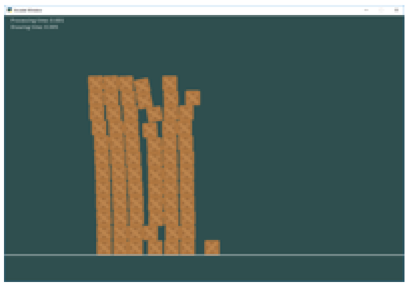


Fig. 92: pymunk_box_stacks

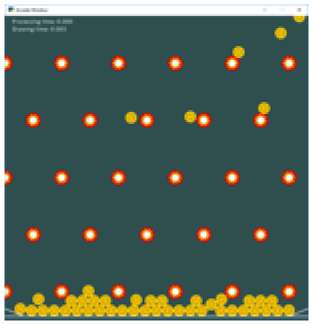


Fig. 93: pymunk_pegboard

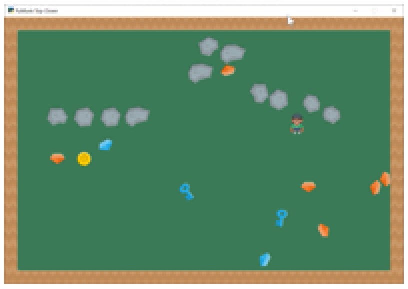


Fig. 94: pymunk_demo_top_down

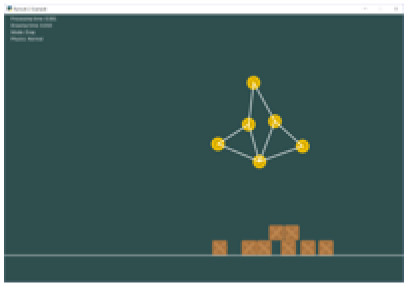


Fig. 95: pymunk_joint_builder

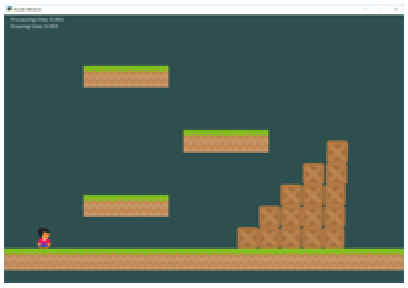
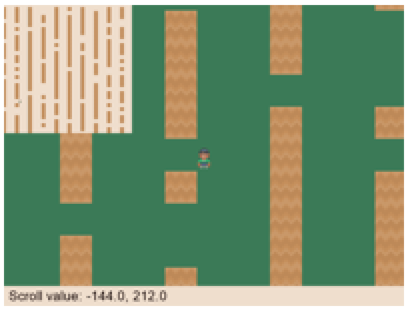
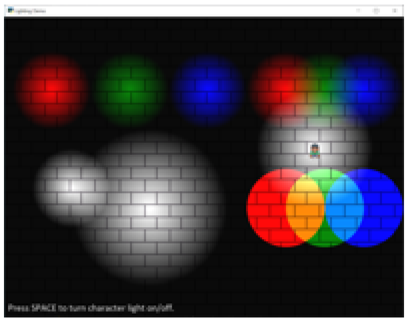


Fig. 96: *Pymunk Platformer*



Scroll value: -144.0, 212.0

Fig. 97: minimap



Press SPACE to turn character light on/off.

Fig. 98: light_demo

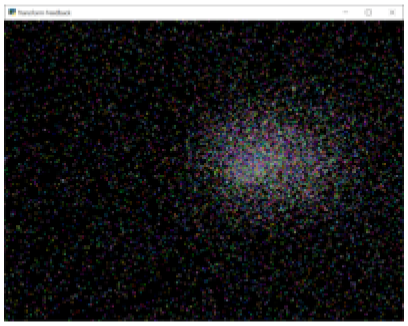


Fig. 99: transform_feedback

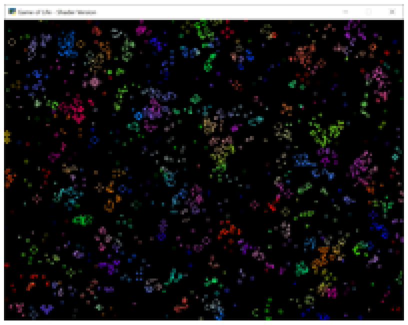


Fig. 100: game_of_life_fbo

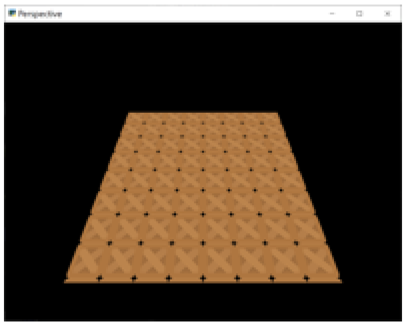


Fig. 101: perspective

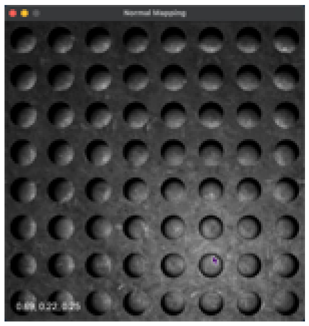


Fig. 102: normal_mapping

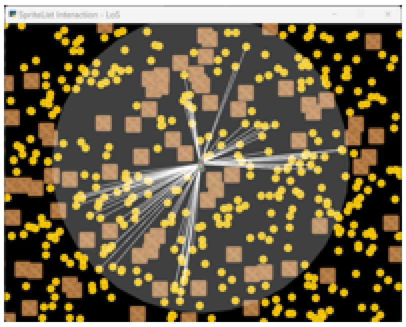


Fig. 103: spritelist_interaction_visualize_dist_los

5.10 Advanced

5.10.1 Using PyMunk for Physics

5.10.2 Frame Buffers

5.10.3 OpenGL

5.11 Concept Games

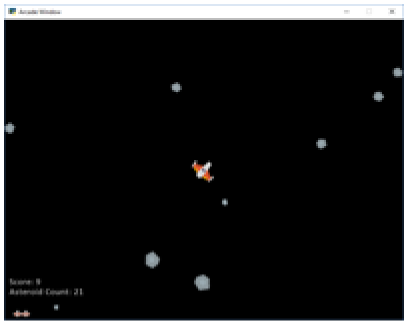


Fig. 104: asteroid_smasher

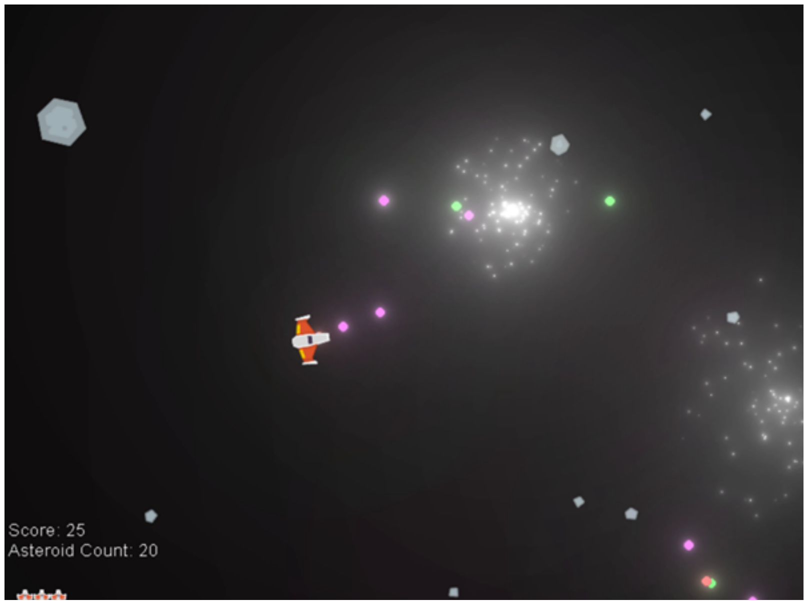


Fig. 105: Asteroids with Shaders

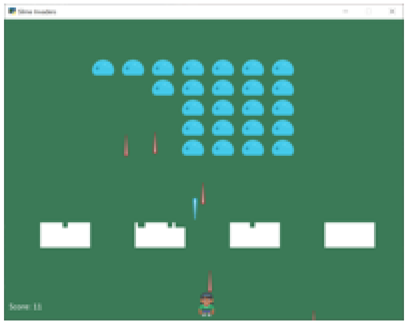


Fig. 106: slime_invaders

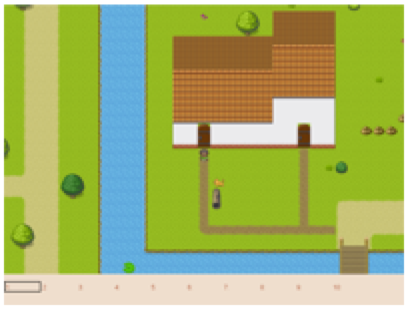


Fig. 107: Community RPG

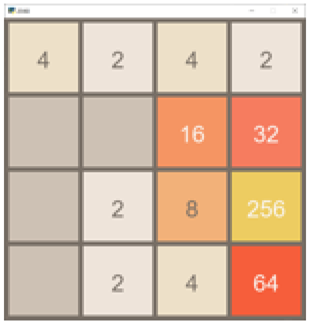


Fig. 108: 2048

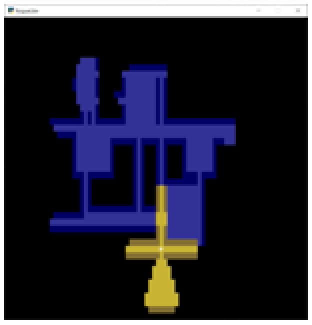


Fig. 109: Rogue-Like

5.12 Odds and Ends

Fig. 110: timer

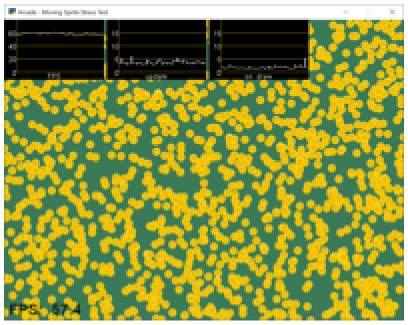


Fig. 111: performance_statistics_example

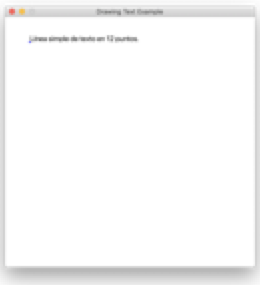


Fig. 112: text_loc_example

5.12.1 Particle System

5.13 Tutorials

5.14 Stress Tests

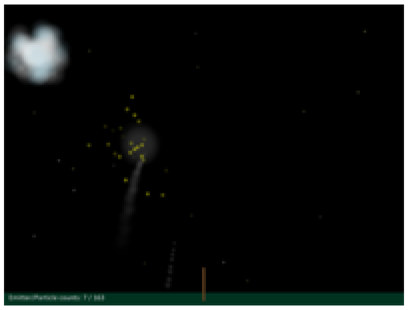


Fig. 113: particle_fireworks

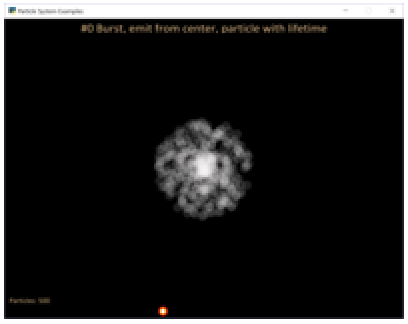


Fig. 114: particle_systems

Fig. 115: *Simple Platformer*

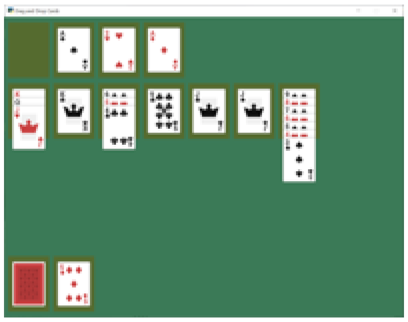


Fig. 116: *Solitaire*

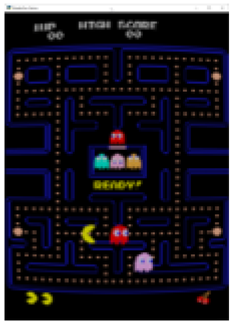


Fig. 117: *CRT Filter*

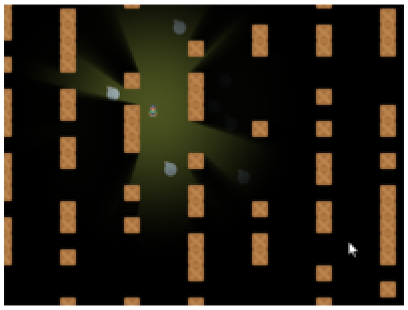


Fig. 118: *Ray-casting Shadows*

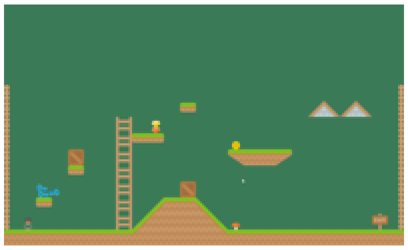


Fig. 119: *Pymunk Platformer*

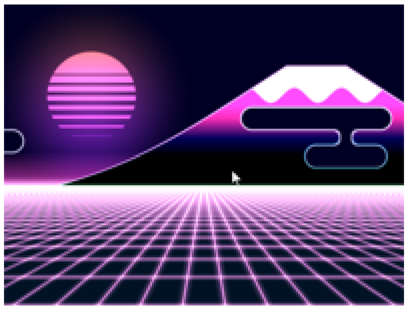


Fig. 120: *Shader Toy - Glow*

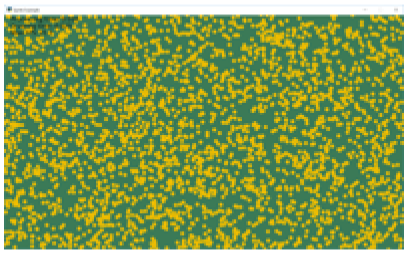


Fig. 121: stress_test_draw_moving

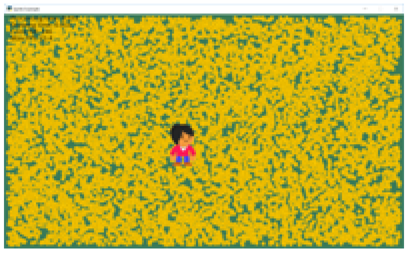


Fig. 122: stress_test_collision

CHAPTER

SIX

PYTHON DISCORD GAMEJAM 2020

The Python Discord 2020 Game Jam finished on April 26, 2020. Participants completed a game in one week. Twenty-three teams completed games, all of which are on the Game Jam 2020 GitHub.

We played the top 10 games on the Game Jam live-stream, which is available for replay.

Here are the games that made it to the top 10:

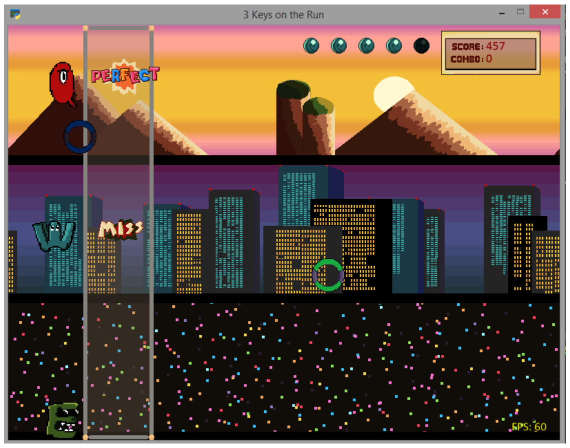


Fig. 1: 1st Place: 3 Keys on the Run

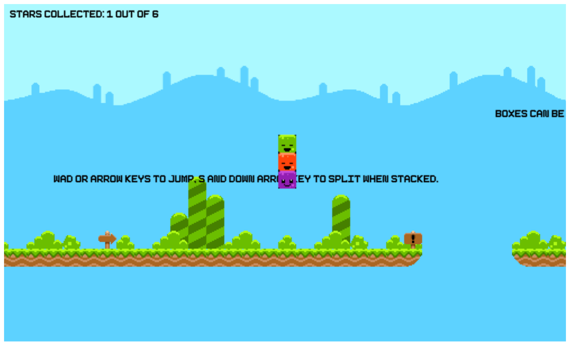

STARS COLLECTED: 1 OUT OF 6
BOXES CAN BE
WAD OR ARROW KEYS TO JUMP. S AND DOWN ARROW KEY TO SPLIT WHEN STACKED.

Fig. 2: 2nd Place: Triple Blocks

Fig. 3: 3nd Place: Triple Vision

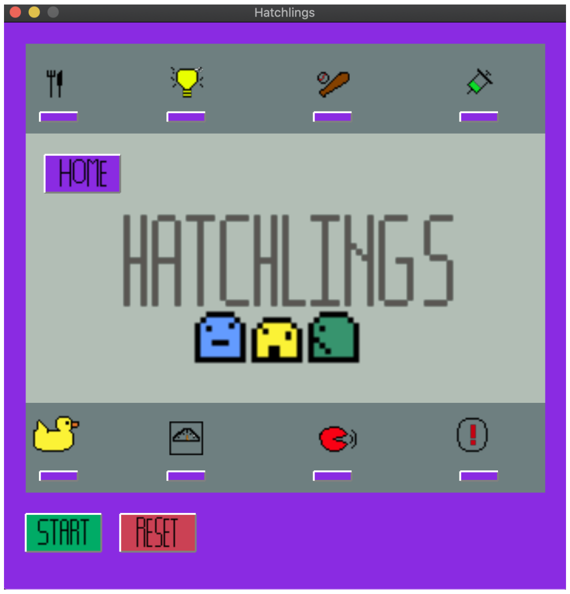


Fig. 4: Honourable Mention: Hatchlings

Fig. 5: Honourable Mention: Gem Matcher

Fig. 6: Tri-Chess

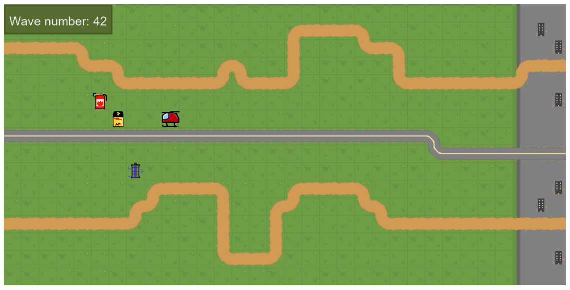


Fig. 7: Insane Irradiated Insectz

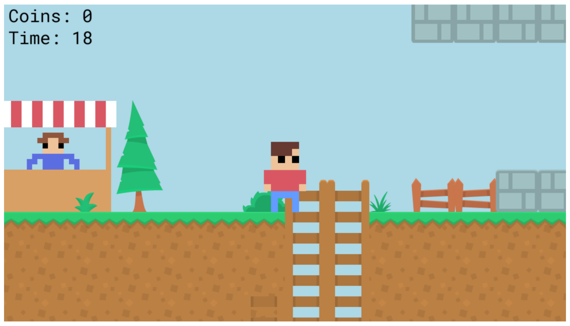


Fig. 8: Flimsy Billy's Coin Dash 3: Super Tag 3 Electric Tree

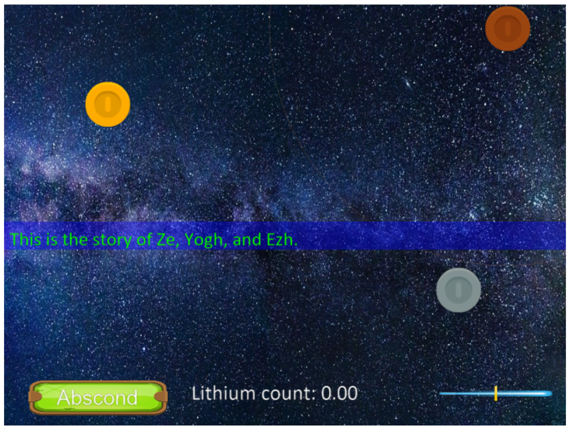


Fig. 9: ZeYoughEzh

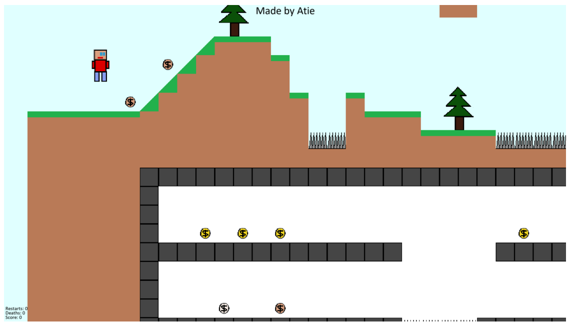


Fig. 10: Coin Collector

CHAPTER SEVEN

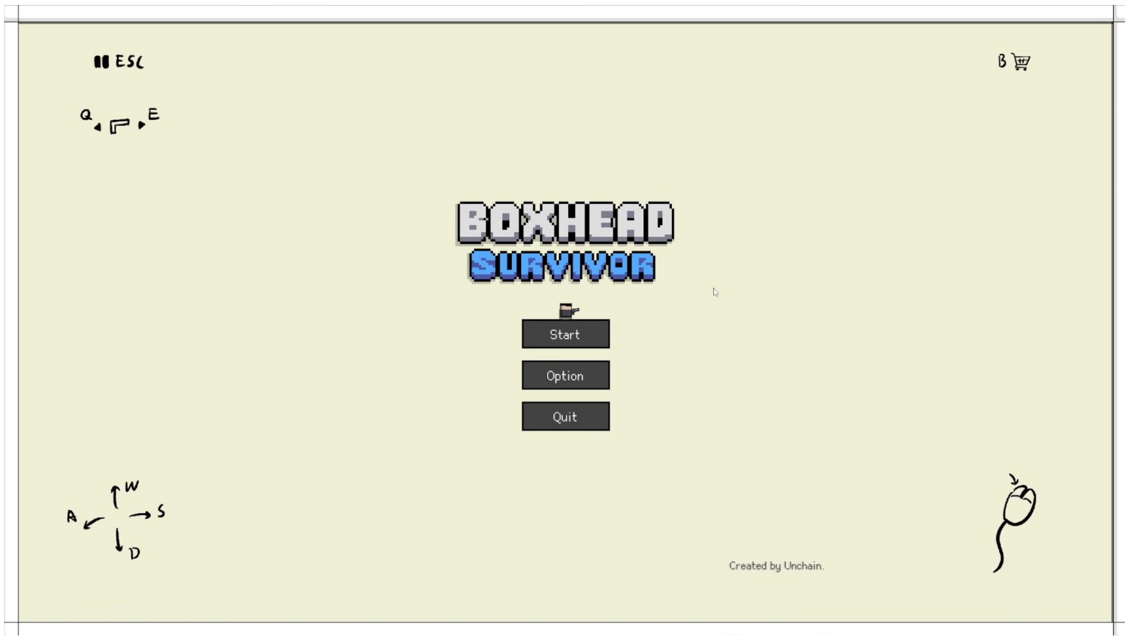
GAMES MADE WITH ARCADE

Here are some sample games made with Arcade. Have a game you'd like to share here? E-mail paul@cravenfamily.com.

You also might want to check out sample Arcade games from:

- *Python Discord GameJam 2020*
- *Concept Games*
- *Simple Platformer*

7.1 BoxHead



- GitHub: Unchained112/BoxHead2D: A top-down shooter game dev with Python Arcade (github.com)
- itch io: BoxHead Survivor by Unchain (itch.io)

7.2 Temporum

Temporum, by DragonMoffon

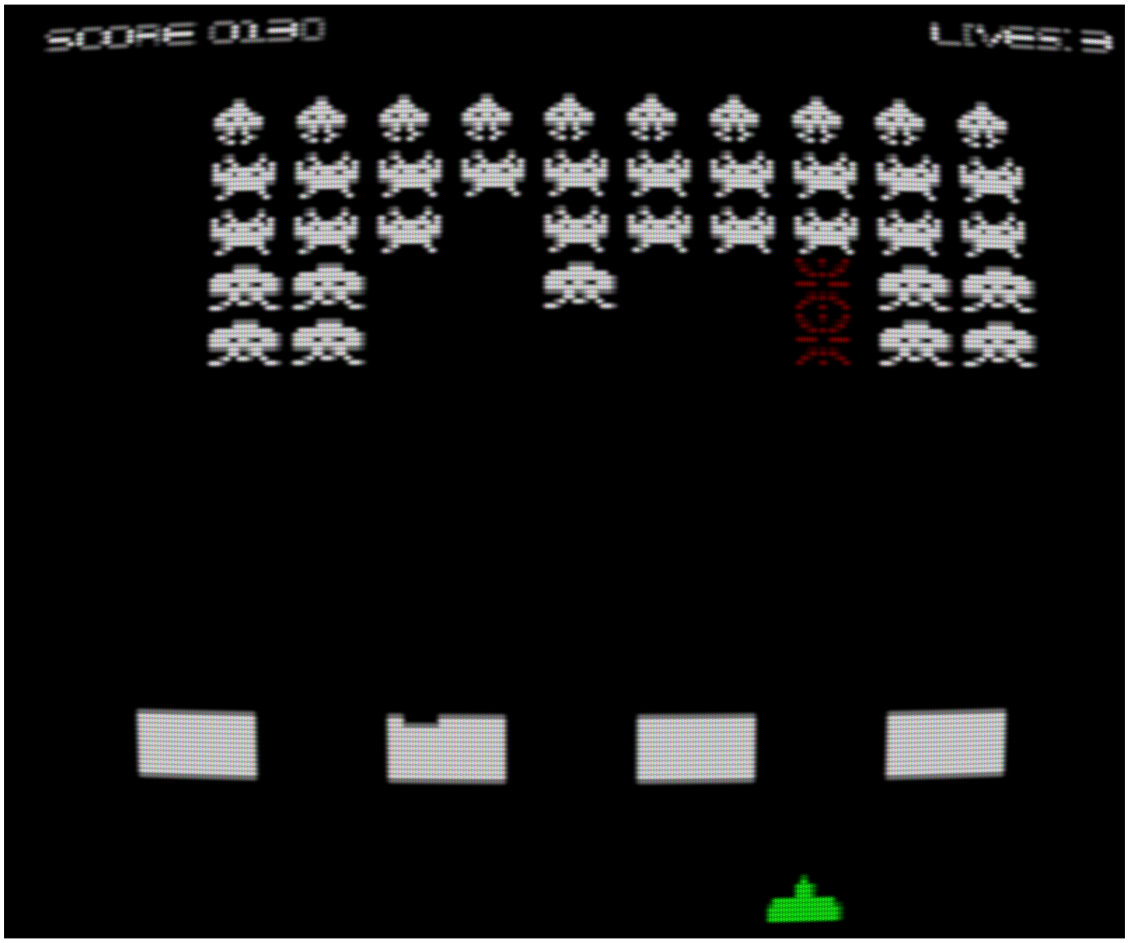
7.3 SOL Defender

SOL Defender, by DragonMoffon

7.4 Binary Defense

Binary Defense by KommentatorForAll

7.5 Space Invaders



Space Invaders

7.6 Ready or Not?

Ready or Not? a local multiplayer action RPG by Akash S Panickar.

7.7 Age of Divisiveness

Age of Divisiveness by Patryk Majewski, Krzysztof Szymaniak, Gabriel Wechta, Błażej Wróbel

Multiplayer LAN game with strong Civilization I and old Settlers vibe! Very extensive.

7.8 Fishy-Game

Fishy Game by LiorAvrahami

7.9 Adventure

Adventure GitHub

7.10 Transcience Animation

Transcience Animation

7.11 Stellar Arena Demo

Stellar Arena Demo

7.12 Battle Bros

Battle Bros Mortal Kombat style game.

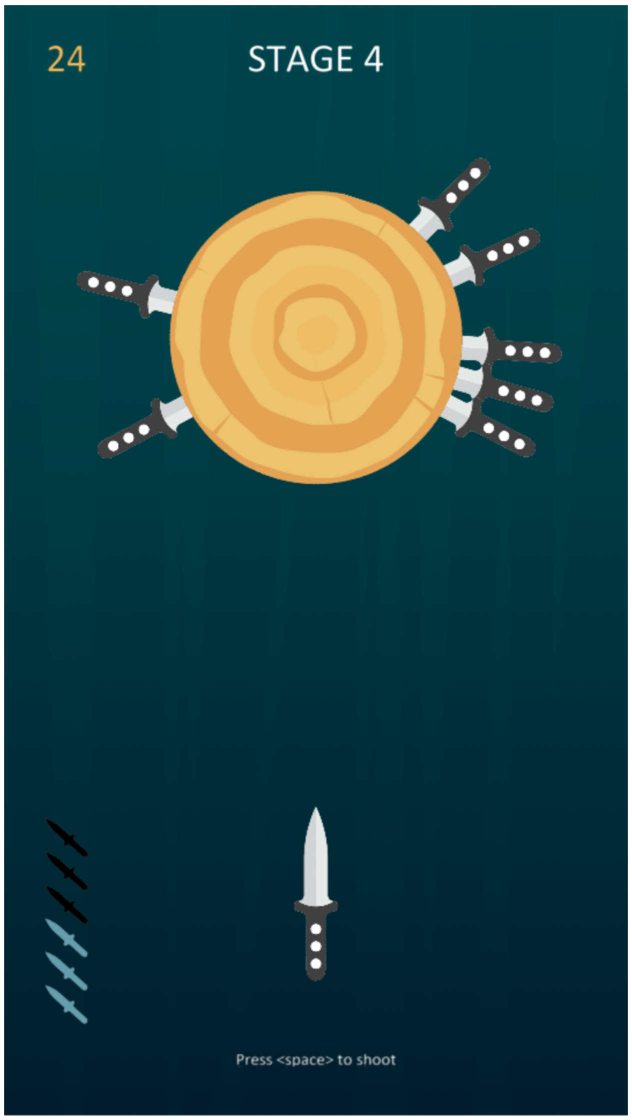
7.13 Rabbit Herder

Rabbit Herder, use carrots and potions to herd a rabbit through a maze.

7.14 The Great Skeleton War

The Great Skeleton War, an intense tower defense game, where there's always something new to discover.

7.15 Python Knife Hit



https://github.com/akmalhakimi1991/python-knife-hit

7.16 Kayzee

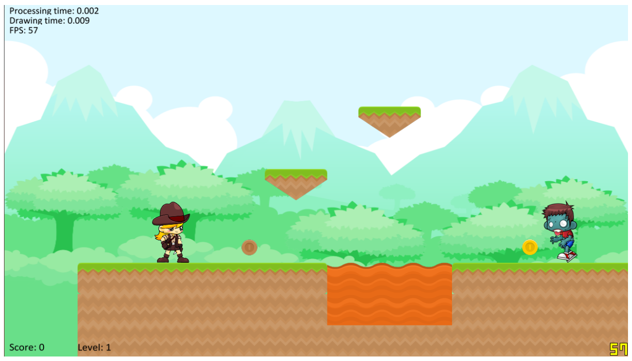


Fig. 1: Kayzee Game

7.17 lixingqiu Games

Fig. 2: An Eight planet simulation

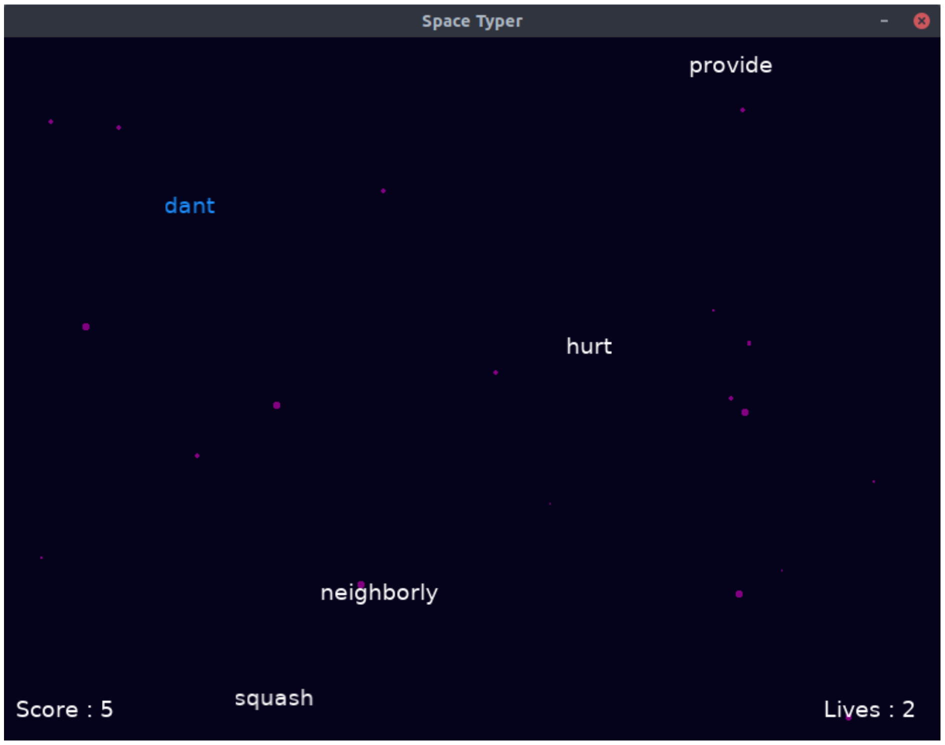
A Simple Version of Midway Island Naval Warfare with Pthon for Easy Learning(Python中途岛海战街机模拟)
1PLAYER
1576940
HI-SCORE
1576940
RR
2PLAYER
0
1943

Fig. 3: Midway Island War

Fig. 4: Angry Bird

Fig. 5: Octopus

7.18 Space Typer

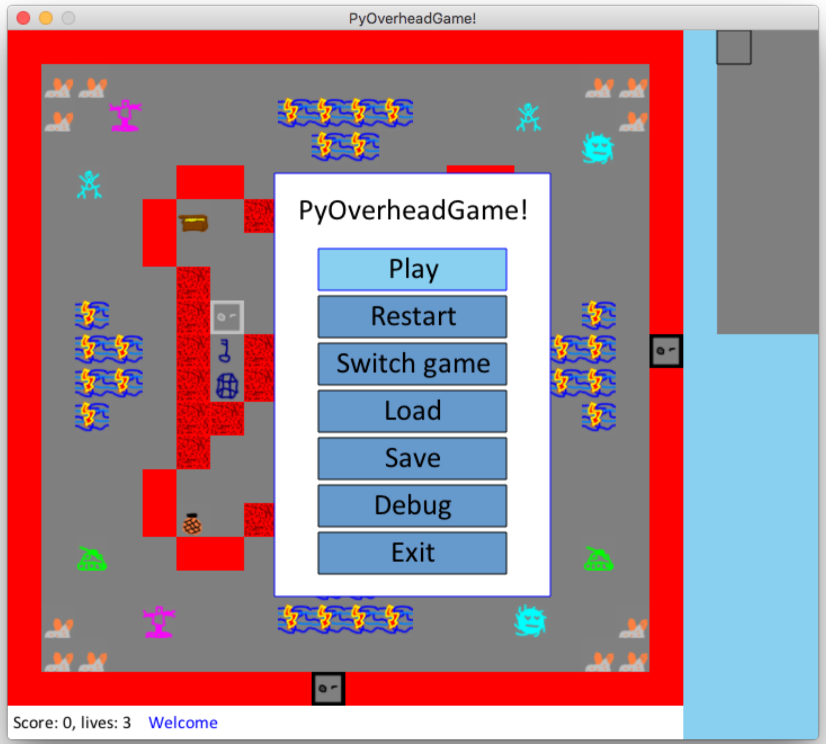


Space Typer - A typing game

7.19 FlapPy Bird

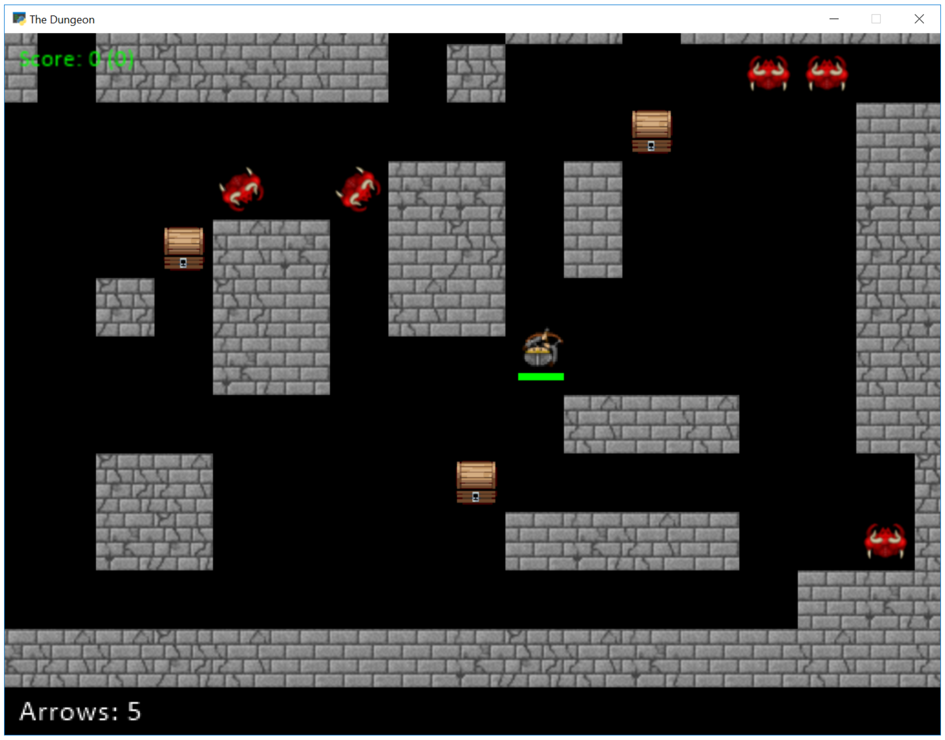
FlapPy-Bird - A bird-game clone.

7.20 PyOverheadGame



PyOverheadGame, a 2D overhead game where you go through several rooms and pick up keys and other objects.

7.21 Dungeon



Dungeon, explore a maze picking up arrows and coins.

7.22 Two Worlds

Two Worlds, a castle adventure with a dungeon and caverns underneath it.

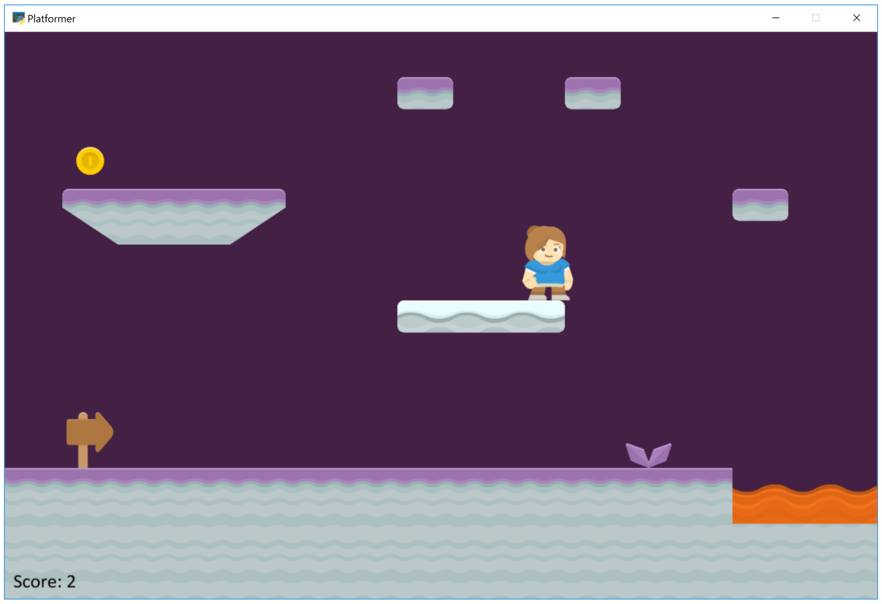
7.22.1 Simpson College Spring 2017 CMSC 150 Course

These games were created by first-semester programming students.

CHAPTER

EIGHT

SIMPLE PLATFORMER



This tutorial shows how to use Python and the Arcade library to create a 2D platformer game. You'll learn to work with Sprites and the Tiled Map Editor to create your own games. You can add coins, ramps, moving platforms, enemies, and more.

At the end of each chapter of this tutorial you will find the full source code that chapter, as well as a command to run that chapter directly. As long as you have Arcade installed the commands will run the exact code for that chapter, so you can compare your game against the tutorial.

8.1 Step 1 - Install and Open a Window

Our first step is to make sure everything is installed, and that we can at least get a window open.

8.1.1 Installation

- Make sure Python is installed. Download Python here if you don't already have it.
- Make sure the Arcade library is installed.
  - You should first setup a virtual environment (venv) and activate it.
  - Install Arcade with `pip install arcade`.
  - Here are the longer, official *Installation*.

8.1.2 Open a Window

The example below opens up a blank window. Set up a project and get the code below working.

Note: This is a fixed-size window. It is possible to have a resizable_window or a full_screen_example, but there are more interesting things we can do first. Therefore we'll stick with a fixed-size window for this tutorial.

Listing 1: 01_open_window.py - Open a Window

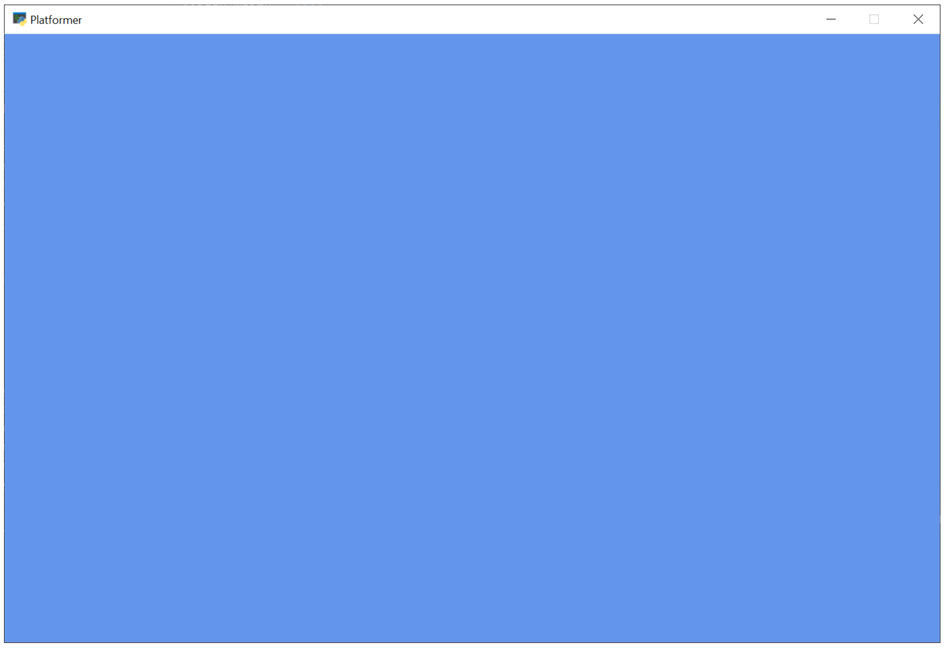
```python
"""
Platformer Game

python -m arcade.examples.platform_tutorial.01_open_window
"""
import arcade

# Constants
SCREEN_WIDTH = 800
SCREEN_HEIGHT = 600
SCREEN_TITLE = "Platformer"


class MyGame(arcade.Window):
    """
    Main application class.
    """

    def __init__(self):

        # Call the parent class to set up the window
        super().__init__(SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_TITLE)

        self.background_color = arcade.csscolor.CORNFLOWER_BLUE

    def setup(self):
        """Set up the game here. Call this function to restart the game."""
```

(continues on next page)

(continued from previous page)

```
            pass

    def on_draw(self):
        """Render the screen."""

        # The clear method should always be called at the start of on_draw.
        # It clears the whole screen to whatever the background color is
        # set to. This ensures that you have a clean slate for drawing each
        # frame of the game.
        self.clear()

        # Code to draw other things will go here


def main():
    """Main function"""
    window = MyGame()
    window.setup()
    arcade.run()


if __name__ == "__main__":
    main()
```

You should end up with a window like this:



Once you get the code working, try figuring out how to adjust the code so you can:

- Change the screen size(or even make the Window resizable or fullscreen)
- Change the title
- Change the background color

  - See the documentation for *arcade.color package*
  - See the documentation for *arcade.csscolor package*
- Look through the documentation for the `arcade.Window` class to get an idea of everything it can do.

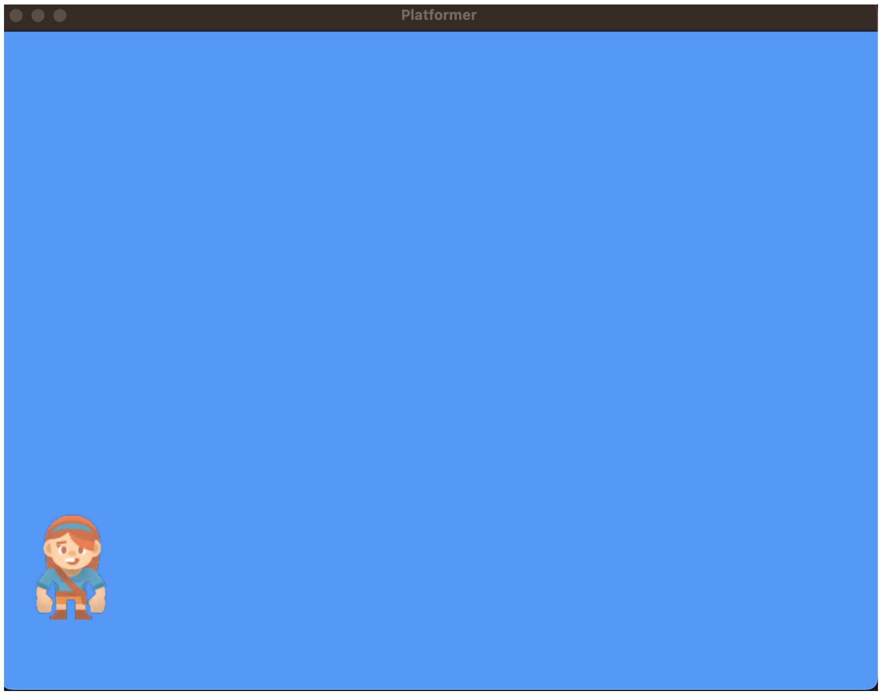
8.1.3 Run This Chapter

```
python -m arcade.examples.platform_tutorial.01_open_window
```

8.2 Step 2 - Textures and Sprites

Our next step in this tutorial is to draw something on the Screen. In order to do that we need to cover two topics, Textures and Sprites.

At the end of this chapter, we'll have something that looks like this. It's largely the same as last chapter, but now we are drawing a character onto the screen:



8.2.1 Textures

Textures are largely just an object to contain image data. Whenever you load an image file in Arcade, for example a `.png` or `.jpeg` file. It becomes a Texture.

To do this, internally Arcade uses Pyglet to load the image data, and the texture is responsible for keeping track of this image data.

We can create a texture with a simple command, this can be done inside of our `__init__` function. Go ahead and create a texture that we will use to draw a player.

```
self.player_texture = arcade.load_texture(":resources:images/animated_characters/female_
↪adventurer/femaleAdventurer_idle.png")
```

Note: You might be wondering where this image file is coming from? And what is `:resources:` about?

The `:resources:` section of the string above is what Arcade calls a resource handle. You can register your own resource handles to different asset directories. For example you might want to have a `:characters:` and a `:objects:` handle.

However, you don't have to use a resource handle here, anywhere that you can load files in Arcade will accept resource handles, or just strings to filepaths, or `Path` objects from `pathlib`

Arcade includes the `:resources:` handle with a bunch of built-in assets from kenney.

For more information checkout *Built-In Resources*

8.2.2 Sprites

If Textures are an instance of a particular image, then *`arcade.Sprite`* is an instance of that image on the screen. Say we have a ground or wall texture. We only have one instance of the texture, but we can create multiple instances of Sprite, because we want to have many walls. These will use the same texture, but draw it in different positions, or even with different scaling, rotation, or colors/post-processing effects.

Creating a Sprite is simple, we can make one for our player in our `__init__` function, and then set it's position.

```
self.player_sprite = arcade.Sprite(self.player_texture)
self.player_sprite.center_x = 64
self.player_sprite.center_y = 128
```

Note: You can also skip `arcade.load_texture` from the previous step and pass the image file to `arcade.Sprite` in place of the Texture object. A Texture will automatically be created for you. However, it may desirable in larger projects to manage your textures directly.

Now we can draw the sprite by adding this to our `on_draw` function:

```
self.player_sprite.draw()
```

We're now drawing a Sprite to the screen! In the next chapter, we will introduce techniques to draw many(even hundreds of thousands) sprites at once.

8.2.3 Source Code

Listing 2: 02_draw_sprites - Draw and Position Sprites

```
"""
Platformer Game

python -m arcade.examples.platform_tutorial.02_draw_sprites
"""
import arcade
```

(continues on next page)

(continued from previous page)

```python

# Constants
SCREEN_WIDTH = 800
SCREEN_HEIGHT = 600
SCREEN_TITLE = "Platformer"


class MyGame(arcade.Window):
    """
    Main application class.
    """

    def __init__(self):

        # Call the parent class and set up the window
        super().__init__(SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_TITLE)

        # Variable to hold our texture for our player
        self.player_texture = arcade.load_texture(":resources:images/animated_characters/
↪female_adventurer/femaleAdventurer_idle.png")

        # Separate variable that holds the player sprite
        self.player_sprite = arcade.Sprite(self.player_texture)
        self.player_sprite.center_x = 64
        self.player_sprite.center_y = 128

        self.background_color = arcade.csscolor.CORNFLOWER_BLUE

    def setup(self):
        """Set up the game here. Call this function to restart the game."""
        pass

    def on_draw(self):
        """Render the screen."""

        # Clear the screen to the background color
        self.clear()

        # Draw our sprites
        self.player_sprite.draw()


def main():
    """Main function"""
    window = MyGame()
    window.setup()
    arcade.run()


if __name__ == "__main__":
    main()
```

Running this code should result in a character being drawn on the screen, like in the image at the start of the chapter.

- Documentation for the *arcade.Texture* class
- Documentation for the *arcade.Sprite* class

Note: Once you have the code up and working, try adjusting the code for the following:

- Adjust the code and try putting the sprite in new positions(Try setting the positions using other attributes of Sprite)
- Use different images for the texture (see *Built-In Resources* for the built-in images, or try using your own images.)
- Practice placing more sprites in different ways(like placing many with a loop)

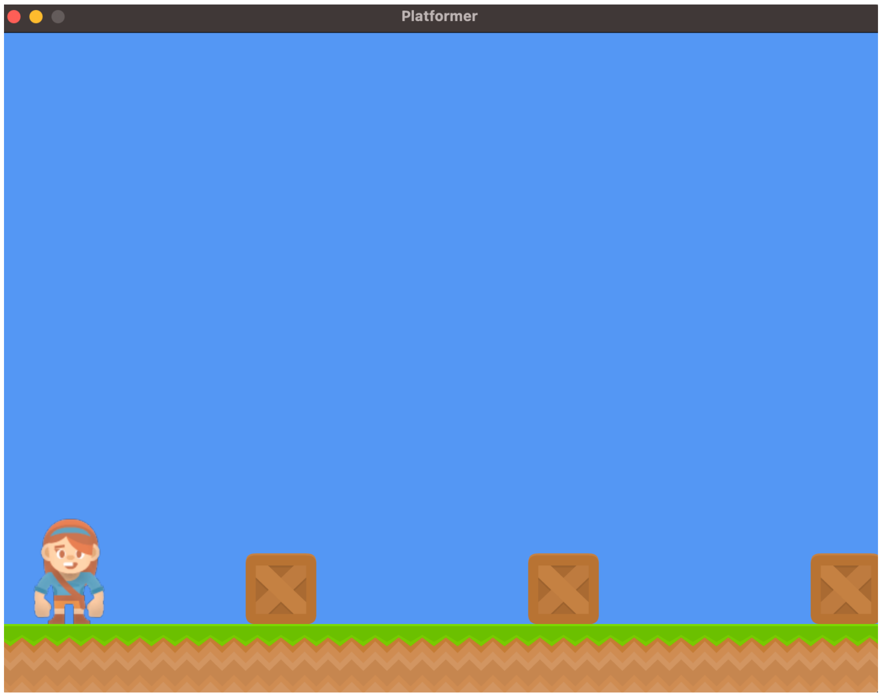
8.2.4 Run This Chapter

```
python -m arcade.examples.platform_tutorial.02_draw_sprites
```

8.3 Step 3 - Many Sprites with SpriteList

So far our game is coming along nicely, we have a character on the screen! Wouldn't it be nice if our character had a world to live in? In order to do that we'll need to draw a lot more sprites. In this chapter we will explore SpriteList, a class Arcade provides to draw tons of Sprites at once.

At the end, we'll have something like this:



8.3.1 SpriteList

arcade.SpriteList exists to draw a collection of Sprites all at once. Let's say for example that you have 100,000 box Sprites that you want to draw. Without SpriteList you would have to put all of your sprites into a list, and then run a for loop over that which calls `draw()` on every sprite.

This approach is extremely un-performant. Instead, you can add all of your boxes to a *arcade.SpriteList* and then draw the SpriteList. Doing this, you are able to draw all 100,000 sprites for approximately the exact same cost as drawing one sprite.

Note: This is due to Arcade being a heavily GPU based library. GPUs are really good at doing things in batches. This means we can send all the information about our sprites to the GPU, and then tell it to draw them all at once. However if we just draw one sprite at a time, then we have to go on a round trip from our CPU to our GPU every time.

Even if you are only drawing one Sprite, you should still create a SpriteList for it. Outside of small debugging it is never better to draw an individual Sprite than it is to add it to a SpriteList. In fact, calling `draw()` on a Sprite just creates a SpriteList internally to draw that Sprite with.

Let's go ahead and create one for our player inside our `__init__` function, and add the player to it.

```python
self.player_list = arcade.SpriteList()
self.player_list.append(self.player_sprite)
```

Then in our `on_draw` function, we can draw the SpriteList for the character instead of drawing the Sprite directly:

```python
self.player_list.draw()
```

Now let's try and build a world for our character. To do this, we'll create a new SpriteList for the objects we'll draw, we can do this in our `__init__` function.

```python
self.wall_list = arcade.SpriteList(use_spatial_hash=True)
```

There's a little bit to unpack in this snippet of code. Let's address each issue:

1. Why not just use the same SpriteList we used for our player, and why is it named walls?

   Eventually we will want to do collision detection between our character and these objects. In addition to drawing, SpriteLists also serve as a utility for collision detection. You can for example check for collisions between two SpriteLists, or pass SpriteLists into several physics engines. We will explore these topics in later chapters.

2. What is `use_spatial_hash`?

   This is also for collision detection. Spatial Hashing is a special algorithm which will make it much more performant, at the cost of being more expensive to move sprites. You will often see this option enabled on SpriteLists which are not expected to move much, such as walls or a floor.

With our newly created SpriteList, let's go ahead and add some objects to it. We can add these lines to our `__init__` function.

```python
for x in range(0, 1250, 64):
    wall = arcade.Sprite(":resources:images/tiles/grassMid.png", TILE_SCALING)
    wall.center_x = x
    wall.center_y = 32
    self.wall_list.append(wall)
```

(continues on next page)

(continued from previous page)

```
coordinate_list = [[512, 96], [256, 96], [768, 96]]
for coordinate in coordinate_list:
    wall = arcade.Sprite(
        ":resources:images/tiles/boxCrate_double.png", scale=0.5
    )
    wall.position = coordinate
    self.wall_list.append(wall)
```

In these lines, we're adding some grass and some crates to our SpriteList.

For the ground we're using Python's `range` function to iterate on a list of X positions, which will give us a horizontal line of Sprites. For the boxes, we're inserting them at specified coordinates from a list.

We're also doing a few new things in the *`arcade.Sprite`* creation. First off we are passing the image file directly instead of creating a texture first. This is ultimately doing the same thing, we're just not managing the texture ourselves, and letting Arcade handle it. We are also adding a scale to these sprites. For fun you can remove the scale, and see how the images will be much larger.

Finally all we need to do in order to draw our new world, is draw the SpriteList for walls in `on_draw`:

```
self.wall_list.draw()
```

8.3.2 Source Code

Listing 3: 03_more_sprites - Many Sprites with a SpriteList

```
"""
Platformer Game

python -m arcade.examples.platform_tutorial.03_more_sprites
"""
import arcade

# Constants
SCREEN_WIDTH = 800
SCREEN_HEIGHT = 600
SCREEN_TITLE = "Platformer"

# Constants used to scale our sprites from their original size
TILE_SCALING = 0.5


class MyGame(arcade.Window):
    """
    Main application class.
    """

    def __init__(self):

        # Call the parent class and set up the window
        super().__init__(SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_TITLE)

```

(continues on next page)

(continued from previous page)

```python
        # Variable to hold our texture for our player
        self.player_texture = arcade.load_texture(":resources:images/animated_characters/
↪female_adventurer/femaleAdventurer_idle.png")

        # Separate variable that holds the player sprite
        self.player_sprite = arcade.Sprite(self.player_texture)
        self.player_sprite.center_x = 64
        self.player_sprite.center_y = 128

        # SpriteList for our player
        self.player_list = arcade.SpriteList()
        self.player_list.append(self.player_sprite)

        # SpriteList for our boxes and ground
        # Putting our ground and box Sprites in the same SpriteList
        # will make it easier to perform collision detection against
        # them later on. Setting the spatial hash to True will make
        # collision detection much faster if the objects in this
        # SpriteList do not move.
        self.wall_list = arcade.SpriteList(use_spatial_hash=True)

        # Create the ground
        # This shows using a loop to place multiple sprites horizontally
        for x in range(0, 1250, 64):
            wall = arcade.Sprite(":resources:images/tiles/grassMid.png", scale=0.5)
            wall.center_x = x
            wall.center_y = 32
            self.wall_list.append(wall)

        # Put some crates on the ground
        # This shows using a coordinate list to place sprites
        coordinate_list = [[512, 96], [256, 96], [768, 96]]

        for coordinate in coordinate_list:
            # Add a crate on the ground
            wall = arcade.Sprite(
                ":resources:images/tiles/boxCrate_double.png", scale=0.5
            )
            wall.position = coordinate
            self.wall_list.append(wall)

        self.background_color = arcade.csscolor.CORNFLOWER_BLUE

    def setup(self):
        """Set up the game here. Call this function to restart the game."""
        pass

    def on_draw(self):
        """Render the screen."""

        # Clear the screen to the background color
        self.clear()
```

(continues on next page)

(continued from previous page)

```python

        # Draw our sprites
        self.player_list.draw()
        self.wall_list.draw()


def main():
    """Main function"""
    window = MyGame()
    window.setup()
    arcade.run()


if __name__ == "__main__":
    main()
```

- Documentation for the *arcade.SpriteList* class

Note: Once you have the code up and working, try-out the following:

- See if you can change the colors of all the boxes and ground using the SpriteList
- Try and make a SpriteList invisible

8.3.3 Run This Chapter

```
python -m arcade.examples.platform_tutorial.03_more_sprites
```

8.4 Step 4 - Add User Control

Now we've got a character and a world for them to exist in, but what fun is a game if you can't control the character and move around? In this Chapter we'll explore adding keyboard input in Arcade.

First, at the top of our program, we'll want to add a new constant that controls how many pixels per update our character travels:

```python
PLAYER_MOVEMENT_SPEED = 5
```

In order to handle the keyboard input, we need to add to add two new functions to our Window class, `on_key_press` and `on_key_release`. These functions will automatically be called by Arcade whenever a key on the keyboard is pressed or released. Inside these functions, based on the key that was pressed or released, we will move our character.

```python
def on_key_press(self, key, modifiers):
    """Called whenever a key is pressed."""

    if key == arcade.key.UP or key == arcade.key.W:
        self.player_sprite.change_y = PLAYER_MOVEMENT_SPEED
    elif key == arcade.key.DOWN or key == arcade.key.S:
        self.player_sprite.change_y = -PLAYER_MOVEMENT_SPEED
```

(continues on next page)

(continued from previous page)

```
    elif key == arcade.key.LEFT or key == arcade.key.A:
        self.player_sprite.change_x = -PLAYER_MOVEMENT_SPEED
    elif key == arcade.key.RIGHT or key == arcade.key.D:
        self.player_sprite.change_x = PLAYER_MOVEMENT_SPEED

def on_key_release(self, key, modifiers):
    """Called whenever a key is released."""

    if key == arcade.key.UP or key == arcade.key.W:
        self.player_sprite.change_y = 0
    elif key == arcade.key.DOWN or key == arcade.key.S:
        self.player_sprite.change_y = 0
    elif key == arcade.key.LEFT or key == arcade.key.A:
        self.player_sprite.change_x = 0
    elif key == arcade.key.RIGHT or key == arcade.key.D:
        self.player_sprite.change_x = 0
```

In these boxes, we are modifying the `change_x` and `change_y` attributes on our player Sprite. Changing these values will not actually perform the move on the Sprite. In order to apply this change, we need to create a physics engine with our Sprite, and update the physics engine every frame. The physics engine will then be responsible for actually moving the sprite.

The reason we give the physics engine this responsibility instead of doing it ourselves, is so that we can let the physics engine do collision detections, and allow/disallow a movement based on the result. In later chapters, we'll use more advanced physics engines which can do things like allow jumping with gravity, or climbing on ladders for example.

Note: This method of tracking the speed to the key the player presses is simple, but isn't perfect. If the player hits both left and right keys at the same time, then lets off the left one, we expect the player to move right. This method won't support that. If you want a slightly more complex method that does, see sprite_move_keyboard_better.

Let's create a simple physics engine in our `__init__` function. We will do this by passing it our player sprite, and the SpriteList containing our walls.

```
self.physics_engine = arcade.PhysicsEngineSimple(
    self.player_sprite, self.wall_list
)
```

Now we have a physics engine, but we still need to update it every frame. In order to do this we will add a new function to our Window class, called `on_update`. This function is similar to `on_draw`, it will be called by Arcade at a default of 60 times per second. It will also give us a `delta_time` parameter that tells the amount of time between the last call and the current one. This value will be used in some calculations in future chapters. Within this function, we will update our physics engine. Which will process collision detections and move our player based on it's `change_x` and `change_y` values.

```
def on_update(self, delta_time):
    """Movement and Game Logic"""

    self.physics_engine.update()
```

At this point you should be able to run the game, and move the character around with the keyboard. If the physics engine is working properly, the character should not be able to move through the ground or the boxes.

For more information about the physics engine we are using in this tutorial, see *arcade.PhysicsEngineSimple*.

Note: It is possible to have multiple physics engines, one per moving sprite. These are very simple, but easy physics engines. See *Pymunk Platformer* for a more advanced physics engine.

Note: If you want to see how the collisions are checked, try using the `draw_hit_boxes()` function on the player and wall SpriteLists inside the `on_draw` function. This will show you what the hitboxes that the physics engine uses look like.

8.4.1 Source Code

Listing 4: 04_user_control.py - User Control

```python
"""
Platformer Game

python -m arcade.examples.platform_tutorial.04_user_control
"""
import arcade

# Constants
SCREEN_WIDTH = 800
SCREEN_HEIGHT = 600
SCREEN_TITLE = "Platformer"

# Constants used to scale our sprites from their original size
TILE_SCALING = 0.5

# Movement speed of player, in pixels per frame
PLAYER_MOVEMENT_SPEED = 5


class MyGame(arcade.Window):
    """
    Main application class.
    """

    def __init__(self):

        # Call the parent class and set up the window
        super().__init__(SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_TITLE)

        # Variable to hold our texture for our player
        self.player_texture = arcade.load_texture(":resources:images/animated_characters/
↪female_adventurer/femaleAdventurer_idle.png")

        # Separate variable that holds the player sprite
        self.player_sprite = arcade.Sprite(self.player_texture)
        self.player_sprite.center_x = 64
        self.player_sprite.center_y = 128

```

(continues on next page)

(continued from previous page)

```
        # SpriteList for our player
        self.player_list = arcade.SpriteList()
        self.player_list.append(self.player_sprite)

        # SpriteList for our boxes and ground
        # Putting our ground and box Sprites in the same SpriteList
        # will make it easier to perform collision detection against
        # them later on. Setting the spatial hash to True will make
        # collision detection much faster if the objects in this
        # SpriteList do not move.
        self.wall_list = arcade.SpriteList(use_spatial_hash=True)

        # Create the ground
        # This shows using a loop to place multiple sprites horizontally
        for x in range(0, 1250, 64):
            wall = arcade.Sprite(":resources:images/tiles/grassMid.png", scale=TILE_
↪SCALING)
            wall.center_x = x
            wall.center_y = 32
            self.wall_list.append(wall)

        # Put some crates on the ground
        # This shows using a coordinate list to place sprites
        coordinate_list = [[512, 96], [256, 96], [768, 96]]

        for coordinate in coordinate_list:
            # Add a crate on the ground
            wall = arcade.Sprite(
                ":resources:images/tiles/boxCrate_double.png", scale=TILE_SCALING
            )
            wall.position = coordinate
            self.wall_list.append(wall)

        # Create a Simple Physics Engine, this will handle moving our
        # player as well as collisions between the player sprite and
        # whatever SpriteList we specify for the walls.
        self.physics_engine = arcade.PhysicsEngineSimple(
            self.player_sprite, self.wall_list
        )

        self.background_color = arcade.csscolor.CORNFLOWER_BLUE

    def setup(self):
        """Set up the game here. Call this function to restart the game."""
        pass

    def on_draw(self):
        """Render the screen."""

        # Clear the screen to the background color
        self.clear()

```

(continues on next page)

(continued from previous page)

```python
        # Draw our sprites
        self.player_list.draw()
        self.wall_list.draw()

    def on_update(self, delta_time):
        """Movement and Game Logic"""

        # Move the player using our physics engine
        self.physics_engine.update()

    def on_key_press(self, key, modifiers):
        """Called whenever a key is pressed."""

        if key == arcade.key.UP or key == arcade.key.W:
            self.player_sprite.change_y = PLAYER_MOVEMENT_SPEED
        elif key == arcade.key.DOWN or key == arcade.key.S:
            self.player_sprite.change_y = -PLAYER_MOVEMENT_SPEED
        elif key == arcade.key.LEFT or key == arcade.key.A:
            self.player_sprite.change_x = -PLAYER_MOVEMENT_SPEED
        elif key == arcade.key.RIGHT or key == arcade.key.D:
            self.player_sprite.change_x = PLAYER_MOVEMENT_SPEED

    def on_key_release(self, key, modifiers):
        """Called whenever a key is released."""

        if key == arcade.key.UP or key == arcade.key.W:
            self.player_sprite.change_y = 0
        elif key == arcade.key.DOWN or key == arcade.key.S:
            self.player_sprite.change_y = 0
        elif key == arcade.key.LEFT or key == arcade.key.A:
            self.player_sprite.change_x = 0
        elif key == arcade.key.RIGHT or key == arcade.key.D:
            self.player_sprite.change_x = 0


def main():
    """Main function"""
    window = MyGame()
    window.setup()
    arcade.run()


if __name__ == "__main__":
    main()
```

8.4.2 Run This Chapter

```
python -m arcade.examples.platform_tutorial.04_user_control
```

8.5 Step 5 - Add Gravity

The previous example is great for top-down games, but what if it is a side view with jumping like our platformer? We need to add gravity. First, let's define a constant to represent the acceleration for gravity, and one for a jump speed.

```
GRAVITY = 1
PLAYER_JUMP_SPEED = 20
```

Now, let's change the Physics Engine we created in the `__init__` function to a *arcade.PhysicsEnginePlatformer* instead of a *arcade.PhysicsEngineSimple*. This new physics engine will handle jumping and gravity for us, and will do even more in later chapters.

```
self.physics_engine = arcade.PhysicsEnginePlatformer(
    self.player_sprite, walls=self.wall_list, gravity_constant=GRAVITY
)
```

This is very similar to how we created the original simple physics engine, with two exceptions. The first being that we have sent it our gravity constant. The second being that we have explicitly sent our wall SpriteList to the `walls` parameter. This is a very important step. The platformer physics engine has two parameters for collidable objects, one named `platforms` and one named `walls`.

The difference is that objects sent to `platforms` are intended to be moved. They are moved in the same way the player is, by modifying their `change_x` and `change_y` values. Objects sent to the `walls` parameter will not be moved. The reason this is so important is that non-moving walls have much faster performance than movable platforms.

Adding static sprites via the `platforms` parameter is roughly an O(n) operation, meaning performance will linearly get worse as you add more sprites. If you add your static sprites via the `walls` parameter, then it is nearly O(1) and there is essentially no difference between for example 100 and 50,000 non-moving sprites.

Lastly we will give our player the ability to jump. Modify the `on_key_press` and `on_key_release` functions. We'll remove the up/down statements we had before, and make `UP` jump when pressed.

```
if key == arcade.key.UP or key == arcade.key.W:
    if self.physics_engine.can_jump():
        self.player_sprite.change_y = PLAYER_JUMP_SPEED
```

The `can_jump()` check from our physics engine will make it so that we can only jump if we are touching the ground. You can remove this function to allow jumping in mid-air for some interesting results. Think about how you might implement a double-jump system using this.

Note: You can change how the user jumps by changing the gravity and jump constants. Lower values for both will make for a more "floaty" character. Higher values make for a faster-paced game.

8.5.1 Source Code

Listing 5: 05_add_gravity.py - Add Gravity

```python
"""
Platformer Game

python -m arcade.examples.platform_tutorial.05_add_gravity
"""
import arcade

# Constants
SCREEN_WIDTH = 800
SCREEN_HEIGHT = 600
SCREEN_TITLE = "Platformer"

# Constants used to scale our sprites from their original size
TILE_SCALING = 0.5

# Movement speed of player, in pixels per frame
PLAYER_MOVEMENT_SPEED = 5
GRAVITY = 1
PLAYER_JUMP_SPEED = 20


class MyGame(arcade.Window):
    """
    Main application class.
    """

    def __init__(self):

        # Call the parent class and set up the window
        super().__init__(SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_TITLE)

        # Variable to hold our texture for our player
        self.player_texture = arcade.load_texture(":resources:images/animated_characters/
↪female_adventurer/femaleAdventurer_idle.png")

        # Separate variable that holds the player sprite
        self.player_sprite = arcade.Sprite(self.player_texture)
        self.player_sprite.center_x = 64
        self.player_sprite.center_y = 128

        # SpriteList for our player
        self.player_list = arcade.SpriteList()
        self.player_list.append(self.player_sprite)

        # SpriteList for our boxes and ground
        # Putting our ground and box Sprites in the same SpriteList
        # will make it easier to perform collision detection against
        # them later on. Setting the spatial hash to True will make
        # collision detection much faster if the objects in this
```

(continues on next page)

(continued from previous page)

```
        # SpriteList do not move.
        self.wall_list = arcade.SpriteList(use_spatial_hash=True)

        # Create the ground
        # This shows using a loop to place multiple sprites horizontally
        for x in range(0, 1250, 64):
            wall = arcade.Sprite(":resources:images/tiles/grassMid.png", scale=TILE_
↪SCALING)
            wall.center_x = x
            wall.center_y = 32
            self.wall_list.append(wall)

        # Put some crates on the ground
        # This shows using a coordinate list to place sprites
        coordinate_list = [[512, 96], [256, 96], [768, 96]]

        for coordinate in coordinate_list:
            # Add a crate on the ground
            wall = arcade.Sprite(
                ":resources:images/tiles/boxCrate_double.png", scale=TILE_SCALING
            )
            wall.position = coordinate
            self.wall_list.append(wall)

        # Create a Platformer Physics Engine.
        # This will handle moving our player as well as collisions between
        # the player sprite and whatever SpriteList we specify for the walls.
        # It is important to supply static platforms to the walls parameter. There is a
        # platforms parameter that is intended for moving platforms.
        # If a platform is supposed to move, and is added to the walls list,
        # it will not be moved.
        self.physics_engine = arcade.PhysicsEnginePlatformer(
            self.player_sprite, walls=self.wall_list, gravity_constant=GRAVITY
        )

        self.background_color = arcade.csscolor.CORNFLOWER_BLUE

    def setup(self):
        """Set up the game here. Call this function to restart the game."""
        pass

    def on_draw(self):
        """Render the screen."""

        # Clear the screen to the background color
        self.clear()

        # Draw our sprites
        self.player_list.draw()
        self.wall_list.draw()

    def on_update(self, delta_time):
```

(continues on next page)

(continued from previous page)

```python
        """Movement and Game Logic"""

        # Move the player using our physics engine
        self.physics_engine.update()

    def on_key_press(self, key, modifiers):
        """Called whenever a key is pressed."""

        if key == arcade.key.UP or key == arcade.key.W:
            if self.physics_engine.can_jump():
                self.player_sprite.change_y = PLAYER_JUMP_SPEED

        if key == arcade.key.LEFT or key == arcade.key.A:
            self.player_sprite.change_x = -PLAYER_MOVEMENT_SPEED
        elif key == arcade.key.RIGHT or key == arcade.key.D:
            self.player_sprite.change_x = PLAYER_MOVEMENT_SPEED

    def on_key_release(self, key, modifiers):
        """Called whenever a key is released."""

        if key == arcade.key.LEFT or key == arcade.key.A:
            self.player_sprite.change_x = 0
        elif key == arcade.key.RIGHT or key == arcade.key.D:
            self.player_sprite.change_x = 0


def main():
    """Main function"""
    window = MyGame()
    window.setup()
    arcade.run()


if __name__ == "__main__":
    main()
```

8.5.2 Run This Chapter

```
python -m arcade.examples.platform_tutorial.05_add_gravity
```

8.6 Step 6 - Resetting

You might have noticed that throughout this tutorial, there has been a `setup` function in our Window class. So far, we haven't used this function at all, so what is it for?

Let's imagine that we want a way to "reset" our game to it's initial state. This could be because the player lost, and we want to restart the game, or perhaps we just want to give the player the option to restart.

With our current architecture of creating everything in our `__init__` function, we would have to duplicate all of that logic in another function in order to make that happen, or completely re-create our Window, which will be an unpleasent experience for a player.

In this chapter, we will do a small amount of re-organizing our existing code to make use of this setup function in a way that allows to simply call the `setup` function whenever we want our game to return to it's original state.

First off, we will change our `__init__` function to look like below. We are setting values to something like `None`, 0, or similar. The purpose of this step is to ensure that the attributes are created on the class. In Python, we cannot add new attributes to a class outside of the `__init__` function.

```
def __init__(self):

    super().__init__(SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_TITLE)

    self.player_texture = None
    self.player_sprite = None
    self.player_list = None

    self.wall_list = None
```

Next we will move the actual creation of these objects into our setup function. This looks almost identical to our original `__init__` function. Try and move these sections of code on your own, if you get stuck you can see the `setup` function in the full source code listing below.

The last thing we need to do is create a way to reset the game. For now we'll add a simple key press to do it. Add the following in your `on_key_press` function to reset the game when the Escape key is pressed.

```
if key == arcade.key.ESCAPE:
    self.setup()
```

8.6.1 Source Code

Listing 6: Resetting

```
"""
Platformer Game

python -m arcade.examples.platform_tutorial.06_reset
"""
import arcade

# Constants
SCREEN_WIDTH = 800
SCREEN_HEIGHT = 600
SCREEN_TITLE = "Platformer"

# Constants used to scale our sprites from their original size
TILE_SCALING = 0.5

# Movement speed of player, in pixels per frame
PLAYER_MOVEMENT_SPEED = 5
GRAVITY = 1
PLAYER_JUMP_SPEED = 20


```

(continues on next page)

(continued from previous page)

```python
class MyGame(arcade.Window):
    """
    Main application class.
    """

    def __init__(self):

        # Call the parent class and set up the window
        super().__init__(SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_TITLE)

        # Variable to hold our texture for our player
        self.player_texture = None

        # Separate variable that holds the player sprite
        self.player_sprite = None

        # SpriteList for our player
        self.player_list = None

        # SpriteList for our boxes and ground
        # Putting our ground and box Sprites in the same SpriteList
        # will make it easier to perform collision detection against
        # them later on. Setting the spatial hash to True will make
        # collision detection much faster if the objects in this
        # SpriteList do not move.
        self.wall_list = None

    def setup(self):
        """Set up the game here. Call this function to restart the game."""
        self.player_texture = arcade.load_texture(":resources:images/animated_characters/
↪female_adventurer/femaleAdventurer_idle.png")

        self.player_sprite = arcade.Sprite(self.player_texture)
        self.player_sprite.center_x = 64
        self.player_sprite.center_y = 128

        self.player_list = arcade.SpriteList()
        self.player_list.append(self.player_sprite)

        self.wall_list = arcade.SpriteList(use_spatial_hash=True)

        # Create the ground
        # This shows using a loop to place multiple sprites horizontally
        for x in range(0, 1250, 64):
            wall = arcade.Sprite(":resources:images/tiles/grassMid.png", scale=TILE_
↪SCALING)
            wall.center_x = x
            wall.center_y = 32
            self.wall_list.append(wall)

        # Put some crates on the ground
        # This shows using a coordinate list to place sprites
```

(continues on next page)

(continued from previous page)

```
        coordinate_list = [[512, 96], [256, 96], [768, 96]]

        for coordinate in coordinate_list:
            # Add a crate on the ground
            wall = arcade.Sprite(
                ":resources:images/tiles/boxCrate_double.png", scale=TILE_SCALING
            )
            wall.position = coordinate
            self.wall_list.append(wall)

        # Create a Platformer Physics Engine, this will handle moving our
        # player as well as collisions between the player sprite and
        # whatever SpriteList we specify for the walls.
        # It is important to supply static to the walls parameter. There is a
        # platforms parameter that is intended for moving platforms.
        # If a platform is supposed to move, and is added to the walls list,
        # it will not be moved.
        self.physics_engine = arcade.PhysicsEnginePlatformer(
            self.player_sprite, walls=self.wall_list, gravity_constant=GRAVITY
        )

        self.background_color = arcade.csscolor.CORNFLOWER_BLUE

    def on_draw(self):
        """Render the screen."""

        # Clear the screen to the background color
        self.clear()

        # Draw our sprites
        self.player_list.draw()
        self.wall_list.draw()

    def on_update(self, delta_time):
        """Movement and Game Logic"""

        # Move the player using our physics engine
        self.physics_engine.update()

    def on_key_press(self, key, modifiers):
        """Called whenever a key is pressed."""

        if key == arcade.key.ESCAPE:
            self.setup()

        if key == arcade.key.UP or key == arcade.key.W:
            if self.physics_engine.can_jump():
                self.player_sprite.change_y = PLAYER_JUMP_SPEED

        if key == arcade.key.LEFT or key == arcade.key.A:
            self.player_sprite.change_x = -PLAYER_MOVEMENT_SPEED
        elif key == arcade.key.RIGHT or key == arcade.key.D:
```

(continues on next page)

(continued from previous page)

```python
                self.player_sprite.change_x = PLAYER_MOVEMENT_SPEED

    def on_key_release(self, key, modifiers):
        """Called whenever a key is released."""

        if key == arcade.key.LEFT or key == arcade.key.A:
            self.player_sprite.change_x = 0
        elif key == arcade.key.RIGHT or key == arcade.key.D:
            self.player_sprite.change_x = 0


def main():
    """Main function"""
    window = MyGame()
    window.setup()
    arcade.run()


if __name__ == "__main__":
    main()
```

8.6.2 Run This Chapter

```
python -m arcade.examples.platform_tutorial.06_reset
```

8.7 Step 7 - Adding a Camera

Now that our player can move and jump around, we need to give them a way to explore the world beyond the original window. If you've ever played a platformer game, you might be familiar with the concept of the screen scrolling to reveal more of the map as the player moves.

To achieve this, we can use a Camera, Arcade provides *arcade.SimpleCamera* and *arcade.Camera*. They both do the same base thing, but Camera has a bit of extra functionality that SimpleCamera doesn't. For now, we will just use the SimpleCamera.

To start with, let's go ahead and add a variable in our `__init__` function to hold it:

```python
self.camera = None
```

Next we can go to our setup function, and initialize it like so:

```python
self.camera = arcade.SimpleCamera(viewport=(0, 0, self.width, self.height))
```

The `viewport` parameter here defines the size of the camera. In most circumstances, you will want this to be the size of your window. So we specify the bottom and left coordinates of our camera viewport as (0, 0), and provide it the width and height of our window.

In order to use our camera when drawing things to the screen, we only need to add one line to our `on_draw` function. This line should typically come before anything you want to draw with the camera. In later chapters, we'll explore using multiple cameras to draw things in different positions. Go ahead and add this line before drawing our SpriteLists

```
self.camera.use()
```

If you run the game at this point, you might notice that nothing has changed, our game is still one static un-moving screen. This is because we are never updating the camera's position. In our platformer game, we want the camera to follow the player, and keep them in the center of the screen. Arcade provides a helpful function to do this with one line of code. In other types of games or more advanced usage you may want to set the cameras position directly in order to create interesting effects, but for now all we need is the `center()` function of our camera.

If we add the following line to our `on_update()` function and run the game, you should now see the player stay at the center of the screen, while being able to scroll the screen around to the rest of our map. For fun, see what happens if you fall off of the map! Later on, we'll revisit a more advanced camera setup that will take the bounds of our world into consideration.

```
self.camera.center(self.player_sprite.position)
```

8.7.1 Source Code

Listing 7: Adding a Camera

```
"""
Platformer Game

python -m arcade.examples.platform_tutorial.07_camera
"""
import arcade

# Constants
SCREEN_WIDTH = 800
SCREEN_HEIGHT = 600
SCREEN_TITLE = "Platformer"

# Constants used to scale our sprites from their original size
TILE_SCALING = 0.5

# Movement speed of player, in pixels per frame
PLAYER_MOVEMENT_SPEED = 5
GRAVITY = 1
PLAYER_JUMP_SPEED = 20


class MyGame(arcade.Window):
    """
    Main application class.
    """

    def __init__(self):

        # Call the parent class and set up the window
        super().__init__(SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_TITLE)

        # Variable to hold our texture for our player
        self.player_texture = None
```

(continues on next page)

(continued from previous page)

```python

        # Separate variable that holds the player sprite
        self.player_sprite = None

        # SpriteList for our player
        self.player_list = None

        # SpriteList for our boxes and ground
        # Putting our ground and box Sprites in the same SpriteList
        # will make it easier to perform collision detection against
        # them later on. Setting the spatial hash to True will make
        # collision detection much faster if the objects in this
        # SpriteList do not move.
        self.wall_list = None

        # A variable to store our camera object
        self.camera = None

    def setup(self):
        """Set up the game here. Call this function to restart the game."""
        self.player_texture = arcade.load_texture(":resources:images/animated_characters/
↪female_adventurer/femaleAdventurer_idle.png")

        self.player_sprite = arcade.Sprite(self.player_texture)
        self.player_sprite.center_x = 64
        self.player_sprite.center_y = 128

        self.player_list = arcade.SpriteList()
        self.player_list.append(self.player_sprite)

        self.wall_list = arcade.SpriteList(use_spatial_hash=True)

        # Create the ground
        # This shows using a loop to place multiple sprites horizontally
        for x in range(0, 1250, 64):
            wall = arcade.Sprite(":resources:images/tiles/grassMid.png", scale=TILE_
↪SCALING)
            wall.center_x = x
            wall.center_y = 32
            self.wall_list.append(wall)

        # Put some crates on the ground
        # This shows using a coordinate list to place sprites
        coordinate_list = [[512, 96], [256, 96], [768, 96]]

        for coordinate in coordinate_list:
            # Add a crate on the ground
            wall = arcade.Sprite(
                ":resources:images/tiles/boxCrate_double.png", scale=TILE_SCALING
            )
            wall.position = coordinate
            self.wall_list.append(wall)
```

(continues on next page)

(continued from previous page)

```python

        # Create a Platformer Physics Engine, this will handle moving our
        # player as well as collisions between the player sprite and
        # whatever SpriteList we specify for the walls.
        # It is important to supply static to the walls parameter. There is a
        # platforms parameter that is intended for moving platforms.
        # If a platform is supposed to move, and is added to the walls list,
        # it will not be moved.
        self.physics_engine = arcade.PhysicsEnginePlatformer(
            self.player_sprite, walls=self.wall_list, gravity_constant=GRAVITY
        )

        # Initialize our camera, setting a viewport the size of our window.
        self.camera = arcade.SimpleCamera(viewport=(0, 0, self.width, self.height))

        self.background_color = arcade.csscolor.CORNFLOWER_BLUE

    def on_draw(self):
        """Render the screen."""

        # Clear the screen to the background color
        self.clear()

        # Activate our camera before drawing
        self.camera.use()

        # Draw our sprites
        self.player_list.draw()
        self.wall_list.draw()

    def on_update(self, delta_time):
        """Movement and Game Logic"""

        # Move the player using our physics engine
        self.physics_engine.update()

        # Center our camera on the player
        self.camera.center(self.player_sprite.position)

    def on_key_press(self, key, modifiers):
        """Called whenever a key is pressed."""

        if key == arcade.key.ESCAPE:
            self.setup()

        if key == arcade.key.UP or key == arcade.key.W:
            if self.physics_engine.can_jump():
                self.player_sprite.change_y = PLAYER_JUMP_SPEED

        if key == arcade.key.LEFT or key == arcade.key.A:
            self.player_sprite.change_x = -PLAYER_MOVEMENT_SPEED
        elif key == arcade.key.RIGHT or key == arcade.key.D:
```

(continues on next page)

(continued from previous page)

```
                self.player_sprite.change_x = PLAYER_MOVEMENT_SPEED

    def on_key_release(self, key, modifiers):
        """Called whenever a key is released."""

        if key == arcade.key.LEFT or key == arcade.key.A:
            self.player_sprite.change_x = 0
        elif key == arcade.key.RIGHT or key == arcade.key.D:
            self.player_sprite.change_x = 0


def main():
    """Main function"""
    window = MyGame()
    window.setup()
    arcade.run()


if __name__ == "__main__":
    main()
```

8.7.2 Run This Chapter

```
python -m arcade.examples.platform_tutorial.07_camera
```

8.8 Step 8 - Collecting Coins

Now that we can fully move around our game, we need to give the player an objective. A classic goal in video games is collecting coins, so let's go ahead and add that.

In this chapter you will learn how to check for collisions with our player, and find out exactly what they collided with and do something with it. For now we will just remove the coin from the screen when they collect it, but in later chapters we will give the character a score, and add to it when they collect a coin. We will also start playing sounds later.

First off we will create a new SpriteList to hold our coins. Exactly like our other spritelist for walls, go ahead and add a variable to the `__init__` function to store it, and then initialize it inside the `setup` function. We will want to turn on spatial hashing for this list for now. If you decided to have moving coins, you would want to turn that off.

```
# Inside __init__
self.coin_list = None

# Inside setup
self.coin_list = arcade.SpriteList(use_spatial_hash=True)
```

See if you can experiment with a way to add the coins to the SpriteList using what we've already learned. The built-in resource for them is `:resources:images/items/coinGold.png`. HINT: You'll want to scale these just like we did with our boxes and ground. If you get stuck, you can check the full source code below to see how we've placed them following the same pattern we used for the ground.

Once you have placed the coins and added them to the `coin_list`, don't forget to add them to `on_draw`.

```
self.coin_list.draw()
```

Now that we're drawing our coins to the screen, how do we make them interact with the player? When the player hits one, we want to remove it from the screen. To do this we will use *arcade.check_for_collision_with_list()* function. This function takes a single Sprite, in this instance our player, and a SpriteList, for us, the coins. It will return a list containing all of the Sprites from the given SpriteList that the Sprite collided with.

We can iterate over that list with a for loop to do something with each sprite that had a collision. This means we can detect the user hitting multiple coins at once if we had them placed close together.

In order to do this, and remove the coin sprites when the player hits them, we will add this to the `on_update` function.

```
coin_hit_list = arcade.check_for_collision_with_list(
    self.player_sprite, self.coin_list
)

for coin in coin_hit_list:
    coin.remove_from_sprite_lists()
```

We use this *arcade.BasicSprite.remove_from_sprite_lists()* function in order to ensure our Sprite is completely removed from all SpriteLists it was a part of.

8.8.1 Source Code

Listing 8: Collecting Coins

```
"""
Platformer Game

python -m arcade.examples.platform_tutorial.08_coins
"""
import arcade

# Constants
SCREEN_WIDTH = 800
SCREEN_HEIGHT = 600
SCREEN_TITLE = "Platformer"

# Constants used to scale our sprites from their original size
TILE_SCALING = 0.5
COIN_SCALING = 0.5

# Movement speed of player, in pixels per frame
PLAYER_MOVEMENT_SPEED = 5
GRAVITY = 1
PLAYER_JUMP_SPEED = 20


class MyGame(arcade.Window):
    """
    Main application class.
    """

```

(continues on next page)

(continued from previous page)

```
    def __init__(self):

        # Call the parent class and set up the window
        super().__init__(SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_TITLE)

        # Variable to hold our texture for our player
        self.player_texture = None

        # Separate variable that holds the player sprite
        self.player_sprite = None

        # SpriteList for our player
        self.player_list = None

        # SpriteList for our boxes and ground
        # Putting our ground and box Sprites in the same SpriteList
        # will make it easier to perform collision detection against
        # them later on. Setting the spatial hash to True will make
        # collision detection much faster if the objects in this
        # SpriteList do not move.
        self.wall_list = None

        # SpriteList for coins the player can collect
        self.coin_list = None

        # A variable to store our camera object
        self.camera = None

    def setup(self):
        """Set up the game here. Call this function to restart the game."""
        self.player_texture = arcade.load_texture(":resources:images/animated_characters/
↪female_adventurer/femaleAdventurer_idle.png")

        self.player_sprite = arcade.Sprite(self.player_texture)
        self.player_sprite.center_x = 64
        self.player_sprite.center_y = 128

        self.player_list = arcade.SpriteList()
        self.player_list.append(self.player_sprite)

        self.wall_list = arcade.SpriteList(use_spatial_hash=True)
        self.coin_list = arcade.SpriteList(use_spatial_hash=True)

        # Create the ground
        # This shows using a loop to place multiple sprites horizontally
        for x in range(0, 1250, 64):
            wall = arcade.Sprite(":resources:images/tiles/grassMid.png", scale=TILE_
↪SCALING)
            wall.center_x = x
            wall.center_y = 32
            self.wall_list.append(wall)

```

(continues on next page)

(continued from previous page)

```
        # Put some crates on the ground
        # This shows using a coordinate list to place sprites
        coordinate_list = [[512, 96], [256, 96], [768, 96]]

        for coordinate in coordinate_list:
            # Add a crate on the ground
            wall = arcade.Sprite(
                ":resources:images/tiles/boxCrate_double.png", scale=TILE_SCALING
            )
            wall.position = coordinate
            self.wall_list.append(wall)

        # Add coins to the world
        for x in range(128, 1250, 256):
            coin = arcade.Sprite(":resources:images/items/coinGold.png", scale=COIN_
↪SCALING)
            coin.center_x = x
            coin.center_y = 96
            self.coin_list.append(coin)

        # Create a Platformer Physics Engine, this will handle moving our
        # player as well as collisions between the player sprite and
        # whatever SpriteList we specify for the walls.
        # It is important to supply static to the walls parameter. There is a
        # platforms parameter that is intended for moving platforms.
        # If a platform is supposed to move, and is added to the walls list,
        # it will not be moved.
        self.physics_engine = arcade.PhysicsEnginePlatformer(
            self.player_sprite, walls=self.wall_list, gravity_constant=GRAVITY
        )

        # Initialize our camera, setting a viewport the size of our window.
        self.camera = arcade.SimpleCamera(viewport=(0, 0, self.width, self.height))

        self.background_color = arcade.csscolor.CORNFLOWER_BLUE

    def on_draw(self):
        """Render the screen."""

        # Clear the screen to the background color
        self.clear()

        # Activate our camera before drawing
        self.camera.use()

        # Draw our sprites
        self.player_list.draw()
        self.wall_list.draw()
        self.coin_list.draw()

    def on_update(self, delta_time):
        """Movement and Game Logic"""
```

(continues on next page)

(continued from previous page)

```python

        # Move the player using our physics engine
        self.physics_engine.update()

        # See if we hit any coins
        coin_hit_list = arcade.check_for_collision_with_list(
            self.player_sprite, self.coin_list
        )

        # Loop through each coin we hit (if any) and remove it
        for coin in coin_hit_list:
            # Remove the coin
            coin.remove_from_sprite_lists()

        # Center our camera on the player
        self.camera.center(self.player_sprite.position)

    def on_key_press(self, key, modifiers):
        """Called whenever a key is pressed."""

        if key == arcade.key.ESCAPE:
            self.setup()

        if key == arcade.key.UP or key == arcade.key.W:
            if self.physics_engine.can_jump():
                self.player_sprite.change_y = PLAYER_JUMP_SPEED

        if key == arcade.key.LEFT or key == arcade.key.A:
            self.player_sprite.change_x = -PLAYER_MOVEMENT_SPEED
        elif key == arcade.key.RIGHT or key == arcade.key.D:
            self.player_sprite.change_x = PLAYER_MOVEMENT_SPEED

    def on_key_release(self, key, modifiers):
        """Called whenever a key is released."""

        if key == arcade.key.LEFT or key == arcade.key.A:
            self.player_sprite.change_x = 0
        elif key == arcade.key.RIGHT or key == arcade.key.D:
            self.player_sprite.change_x = 0


def main():
    """Main function"""
    window = MyGame()
    window.setup()
    arcade.run()


if __name__ == "__main__":
    main()
```

8.8.2 Run This Chapter

```
python -m arcade.examples.platform_tutorial.08_coins
```

8.9 Step 9 - Adding Sound

Our game has a lot of graphics so far, but doesn't have any sound yet. Let's change that! In this chapter we will add a sound when the player collects the coins, as well as when they jump.

Loading and playing sounds in Arcade is very easy. We will only need two functions for this:

- *arcade.load_sound()*
- *arcade.play_sound()*

In our `__init__` function, we will add these two lines to load our coin collection and jump sounds.

```
self.collect_coin_sound = arcade.load_sound(":resources:sounds/coin1.wav")
self.jump_sound = arcade.load_sound(":resources:sounds/jump1.wav")
```

Note: Why are we not adding empty variables to `__init__` and initializing them in `setup` like our other objects?

This is because sounds are a static asset within our game. If we reset the game, the sounds don't change, so it's not worth re-loading them.

Now we can play these sounds by simple adding the `play_sound` function wherever we want them to occur. Let's add one alongside our removal of coins in the `on_update` function.

```
# Within on_update
for coin in coin_hit_list:
    coin.remove_from_sprite_lists()
    arcade.play_sound(self.collect_coin_sound)
```

This will play a sound whenever we collect a coin. We can add a jump sound by adding this to our UP block for jumping in the `on_key_press` function:

```
# Within on_key_press
if key == arcade.key.UP or key == arcade.key.W:
    if self.physics_engine.can_jump():
        self.player_sprite.change_y = PLAYER_JUMP_SPEED
        arcade.play_sound(self.jump_sound)
```

Now we will also have a sound whenever we jump.

Documentation for *arcade.Sound*

8.9.1 Source Code

Listing 9: Load the Map

```python
"""
Platformer Game

python -m arcade.examples.platform_tutorial.09_sound
"""
import arcade

# Constants
SCREEN_WIDTH = 800
SCREEN_HEIGHT = 600
SCREEN_TITLE = "Platformer"

# Constants used to scale our sprites from their original size
TILE_SCALING = 0.5
COIN_SCALING = 0.5

# Movement speed of player, in pixels per frame
PLAYER_MOVEMENT_SPEED = 5
GRAVITY = 1
PLAYER_JUMP_SPEED = 20


class MyGame(arcade.Window):
    """
    Main application class.
    """

    def __init__(self):

        # Call the parent class and set up the window
        super().__init__(SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_TITLE)

        # Variable to hold our texture for our player
        self.player_texture = None

        # Separate variable that holds the player sprite
        self.player_sprite = None

        # SpriteList for our player
        self.player_list = None

        # SpriteList for our boxes and ground
        # Putting our ground and box Sprites in the same SpriteList
        # will make it easier to perform collision detection against
        # them later on. Setting the spatial hash to True will make
        # collision detection much faster if the objects in this
        # SpriteList do not move.
        self.wall_list = None

```

(continues on next page)

(continued from previous page)

```
        # SpriteList for coins the player can collect
        self.coin_list = None

        # A variable to store our camera object
        self.camera = None

        # Load sounds
        self.collect_coin_sound = arcade.load_sound(":resources:sounds/coin1.wav")
        self.jump_sound = arcade.load_sound(":resources:sounds/jump1.wav")

    def setup(self):
        """Set up the game here. Call this function to restart the game."""
        self.player_texture = arcade.load_texture(":resources:images/animated_characters/
↪female_adventurer/femaleAdventurer_idle.png")

        self.player_sprite = arcade.Sprite(self.player_texture)
        self.player_sprite.center_x = 64
        self.player_sprite.center_y = 128

        self.player_list = arcade.SpriteList()
        self.player_list.append(self.player_sprite)

        self.wall_list = arcade.SpriteList(use_spatial_hash=True)
        self.coin_list = arcade.SpriteList(use_spatial_hash=True)

        # Create the ground
        # This shows using a loop to place multiple sprites horizontally
        for x in range(0, 1250, 64):
            wall = arcade.Sprite(":resources:images/tiles/grassMid.png", scale=TILE_
↪SCALING)
            wall.center_x = x
            wall.center_y = 32
            self.wall_list.append(wall)

        # Put some crates on the ground
        # This shows using a coordinate list to place sprites
        coordinate_list = [[512, 96], [256, 96], [768, 96]]

        for coordinate in coordinate_list:
            # Add a crate on the ground
            wall = arcade.Sprite(
                ":resources:images/tiles/boxCrate_double.png", scale=TILE_SCALING
            )
            wall.position = coordinate
            self.wall_list.append(wall)

        # Add coins to the world
        for x in range(128, 1250, 256):
            coin = arcade.Sprite(":resources:images/items/coinGold.png", scale=COIN_
↪SCALING)
            coin.center_x = x
            coin.center_y = 96
```

(continues on next page)

(continued from previous page)

```
            self.coin_list.append(coin)

        # Create a Platformer Physics Engine, this will handle moving our
        # player as well as collisions between the player sprite and
        # whatever SpriteList we specify for the walls.
        # It is important to supply static to the walls parameter. There is a
        # platforms parameter that is intended for moving platforms.
        # If a platform is supposed to move, and is added to the walls list,
        # it will not be moved.
        self.physics_engine = arcade.PhysicsEnginePlatformer(
            self.player_sprite, walls=self.wall_list, gravity_constant=GRAVITY
        )

        # Initialize our camera, setting a viewport the size of our window.
        self.camera = arcade.SimpleCamera(viewport=(0, 0, self.width, self.height))

        self.background_color = arcade.csscolor.CORNFLOWER_BLUE

    def on_draw(self):
        """Render the screen."""

        # Clear the screen to the background color
        self.clear()

        # Activate our camera before drawing
        self.camera.use()

        # Draw our sprites
        self.player_list.draw()
        self.wall_list.draw()
        self.coin_list.draw()

    def on_update(self, delta_time):
        """Movement and Game Logic"""

        # Move the player using our physics engine
        self.physics_engine.update()

        # See if we hit any coins
        coin_hit_list = arcade.check_for_collision_with_list(
            self.player_sprite, self.coin_list
        )

        # Loop through each coin we hit (if any) and remove it
        for coin in coin_hit_list:
            # Remove the coin
            coin.remove_from_sprite_lists()
            arcade.play_sound(self.collect_coin_sound)

        # Center our camera on the player
        self.camera.center(self.player_sprite.position)

```

(continues on next page)

(continued from previous page)

```python
    def on_key_press(self, key, modifiers):
        """Called whenever a key is pressed."""

        if key == arcade.key.ESCAPE:
            self.setup()

        if key == arcade.key.UP or key == arcade.key.W:
            if self.physics_engine.can_jump():
                self.player_sprite.change_y = PLAYER_JUMP_SPEED
                arcade.play_sound(self.jump_sound)

        if key == arcade.key.LEFT or key == arcade.key.A:
            self.player_sprite.change_x = -PLAYER_MOVEMENT_SPEED
        elif key == arcade.key.RIGHT or key == arcade.key.D:
            self.player_sprite.change_x = PLAYER_MOVEMENT_SPEED

    def on_key_release(self, key, modifiers):
        """Called whenever a key is released."""

        if key == arcade.key.LEFT or key == arcade.key.A:
            self.player_sprite.change_x = 0
        elif key == arcade.key.RIGHT or key == arcade.key.D:
            self.player_sprite.change_x = 0


def main():
    """Main function"""
    window = MyGame()
    window.setup()
    arcade.run()


if __name__ == "__main__":
    main()
```

8.9.2 Run This Chapter

```
python -m arcade.examples.platform_tutorial.09_sound
```

8.10 Step 10 - Adding a Score

Our game is starting to take shape, but we still need to give the player a reward for their hard work collecting coins. To do this we will add a score which will be increased everytime they collect a coin, and display that on the screen.

In this chapter we will cover using *arcade.Text* objects, as well as a technique for using two cameras to draw objects in “screen space” and objects in “world space”.

Note: What is screen space and world space? Think about other games you may have played, and let’s compare it to our game. A player moves around in the world, and we scroll a camera around based on that position. This is an example of "world space" coordinates. They can expand beyond our window and need to be positioned within the window accordingly.

An example of "screen space" coordinates is our score indicator. We will draw this on our screen, but we don't want it to move around the screen when the camera scrolls around. To achieve this we will use two different cameras, and move the world space camera, but not move the screen space camera.

In our code, we will call this screen space camera, `gui_camera`

Let's go ahead and add a variable for our new camera and initialize it in `setup`. We will also add a variable for our score. This will just be an integer initially set to 0. We will set this in both `__init__` and `setup`.

```python
# Within __init__
self.gui_camera = None
self.score = 0

# Within setup
self.gui_camera = arcade.SimpleCamera(viewport=(0, 0, width, height))
self.score = 0
```

Now we can go into our `on_update` function, and when the player collects a coin, we can increment our score variable. For now we will give the player 75 points for collecting a coin. You can change this, or as an exercise try adding different types of coins with different point values. In later chapters we'll explore dynamically providing point values for coins from a map editor.

```python
# Within on_update
for coin in coin_hit_list:
    coin.remove_from_sprite_lists()
    arcade.play_sound(self.collect_coin_sound)
    self.score += 75
```

Now that we're incrementing our score, how do we draw it onto the screen? Well we will be using our GUI camera, but so far we haven't talked about drawing Text in Arcade. There are a couple of ways we can do this in Arcade, the first way is using the *`arcade.draw_text()`* function. This is a simple function that you can put directly in `on_draw` to draw a string of text.

This function however, is not very performant, and there is a better way. We will instead use *`arcade.Text`* objects. These have many advantages, like not needing to re-calculate the text everytime it's drawn, and also can be batch drawn much like how we do with Sprite and SpriteList. We will explore batch drawing Text later.

For now, let's create an *`arcade.Text`* object to hold our score text. First create the empty variable in `__init__` and initialize in `setup`.

```python
# Within __init__
self.score_text = None

# Within setup
self.score_text = arcade.Text(f"Score: {self.score}", start_x = 0, start_y = 5)
```

The first parameter we send to `arcade.Text` is a String containing the text we want to draw. In our example we provide an f-string which adds our value from `self.score` into the text. The other parameters are defining the bottom left point that our text will be drawn at.

I've set it to draw in the bottom left of our screen here. You can try moving it around.

Now we need to add this to our `on_draw` function in order to get it to display on the screen.

```python
# Within on_draw
self.gui_camera.use()
self.score_text.draw()
```

This will now draw our text in the bottom left of the screen. However, we stil have one problem left, we're not updating the text when our user gets a new score. In order to do this we will go back to our `on_update` function, where we incremented the score when the user collects a coin, and add one more line to it:

```python
for coin in coin_hit_list:
    coin.remove_from_sprite_lists()
    arcade.play_sound(self.collect_coin_sound)
    self.score += 75
    self.score_text.text = f"Score: {self.score}"
```

In this new line we're udpating the actual text of our Text object to contain the new score value.

8.10.1 Source Code

Listing 10: Multiple Levels

```python
"""
Platformer Game

python -m arcade.examples.platform_tutorial.10_score
"""
import arcade

# Constants
SCREEN_WIDTH = 800
SCREEN_HEIGHT = 600
SCREEN_TITLE = "Platformer"

# Constants used to scale our sprites from their original size
TILE_SCALING = 0.5
COIN_SCALING = 0.5

# Movement speed of player, in pixels per frame
PLAYER_MOVEMENT_SPEED = 5
GRAVITY = 1
PLAYER_JUMP_SPEED = 20


class MyGame(arcade.Window):
    """
    Main application class.
    """

    def __init__(self):

        # Call the parent class and set up the window
        super().__init__(SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_TITLE)

```

(continues on next page)

(continued from previous page)

```
        # Variable to hold our texture for our player
        self.player_texture = None

        # Separate variable that holds the player sprite
        self.player_sprite = None

        # SpriteList for our player
        self.player_list = None

        # SpriteList for our boxes and ground
        # Putting our ground and box Sprites in the same SpriteList
        # will make it easier to perform collision detection against
        # them later on. Setting the spatial hash to True will make
        # collision detection much faster if the objects in this
        # SpriteList do not move.
        self.wall_list = None

        # SpriteList for coins the player can collect
        self.coin_list = None

        # A variable to store our camera object
        self.camera = None

        # A variable to store our gui camera object
        self.gui_camera = None

        # This variable will store our score as an integer.
        self.score = 0

        # This variable will store the text for score that we will draw to the screen.
        self.score_text = None

        # Load sounds
        self.collect_coin_sound = arcade.load_sound(":resources:sounds/coin1.wav")
        self.jump_sound = arcade.load_sound(":resources:sounds/jump1.wav")

    def setup(self):
        """Set up the game here. Call this function to restart the game."""
        self.player_texture = arcade.load_texture(":resources:images/animated_characters/
↪female_adventurer/femaleAdventurer_idle.png")

        self.player_sprite = arcade.Sprite(self.player_texture)
        self.player_sprite.center_x = 64
        self.player_sprite.center_y = 128

        self.player_list = arcade.SpriteList()
        self.player_list.append(self.player_sprite)

        self.wall_list = arcade.SpriteList(use_spatial_hash=True)
        self.coin_list = arcade.SpriteList(use_spatial_hash=True)

        # Create the ground
```

(continues on next page)

(continued from previous page)

```
        # This shows using a loop to place multiple sprites horizontally
        for x in range(0, 1250, 64):
            wall = arcade.Sprite(":resources:images/tiles/grassMid.png", scale=TILE_
↪SCALING)
            wall.center_x = x
            wall.center_y = 32
            self.wall_list.append(wall)

        # Put some crates on the ground
        # This shows using a coordinate list to place sprites
        coordinate_list = [[512, 96], [256, 96], [768, 96]]

        for coordinate in coordinate_list:
            # Add a crate on the ground
            wall = arcade.Sprite(
                ":resources:images/tiles/boxCrate_double.png", scale=TILE_SCALING
            )
            wall.position = coordinate
            self.wall_list.append(wall)

        # Add coins to the world
        for x in range(128, 1250, 256):
            coin = arcade.Sprite(":resources:images/items/coinGold.png", scale=COIN_
↪SCALING)
            coin.center_x = x
            coin.center_y = 96
            self.coin_list.append(coin)

        # Create a Platformer Physics Engine, this will handle moving our
        # player as well as collisions between the player sprite and
        # whatever SpriteList we specify for the walls.
        # It is important to supply static to the walls parameter. There is a
        # platforms parameter that is intended for moving platforms.
        # If a platform is supposed to move, and is added to the walls list,
        # it will not be moved.
        self.physics_engine = arcade.PhysicsEnginePlatformer(
            self.player_sprite, walls=self.wall_list, gravity_constant=GRAVITY
        )

        # Initialize our camera, setting a viewport the size of our window.
        self.camera = arcade.SimpleCamera(viewport=(0, 0, self.width, self.height))

        # Initialize our gui camera, initial settings are the same as our world camera.
        self.gui_camera = arcade.SimpleCamera(viewport=(0, 0, self.width, self.height))

        # Reset our score to 0
        self.score = 0

        # Initialize our arcade.Text object for score
        self.score_text = arcade.Text(f"Score: {self.score}", start_x = 0, start_y = 5)

        self.background_color = arcade.csscolor.CORNFLOWER_BLUE
```

(continues on next page)

(continued from previous page)

```python

    def on_draw(self):
        """Render the screen."""

        # Clear the screen to the background color
        self.clear()

        # Activate our camera before drawing
        self.camera.use()

        # Draw our sprites
        self.player_list.draw()
        self.wall_list.draw()
        self.coin_list.draw()

        # Activate our GUI camera
        self.gui_camera.use()

        # Draw our Score
        self.score_text.draw()

    def on_update(self, delta_time):
        """Movement and Game Logic"""

        # Move the player using our physics engine
        self.physics_engine.update()

        # See if we hit any coins
        coin_hit_list = arcade.check_for_collision_with_list(
            self.player_sprite, self.coin_list
        )

        # Loop through each coin we hit (if any) and remove it
        for coin in coin_hit_list:
            # Remove the coin
            coin.remove_from_sprite_lists()
            arcade.play_sound(self.collect_coin_sound)
            self.score += 75
            self.score_text.text = f"Score: {self.score}"

        # Center our camera on the player
        self.camera.center(self.player_sprite.position)

    def on_key_press(self, key, modifiers):
        """Called whenever a key is pressed."""

        if key == arcade.key.ESCAPE:
            self.setup()

        if key == arcade.key.UP or key == arcade.key.W:
            if self.physics_engine.can_jump():
                self.player_sprite.change_y = PLAYER_JUMP_SPEED
```

(continues on next page)

(continued from previous page)

```
                arcade.play_sound(self.jump_sound)

        if key == arcade.key.LEFT or key == arcade.key.A:
            self.player_sprite.change_x = -PLAYER_MOVEMENT_SPEED
        elif key == arcade.key.RIGHT or key == arcade.key.D:
            self.player_sprite.change_x = PLAYER_MOVEMENT_SPEED

    def on_key_release(self, key, modifiers):
        """Called whenever a key is released."""

        if key == arcade.key.LEFT or key == arcade.key.A:
            self.player_sprite.change_x = 0
        elif key == arcade.key.RIGHT or key == arcade.key.D:
            self.player_sprite.change_x = 0


def main():
    """Main function"""
    window = MyGame()
    window.setup()
    arcade.run()


if __name__ == "__main__":
    main()
```

8.10.2 Run This Chapter

```
python -m arcade.examples.platform_tutorial.10_score
```

8.11 Step 11 - Using a Scene

So far in our game, we have three SpriteLists. One for our player, one for our walls(ground and boxes), and one for our coins. This is still manageable, but whatabout as our game grows? You can probably imagine a game could end up with hundreds of SpriteLists. Using just our current approach, we would have to keep track of variables for each one, and ensure we're drawing them in the proper order.

Arcade provides a better way to handle this, with the *arcade.Scene* class. This class will hold all of our spritelists for us, allow us to create new ones, change around the order they get drawn in, and more. In later chapters we will we use a special function to load a map from a map editor tool, and automatically create a Scene based on the map.

At the end of this chapter, you will have the same result as before, but the code will be a bit different to use the Scene object.

First-off, we can remove all of our SpriteList variables from `__init__` and replace them with on variable to hold the scene object:

```
self.scene = None
```

Now at the very top of our `setup` function we can initialize the scene by doing:

```
self.scene = arcade.Scene()
```

Next, we will remove the line in `setup` that initializes our Player spritelist, that line looked like this:

```
self.player_list = arcade.SpriteList()
```

Then, instead of adding our player to the SpriteList using `self.player_sprite.append()`. We will add the player to the Scene directly:

```
self.player_sprite = arcade.Sprite(self.player_texture)
self.player_sprite.center_x = 64
self.player_sprite.center_y = 128
self.scene.add_sprite("Player", self.player_sprite)
```

Let's analyze what happens when we do *`arcade.Scene.add_sprite()`*. The first parameter to it is a String, this defines the layer name that we want to add a Sprite to. This can be an already existing layer or a new one. If the layer already exists, the Sprite will be added to it, and if it doesn't, Scene will automatically create it. Under the hood, a layer is just a SpriteList with a name. So when we specify `Player` as our Layer. Scene is creating a new SpriteList, giving it that name, and then adding our Player Sprite to it.

Next we will replace our initialization of the wall and coin SpriteLists with these functions:

```
self.scene.add_sprite_list("Walls", use_spatial_hash=True)
self.scene.add_sprite_list("Coins", use_spatial_hash=True)
```

Here we are taking a little bit different approach than we did for our `Player` layer. For our player, we just added a Sprite directly. Here we are initialization new empty layers, named `Walls` and `Coins`. The advantage to this approach is that we can specify that this layer should use spatial hashing, like we specified for those SpriteLists before.

Now when we use the `add_sprite` function on these lists later, those Sprites will be added into these existing layers.

In order to add Sprites to these, let's modify the `self.wall_list.append()` functions within the for loops for placing our walls and coins in the `setup` function. The only part we're actually changing of these loops is the last line where we were adding it to the SpriteList, but I've included the loops so you can see where all it should be changed.

```
# Create the ground
for x in range(0, 1250, 64):
    wall = arcade.Sprite(":resources:images/tiles/grassMid.png", scale=TILE_SCALING)
    wall.center_x = x
    wall.center_y = 32
    self.scene.add_sprite("Walls", wall)

# Putting Crates on the Ground
coordinate_list = [[512, 96], [256, 96], [768, 96]]

for coordinate in coordinate_li
    wall = arcade.Sprite(
        ":resources:images/tiles/boxCrate_double.png", scale=TILE_SCALING
    )
    wall.position = coordinate
    self.scene.add_sprite("Walls", wall)

# Add coins to the world
for x in range(128, 1250, 256):
    coin = arcade.Sprite(":resources:images/items/coinGold.png", scale=COIN_SCALING)
```

(continues on next page)

(continued from previous page)

```
    coin.center_x = x
    coin.center_y = 96
    self.scene.add_sprite("Coins", coin)
```

The next thing we need to do is fix our Physics Engine. If you remember back in Chapter 4, we added a physics engine and sent our Wall spritelist to in the `walls` parameter.

We'll need to modify that our PhysicsEnginePlatformer initialization to this:

```
self.physics_engine = arcade.PhysicsEnginePlatformer(
    self.player_sprite, walls=self.scene["Walls"], gravity_constant=GRAVITY
)
```

This is mostly the same as before, but we are pulling the Walls SpriteList from our Scene. If you are familiar with Python dictionaries, the *arcade.Scene* class can be interacted with in a very similar way. You can get any specific SpriteList within the scene by passing the name in brackets to the scene.

We need to also change our *arcade.check_for_collision_with_list()* function in `on_update` that we are using to get the coins we hit to use this new syntax.

```
coin_hit_list = arcade.check_for_collision_with_list(
    self.player_sprite, self.scene["Coins"]
)
```

The last thing that we need to do is update our `on_draw` function. In here we will remove all our SpriteLists draws, and replace them with one line drawing our Scene.

```
self.scene.draw()
```

Note: Make sure to keep this after our world camera is activated and before our GUI camera is activated. If you draw the scene while the GUI camera is activated, the centering on the player and scrolling will not work.

8.11.1 Source Code

Listing 11: Using a Scene

```
"""
Platformer Game

python -m arcade.examples.platform_tutorial.11_scene
"""
import arcade

# Constants
SCREEN_WIDTH = 800
SCREEN_HEIGHT = 600
SCREEN_TITLE = "Platformer"

# Constants used to scale our sprites from their original size
TILE_SCALING = 0.5
COIN_SCALING = 0.5
```

(continues on next page)

(continued from previous page)

```python

# Movement speed of player, in pixels per frame
PLAYER_MOVEMENT_SPEED = 5
GRAVITY = 1
PLAYER_JUMP_SPEED = 20


class MyGame(arcade.Window):
    """
    Main application class.
    """

    def __init__(self):

        # Call the parent class and set up the window
        super().__init__(SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_TITLE)

        # Variable to hold our texture for our player
        self.player_texture = None

        # Separate variable that holds the player sprite
        self.player_sprite = None

        # Replacing all of our SpriteLists with a Scene variable
        self.scene = None

        # A variable to store our camera object
        self.camera = None

        # A variable to store our gui camera object
        self.gui_camera = None

        # This variable will store our score as an integer.
        self.score = 0

        # This variable will store the text for score that we will draw to the screen.
        self.score_text = None

        # Load sounds
        self.collect_coin_sound = arcade.load_sound(":resources:sounds/coin1.wav")
        self.jump_sound = arcade.load_sound(":resources:sounds/jump1.wav")

    def setup(self):
        """Set up the game here. Call this function to restart the game."""
        self.scene = arcade.Scene()

        self.player_texture = arcade.load_texture(":resources:images/animated_characters/
↪female_adventurer/femaleAdventurer_idle.png")

        self.player_sprite = arcade.Sprite(self.player_texture)
        self.player_sprite.center_x = 64
        self.player_sprite.center_y = 128
```

(continues on next page)

(continued from previous page)

```python
        self.scene.add_sprite("Player", self.player_sprite)

        self.scene.add_sprite_list("Walls", use_spatial_hash=True)
        self.scene.add_sprite_list("Coins", use_spatial_hash=True)

        # Create the ground
        # This shows using a loop to place multiple sprites horizontally
        for x in range(0, 1250, 64):
            wall = arcade.Sprite(":resources:images/tiles/grassMid.png", scale=TILE_
↪SCALING)
            wall.center_x = x
            wall.center_y = 32
            self.scene.add_sprite("Walls", wall)

        # Put some crates on the ground
        # This shows using a coordinate list to place sprites
        coordinate_list = [[512, 96], [256, 96], [768, 96]]

        for coordinate in coordinate_list:
            # Add a crate on the ground
            wall = arcade.Sprite(
                ":resources:images/tiles/boxCrate_double.png", scale=TILE_SCALING
            )
            wall.position = coordinate
            self.scene.add_sprite("Walls", wall)

        # Add coins to the world
        for x in range(128, 1250, 256):
            coin = arcade.Sprite(":resources:images/items/coinGold.png", scale=COIN_
↪SCALING)
            coin.center_x = x
            coin.center_y = 96
            self.scene.add_sprite("Coins", coin)

        # Create a Platformer Physics Engine, this will handle moving our
        # player as well as collisions between the player sprite and
        # whatever SpriteList we specify for the walls.
        # It is important to supply static to the walls parameter. There is a
        # platforms parameter that is intended for moving platforms.
        # If a platform is supposed to move, and is added to the walls list,
        # it will not be moved.
        self.physics_engine = arcade.PhysicsEnginePlatformer(
            self.player_sprite, walls=self.scene["Walls"], gravity_constant=GRAVITY
        )

        # Initialize our camera, setting a viewport the size of our window.
        self.camera = arcade.SimpleCamera(viewport=(0, 0, self.width, self.height))

        # Initialize our gui camera, initial settings are the same as our world camera.
        self.gui_camera = arcade.SimpleCamera(viewport=(0, 0, self.width, self.height))

        # Reset our score to 0
```

(continues on next page)

(continued from previous page)

```python
        self.score = 0

        # Initialize our arcade.Text object for score
        self.score_text = arcade.Text(f"Score: {self.score}", start_x = 0, start_y = 5)

        self.background_color = arcade.csscolor.CORNFLOWER_BLUE

    def on_draw(self):
        """Render the screen."""

        # Clear the screen to the background color
        self.clear()

        # Activate our camera before drawing
        self.camera.use()

        # Draw our Scene
        self.scene.draw()

        # Activate our GUI camera
        self.gui_camera.use()

        # Draw our Score
        self.score_text.draw()

    def on_update(self, delta_time):
        """Movement and Game Logic"""

        # Move the player using our physics engine
        self.physics_engine.update()

        # See if we hit any coins
        coin_hit_list = arcade.check_for_collision_with_list(
            self.player_sprite, self.scene["Coins"]
        )

        # Loop through each coin we hit (if any) and remove it
        for coin in coin_hit_list:
            # Remove the coin
            coin.remove_from_sprite_lists()
            arcade.play_sound(self.collect_coin_sound)
            self.score += 75
            self.score_text.text = f"Score: {self.score}"

        # Center our camera on the player
        self.camera.center(self.player_sprite.position)

    def on_key_press(self, key, modifiers):
        """Called whenever a key is pressed."""

        if key == arcade.key.ESCAPE:
            self.setup()
```

(continues on next page)

(continued from previous page)

```python

        if key == arcade.key.UP or key == arcade.key.W:
            if self.physics_engine.can_jump():
                self.player_sprite.change_y = PLAYER_JUMP_SPEED
                arcade.play_sound(self.jump_sound)

        if key == arcade.key.LEFT or key == arcade.key.A:
            self.player_sprite.change_x = -PLAYER_MOVEMENT_SPEED
        elif key == arcade.key.RIGHT or key == arcade.key.D:
            self.player_sprite.change_x = PLAYER_MOVEMENT_SPEED

    def on_key_release(self, key, modifiers):
        """Called whenever a key is released."""

        if key == arcade.key.LEFT or key == arcade.key.A:
            self.player_sprite.change_x = 0
        elif key == arcade.key.RIGHT or key == arcade.key.D:
            self.player_sprite.change_x = 0


def main():
    """Main function"""
    window = MyGame()
    window.setup()
    arcade.run()


if __name__ == "__main__":
    main()
```

8.11.2 Run This Chapter

```
python -m arcade.examples.platform_tutorial.11_scene
```

8.12 Step 12 - Loading a Map From a Map Editor

In this chapter we will start using a map editor called Tiled. Tiled is a popular 2D map editor, it can be used with any game engine, but Arcade has specific integrations for working with Tiled.

We'll explore how to load maps from Tiled in this tutorial using Arcade's built-in `arcade.TileMap` class using some maps from the built-in resources that Arcade comes with. We won't cover actually building a map in Tiled this tutorial, but if you want to learn more about Tiled check out the resources below:

- Download Tiled: https://www.mapeditor.org/
- Tiled's Documentation: https://doc.mapeditor.org/en/stable/

You won't actually need Tiled to continue following this tutorial. We will be using all pre-built maps included with Arcade. However if you want to experiment with your own maps or changing things, I recommend getting Tiled and getting familiar with it, it is a really useful tool for 2D Game Development.

To start off with, we're going to remove a bunch of code. Namely we'll remove the creation of our ground, boxes, and coin sprites(We'll leave the player one). Go ahead and remove the following blocks of code from the `setup` function.

```python
self.scene.add_sprite_list("Walls", use_spatial_hash=True)
self.scene.add_sprite_list("Coins", use_spatial_hash=True)

for x in range(0, 1250, 64):
    wall = arcade.Sprite(":resources:images/tiles/grassMid.png", scale=TILE_SCALING)
    wall.center_x = x
    wall.center_y = 32
    self.scene.add_sprite("Walls", wall)

coordinate_list = [[512, 96], [256, 96], [768, 96]]

for coordinate in coordinate_list:
    wall = arcade.Sprite(
        ":resources:images/tiles/boxCrate_double.png", scale=TILE_SCALING
    )
    wall.position = coordinate
    self.scene.add_sprite("Walls", wall)

for x in range(128, 1250, 256):
    coin = arcade.Sprite(":resources:images/items/coinGold.png", scale=COIN_SCALING)
    coin.center_x = x
    coin.center_y = 96
    self.scene.add_sprite("Coins", coin)
```

These things will now be handled by our map file automatically once we start loading it.

In order to load our map, we will first create a variable for it in `__init__`:

```python
self.tile_map = None
```

Next we will load our map in our `setup` function, and then create a Scene from it using a built-in function Arcade provides. This will give us a drawable scene completely based off of the map file automatically. This code will all go at the top of the `setup` function.

Make sure to replace the line that sets `self.scene` with the new one below.

```python
layer_options = {
    "Platforms": {
        "use_spatial_hash": True
    }
}

self.tile_map = arcade.load_tilemap(
    ":resources:tiled_maps/map.json",
    scaling=TILE_SCALING,
    layer_options=layer_options
)

self.scene = arcade.Scene.from_tilemap(self.tile_map)
```

This code will load in our built-in Tiled Map and automatically build a Scene from it. The Scene at this stage is ready for drawing and we don't need to do anything else to it(other than add our player).

Note: What is `layer_options` and where are those values in it coming from?

`layer_options` is a special dictionary that can be provided to the `load_tilemap` function. This will send special options for each layer into the map loader. In this example our map has a layer called `Platforms`, and we want to enable spatial hashing on it. Much like we did for our `wall` SpriteList before. For more info on the layer options dictionary and the available keys, check out :class`arcade.TileMap`

At this point we only have one piece of code left to change. In switching to our new map, you may have noticed by the `layer_options` dictionary that we now have a layer named `Platforms`. Previously in our Scene we were calling this layer `Walls`. We'll need to go update that reference when we create our Physics Engine.

In the `setup` function update the Physics Engine creation to use the the new `Platforms` layer:

```python
self.physics_engine = arcade.PhysicsEnginePlatformer(
    self.player_sprite, walls=self.scene["Platforms"], gravity_constant=GRAVITY
)
```

8.12.1 Source Code

Listing 12: Loading a Map From a Map Editor

```python
"""
Platformer Game

python -m arcade.examples.platform_tutorial.12_tiled
"""
import arcade

# Constants
SCREEN_WIDTH = 800
SCREEN_HEIGHT = 600
SCREEN_TITLE = "Platformer"

# Constants used to scale our sprites from their original size
TILE_SCALING = 0.5
COIN_SCALING = 0.5

# Movement speed of player, in pixels per frame
PLAYER_MOVEMENT_SPEED = 5
GRAVITY = 1
PLAYER_JUMP_SPEED = 20


class MyGame(arcade.Window):
    """
    Main application class.
    """

    def __init__(self):

        # Call the parent class and set up the window
```

(continues on next page)

(continued from previous page)

```
        super().__init__(SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_TITLE)

        # Variable to hold our texture for our player
        self.player_texture = None

        # Separate variable that holds the player sprite
        self.player_sprite = None

        # Variable to hold our Tiled Map
        self.tile_map = None

        # Replacing all of our SpriteLists with a Scene variable
        self.scene = None

        # A variable to store our camera object
        self.camera = None

        # A variable to store our gui camera object
        self.gui_camera = None

        # This variable will store our score as an integer.
        self.score = 0

        # This variable will store the text for score that we will draw to the screen.
        self.score_text = None

        # Load sounds
        self.collect_coin_sound = arcade.load_sound(":resources:sounds/coin1.wav")
        self.jump_sound = arcade.load_sound(":resources:sounds/jump1.wav")

    def setup(self):
        """Set up the game here. Call this function to restart the game."""
        layer_options = {
            "Platforms": {
                "use_spatial_hash": True
            }
        }

        # Load our TileMap
        self.tile_map = arcade.load_tilemap(":resources:tiled_maps/map.json",␣
↪scaling=TILE_SCALING, layer_options=layer_options)

        # Create our Scene Based on the TileMap
        self.scene = arcade.Scene.from_tilemap(self.tile_map)

        self.player_texture = arcade.load_texture(":resources:images/animated_characters/
↪female_adventurer/femaleAdventurer_idle.png")

        self.player_sprite = arcade.Sprite(self.player_texture)
        self.player_sprite.center_x = 128
        self.player_sprite.center_y = 128
        self.scene.add_sprite("Player", self.player_sprite)
```

(continues on next page)

(continued from previous page)

```python

        # Create a Platformer Physics Engine, this will handle moving our
        # player as well as collisions between the player sprite and
        # whatever SpriteList we specify for the walls.
        # It is important to supply static to the walls parameter. There is a
        # platforms parameter that is intended for moving platforms.
        # If a platform is supposed to move, and is added to the walls list,
        # it will not be moved.
        self.physics_engine = arcade.PhysicsEnginePlatformer(
            self.player_sprite, walls=self.scene["Platforms"], gravity_constant=GRAVITY
        )

        # Initialize our camera, setting a viewport the size of our window.
        self.camera = arcade.SimpleCamera(viewport=(0, 0, self.width, self.height))

        # Initialize our gui camera, initial settings are the same as our world camera.
        self.gui_camera = arcade.SimpleCamera(viewport=(0, 0, self.width, self.height))

        # Reset our score to 0
        self.score = 0

        # Initialize our arcade.Text object for score
        self.score_text = arcade.Text(f"Score: {self.score}", start_x = 0, start_y = 5)

        self.background_color = arcade.csscolor.CORNFLOWER_BLUE

    def on_draw(self):
        """Render the screen."""

        # Clear the screen to the background color
        self.clear()

        # Activate our camera before drawing
        self.camera.use()

        # Draw our Scene
        self.scene.draw()

        # Activate our GUI camera
        self.gui_camera.use()

        # Draw our Score
        self.score_text.draw()

    def on_update(self, delta_time):
        """Movement and Game Logic"""

        # Move the player using our physics engine
        self.physics_engine.update()

        # See if we hit any coins
        coin_hit_list = arcade.check_for_collision_with_list(
```

(continues on next page)

(continued from previous page)

```
            self.player_sprite, self.scene["Coins"]
        )

        # Loop through each coin we hit (if any) and remove it
        for coin in coin_hit_list:
            # Remove the coin
            coin.remove_from_sprite_lists()
            arcade.play_sound(self.collect_coin_sound)
            self.score += 75
            self.score_text.text = f"Score: {self.score}"

        # Center our camera on the player
        self.camera.center(self.player_sprite.position)

    def on_key_press(self, key, modifiers):
        """Called whenever a key is pressed."""

        if key == arcade.key.ESCAPE:
            self.setup()

        if key == arcade.key.UP or key == arcade.key.W:
            if self.physics_engine.can_jump():
                self.player_sprite.change_y = PLAYER_JUMP_SPEED
                arcade.play_sound(self.jump_sound)

        if key == arcade.key.LEFT or key == arcade.key.A:
            self.player_sprite.change_x = -PLAYER_MOVEMENT_SPEED
        elif key == arcade.key.RIGHT or key == arcade.key.D:
            self.player_sprite.change_x = PLAYER_MOVEMENT_SPEED

    def on_key_release(self, key, modifiers):
        """Called whenever a key is released."""

        if key == arcade.key.LEFT or key == arcade.key.A:
            self.player_sprite.change_x = 0
        elif key == arcade.key.RIGHT or key == arcade.key.D:
            self.player_sprite.change_x = 0


def main():
    """Main function"""
    window = MyGame()
    window.setup()
    arcade.run()


if __name__ == "__main__":
    main()
```

8.13 Step 13 - More Types of Layers

For this example, we'll switch to a different built-in map that has more layers we can do things with.

In our setup function, load this map instead of the one from Chapter 12:

```
self.tile_map = arcade.load_tilemap(":resources:tiled_maps/map2_level_1.json",␣
↪scaling=TILE_SCALING, layer_options=layer_options)
```

You can run this and check out the map we will be working with this chapter. You'll notice in addition to the normal platforms and coins we've had. We now have some extra signs and decoration objects, as well as a pit of lava.

Back in chapter 6 we made use of our `setup` function to reset the game. Let's go ahead and use that system here to reset the game when the player touches the lava pit. You can remove the section for resetting when the Escape key is pressed if you want, or you can leave it in place. We can also play a game over sound when this happens.

Let's first add a new sound to our `__init__` function for this:

```
self.gameover_sound = arcade.load_sound(":resources:sounds/gameover1.wav")
```

In order to do this, we'll add this code in our `on_update` function:

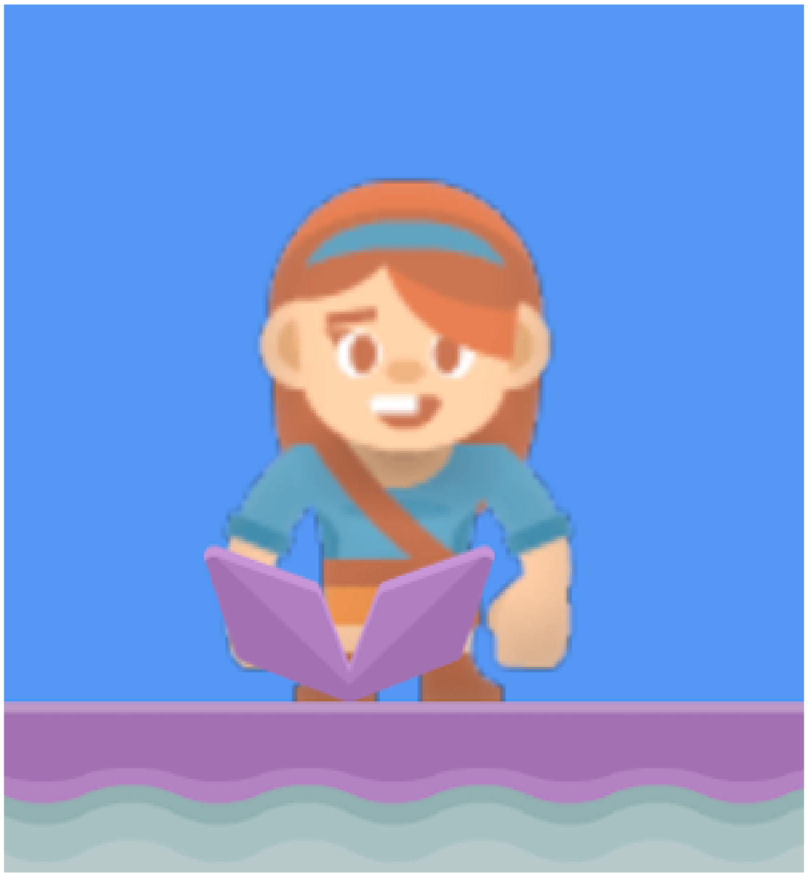
```
if arcade.check_for_collision_with_list(
    self.player_sprite, self.scene["Don't Touch"]
):
    arcade.play_sound(self.gameover_sound)
    self.setup()
```

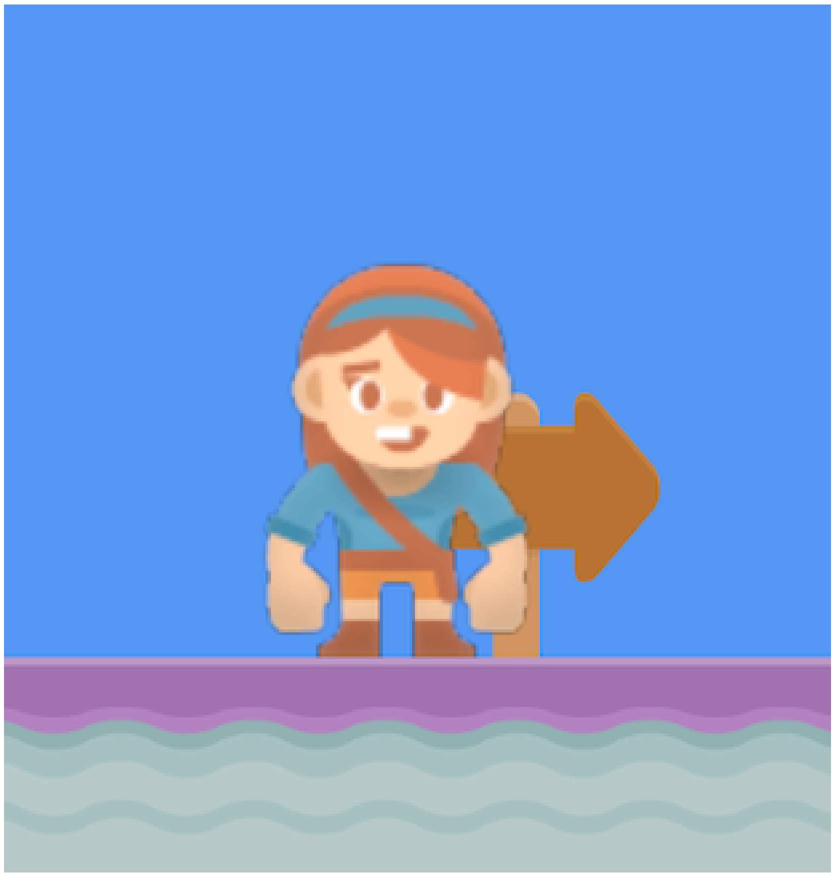
The map we are using here has some extra layers in it we haven't used yet. In the code above we made use of the `Don't Touch` to reset the game when the player touches it. In this section we will make use of two other layers in our new map, `Background` and `Foreground`.

We will use these layers as a way to separate objects that should be drawn in front of our player, and objects that should be drawn behind the player. In our `setup` function, before we create the player sprite, add this code.

```
self.scene.add_sprite_list_after("Player", "Foreground")
```

This code will cause our player spritelist to be inserted at a specific point in the Scene. Causing spritelists which are in front of it to be drawn before it, and ones behind it to be drawn after. By doing this we can make objects appear to be in front of or behind our player like the images below:





8.13.1 Source Code

Listing 13: More Layers

```python
"""
Platformer Game

python -m arcade.examples.platform_tutorial.13_more_layers
"""
import arcade

# Constants
SCREEN_WIDTH = 800
SCREEN_HEIGHT = 600
SCREEN_TITLE = "Platformer"

# Constants used to scale our sprites from their original size
TILE_SCALING = 0.5
COIN_SCALING = 0.5

# Movement speed of player, in pixels per frame
PLAYER_MOVEMENT_SPEED = 5
GRAVITY = 1
PLAYER_JUMP_SPEED = 20
```

(continues on next page)

(continued from previous page)

```python


class MyGame(arcade.Window):
    """
    Main application class.
    """

    def __init__(self):

        # Call the parent class and set up the window
        super().__init__(SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_TITLE)

        # Variable to hold our texture for our player
        self.player_texture = None

        # Separate variable that holds the player sprite
        self.player_sprite = None

        # Variable to hold our Tiled Map
        self.tile_map = None

        # Replacing all of our SpriteLists with a Scene variable
        self.scene = None

        # A variable to store our camera object
        self.camera = None

        # A variable to store our gui camera object
        self.gui_camera = None

        # This variable will store our score as an integer.
        self.score = 0

        # This variable will store the text for score that we will draw to the screen.
        self.score_text = None

        # Load sounds
        self.collect_coin_sound = arcade.load_sound(":resources:sounds/coin1.wav")
        self.jump_sound = arcade.load_sound(":resources:sounds/jump1.wav")
        self.gameover_sound = arcade.load_sound(":resources:sounds/gameover1.wav")

    def setup(self):
        """Set up the game here. Call this function to restart the game."""
        layer_options = {
            "Platforms": {
                "use_spatial_hash": True
            }
        }

        # Load our TileMap
        self.tile_map = arcade.load_tilemap(":resources:tiled_maps/map2_level_1.json",␣
↪scaling=TILE_SCALING, layer_options=layer_options)
```

(continues on next page)

(continued from previous page)

```python

        # Create our Scene Based on the TileMap
        self.scene = arcade.Scene.from_tilemap(self.tile_map)

        self.player_texture = arcade.load_texture(":resources:images/animated_characters/
↪female_adventurer/femaleAdventurer_idle.png")

        # Add Player Spritelist before "Foreground" layer. This will make the foreground
        # be drawn after the player, making it appear to be in front of the Player.
        # Setting before using scene.add_sprite allows us to define where the SpriteList
        # will be in the draw order. If we just use add_sprite, it will be appended to␣
↪the
        # end of the order.
        self.scene.add_sprite_list_after("Player", "Foreground")

        self.player_sprite = arcade.Sprite(self.player_texture)
        self.player_sprite.center_x = 128
        self.player_sprite.center_y = 128
        self.scene.add_sprite("Player", self.player_sprite)

        # Create a Platformer Physics Engine, this will handle moving our
        # player as well as collisions between the player sprite and
        # whatever SpriteList we specify for the walls.
        # It is important to supply static to the walls parameter. There is a
        # platforms parameter that is intended for moving platforms.
        # If a platform is supposed to move, and is added to the walls list,
        # it will not be moved.
        self.physics_engine = arcade.PhysicsEnginePlatformer(
            self.player_sprite, walls=self.scene["Platforms"], gravity_constant=GRAVITY
        )

        # Initialize our camera, setting a viewport the size of our window.
        self.camera = arcade.SimpleCamera(viewport=(0, 0, self.width, self.height))

        # Initialize our gui camera, initial settings are the same as our world camera.
        self.gui_camera = arcade.SimpleCamera(viewport=(0, 0, self.width, self.height))

        # Reset our score to 0
        self.score = 0

        # Initialize our arcade.Text object for score
        self.score_text = arcade.Text(f"Score: {self.score}", start_x = 0, start_y = 5)

        self.background_color = arcade.csscolor.CORNFLOWER_BLUE

    def on_draw(self):
        """Render the screen."""

        # Clear the screen to the background color
        self.clear()

        # Activate our camera before drawing
```

(continues on next page)

(continued from previous page)

```python
        self.camera.use()

        # Draw our Scene
        self.scene.draw()

        # Activate our GUI camera
        self.gui_camera.use()

        # Draw our Score
        self.score_text.draw()

    def on_update(self, delta_time):
        """Movement and Game Logic"""

        # Move the player using our physics engine
        self.physics_engine.update()

        # See if we hit any coins
        coin_hit_list = arcade.check_for_collision_with_list(
            self.player_sprite, self.scene["Coins"]
        )

        # Loop through each coin we hit (if any) and remove it
        for coin in coin_hit_list:
            # Remove the coin
            coin.remove_from_sprite_lists()
            arcade.play_sound(self.collect_coin_sound)
            self.score += 75
            self.score_text.text = f"Score: {self.score}"

        if arcade.check_for_collision_with_list(
            self.player_sprite, self.scene["Don't Touch"]
        ):
            arcade.play_sound(self.gameover_sound)
            self.setup()

        # Center our camera on the player
        self.camera.center(self.player_sprite.position)

    def on_key_press(self, key, modifiers):
        """Called whenever a key is pressed."""

        if key == arcade.key.ESCAPE:
            self.setup()

        if key == arcade.key.UP or key == arcade.key.W:
            if self.physics_engine.can_jump():
                self.player_sprite.change_y = PLAYER_JUMP_SPEED
                arcade.play_sound(self.jump_sound)

        if key == arcade.key.LEFT or key == arcade.key.A:
            self.player_sprite.change_x = -PLAYER_MOVEMENT_SPEED
```

(continues on next page)

(continued from previous page)

```
        elif key == arcade.key.RIGHT or key == arcade.key.D:
            self.player_sprite.change_x = PLAYER_MOVEMENT_SPEED

    def on_key_release(self, key, modifiers):
        """Called whenever a key is released."""

        if key == arcade.key.LEFT or key == arcade.key.A:
            self.player_sprite.change_x = 0
        elif key == arcade.key.RIGHT or key == arcade.key.D:
            self.player_sprite.change_x = 0


def main():
    """Main function"""
    window = MyGame()
    window.setup()
    arcade.run()


if __name__ == "__main__":
    main()
```

8.14 Step 14 - Multiple Levels

Now we will make it so that our game has multiple levels. For now we will just have two levels, but this technique can be easily expanded to include more.

To start off, create two new variables in the `__init__` function to represent the position that marks the end of the map, and what level we should be loading.

```
# Where is the right edge of the map?
self.end_of_map = 0

# Level number to load
self.level = 1
```

Next in the `setup` function we will change the map loading call to use an f-string to load a map file depending on the level variable we created.

```
# Load our TileMap
self.tile_map = arcade.load_tilemap(f":resources:tiled_maps/map2_level_{self.level}.json
↪", scaling=TILE_SCALING, layer_options=layer_options)
```

Again in the setup function, we will calculate where the edge of the currently loaded map is, in pixels. To do this we get the width of the map, which is represented in number of tiles, and multiply it by the tile width. We also need to consider the scaling of the tiles, because we are measuring this in pixels.

```
# Calculate the right edge of the map in pixels
self.end_of_map = (self.tile_map.width * self.tile_map.tile_width) * self.tile_map.
↪scaling
```

Now in the on_update function, we will add a block to check the player position against the end of the map value. We will do this right before the center_camera_to_player function call at the end. This will increment our current level, and leverage the setup function in order to re-load the game with the new level.

```
# Check if the player got to the end of the level
if self.player_sprite.center_x >= self.end_of_map:
    # Advance to the next level
    self.level += 1

    # Reload game with new level
    self.setup()
```

If you run the game at this point, you will be able to reach the end of the first level and have the next level load and play through it. We have two problems at this point, did you notice them? The first problem is that the player's score resets in between levels, maybe you want this to happen in your game, but we will fix it here so that when switching levels we don't reset the score.

To do this, first add a new variable to the __init__ function which will serve as a trigger to know if the score should be reset or not. We want to be able to reset it when the player loses, so this trigger will help us only reset the score when we want to.

```
# Should we reset the score?
self.reset_score = True
```

Now in the setup function we can replace the score reset with this block of code. We change the reset_score variable back to True after resetting the score, because the default in our game should be to reset it, and we only turn off the reset when we want it off.

```
# Reset the score if we should
if self.reset_score:
    self.score = 0
self.reset_score = True
```

Finally, in the section of on_update that we advance the level, we can add this line to turn off the score reset

```
# Turn off score reset when advancing level
self.reset_score = False
```

Now the player's score will persist between levels, but we still have one more problem. If you reach the end of the second level, the game crashes! This is because we only actually have two levels available, but we are still trying to advance the level to 3 when we hit the end of level 2.

There's a few ways this can be handled, one way is to simply make more levels. Eventually you have to have a final level though, so this probably isn't the best solution. As an exercise, see if you can find a way to gracefully handle the final level. You could display an end screen, or restart the game from the beginning, or anything you want.

8.14.1 Source Code

Listing 14: Moving the enemies

```python
"""
Platformer Game

python -m arcade.examples.platform_tutorial.14_multiple_levels
"""
import arcade

# Constants
SCREEN_WIDTH = 800
SCREEN_HEIGHT = 600
SCREEN_TITLE = "Platformer"

# Constants used to scale our sprites from their original size
TILE_SCALING = 0.5
COIN_SCALING = 0.5

# Movement speed of player, in pixels per frame
PLAYER_MOVEMENT_SPEED = 5
GRAVITY = 1
PLAYER_JUMP_SPEED = 20


class MyGame(arcade.Window):
    """
    Main application class.
    """

    def __init__(self):

        # Call the parent class and set up the window
        super().__init__(SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_TITLE)

        # Variable to hold our texture for our player
        self.player_texture = None

        # Separate variable that holds the player sprite
        self.player_sprite = None

        # Variable to hold our Tiled Map
        self.tile_map = None

        # Replacing all of our SpriteLists with a Scene variable
        self.scene = None

        # A variable to store our camera object
        self.camera = None

        # A variable to store our gui camera object
        self.gui_camera = None
```

(continues on next page)

(continued from previous page)

```python

        # This variable will store our score as an integer.
        self.score = 0

        # This variable will store the text for score that we will draw to the screen.
        self.score_text = None

        # Where is the right edge of the map?
        self.end_of_map = 0

        # Level number to load
        self.level = 1

        # Should we reset the score?
        self.reset_score = True

        # Load sounds
        self.collect_coin_sound = arcade.load_sound(":resources:sounds/coin1.wav")
        self.jump_sound = arcade.load_sound(":resources:sounds/jump1.wav")
        self.gameover_sound = arcade.load_sound(":resources:sounds/gameover1.wav")

    def setup(self):
        """Set up the game here. Call this function to restart the game."""
        layer_options = {
            "Platforms": {
                "use_spatial_hash": True
            }
        }

        # Load our TileMap
        self.tile_map = arcade.load_tilemap(f":resources:tiled_maps/map2_level_{self.
↪level}.json", scaling=TILE_SCALING, layer_options=layer_options)

        # Create our Scene Based on the TileMap
        self.scene = arcade.Scene.from_tilemap(self.tile_map)

        self.player_texture = arcade.load_texture(":resources:images/animated_characters/
↪female_adventurer/femaleAdventurer_idle.png")

        # Add Player Spritelist before "Foreground" layer. This will make the foreground
        # be drawn after the player, making it appear to be in front of the Player.
        # Setting before using scene.add_sprite allows us to define where the SpriteList
        # will be in the draw order. If we just use add_sprite, it will be appended to␣
↪the
        # end of the order.
        self.scene.add_sprite_list_after("Player", "Foreground")

        self.player_sprite = arcade.Sprite(self.player_texture)
        self.player_sprite.center_x = 128
        self.player_sprite.center_y = 128
        self.scene.add_sprite("Player", self.player_sprite)

```

(continues on next page)

(continued from previous page)

```
        # Create a Platformer Physics Engine, this will handle moving our
        # player as well as collisions between the player sprite and
        # whatever SpriteList we specify for the walls.
        # It is important to supply static to the walls parameter. There is a
        # platforms parameter that is intended for moving platforms.
        # If a platform is supposed to move, and is added to the walls list,
        # it will not be moved.
        self.physics_engine = arcade.PhysicsEnginePlatformer(
            self.player_sprite, walls=self.scene["Platforms"], gravity_constant=GRAVITY
        )

        # Initialize our camera, setting a viewport the size of our window.
        self.camera = arcade.SimpleCamera(viewport=(0, 0, self.width, self.height))

        # Initialize our gui camera, initial settings are the same as our world camera.
        self.gui_camera = arcade.SimpleCamera(viewport=(0, 0, self.width, self.height))

        # Reset the score if we should
        if self.reset_score:
            self.score = 0
        self.reset_score = True

        # Initialize our arcade.Text object for score
        self.score_text = arcade.Text(f"Score: {self.score}", start_x = 0, start_y = 5)

        self.background_color = arcade.csscolor.CORNFLOWER_BLUE

        # Calculate the right edge of the map in pixels
        self.end_of_map = (self.tile_map.width * self.tile_map.tile_width) * self.tile_
↪map.scaling
        print(self.end_of_map)

    def on_draw(self):
        """Render the screen."""

        # Clear the screen to the background color
        self.clear()

        # Activate our camera before drawing
        self.camera.use()

        # Draw our Scene
        self.scene.draw()

        # Activate our GUI camera
        self.gui_camera.use()

        # Draw our Score
        self.score_text.draw()

    def on_update(self, delta_time):
        """Movement and Game Logic"""
```

(continues on next page)

(continued from previous page)

```python

        # Move the player using our physics engine
        self.physics_engine.update()

        # See if we hit any coins
        coin_hit_list = arcade.check_for_collision_with_list(
            self.player_sprite, self.scene["Coins"]
        )

        # Loop through each coin we hit (if any) and remove it
        for coin in coin_hit_list:
            # Remove the coin
            coin.remove_from_sprite_lists()
            arcade.play_sound(self.collect_coin_sound)
            self.score += 75
            self.score_text.text = f"Score: {self.score}"

        if arcade.check_for_collision_with_list(
            self.player_sprite, self.scene["Don't Touch"]
        ):
            arcade.play_sound(self.gameover_sound)
            self.setup()

        # Check if the player got to the end of the level
        if self.player_sprite.center_x >= self.end_of_map:
            # Advance to the next level
            self.level += 1

            # Turn off score reset when advancing level
            self.reset_score = False

            # Reload game with new level
            self.setup()

        # Center our camera on the player
        self.camera.center(self.player_sprite.position)

    def on_key_press(self, key, modifiers):
        """Called whenever a key is pressed."""

        if key == arcade.key.ESCAPE:
            self.setup()

        if key == arcade.key.UP or key == arcade.key.W:
            if self.physics_engine.can_jump():
                self.player_sprite.change_y = PLAYER_JUMP_SPEED
                arcade.play_sound(self.jump_sound)

        if key == arcade.key.LEFT or key == arcade.key.A:
            self.player_sprite.change_x = -PLAYER_MOVEMENT_SPEED
        elif key == arcade.key.RIGHT or key == arcade.key.D:
            self.player_sprite.change_x = PLAYER_MOVEMENT_SPEED
```

(continues on next page)

(continued from previous page)

```python

    def on_key_release(self, key, modifiers):
        """Called whenever a key is released."""

        if key == arcade.key.LEFT or key == arcade.key.A:
            self.player_sprite.change_x = 0
        elif key == arcade.key.RIGHT or key == arcade.key.D:
            self.player_sprite.change_x = 0


def main():
    """Main function"""
    window = MyGame()
    window.setup()
    arcade.run()


if __name__ == "__main__":
    main()
```

Currently there are a few more examples that expand beyond where the tutorial leaves off. You can see the source code for those examples as well as every chapter in the tutorial on the Arcade Github at https://github.com/pythonarcade/arcade/tree/development/arcade/examples/platform_tutorial

CHAPTER

NINE

PYMUNK PLATFORMER

This tutorial covers how to write a platformer using Arcade and its Pymunk API. This tutorial assumes the you are somewhat familiar with Python, Arcade, and the Tiled Map Editor.

- If you aren't familiar with programming in Python, check out https://learn.arcade.academy
- If you aren't familiar with the Arcade library, work through the *Simple Platformer*.
- If you aren't familiar with the Tiled Map Editor, the *Simple Platformer* also introduces how to create a map with the Tiled Map Editor.

9.1 Common Issues

There are a few items with the Pymunk physics engine that should be pointed out before you get started:

- Object overlap - A fast moving object is allowed to overlap with the object it collides with, and Pymunk will push them apart later. See collision bias for more information.
- Pass-through - A fast moving object can pass through another object if its speed is so quick it never overlaps the other object between frames. See object tunneling.
- When stepping the physics engine forward in time, the default is to move forward 1/60th of a second. Whatever increment is picked, increments should always be kept the same. Don't use the variable delta_time from the `update` method as a unit, or results will be unstable and unpredictable. For a more accurate simulation, you can step forward 1/120th of a second twice per frame. This increases the time required, but takes more time to calculate.
- A sprite moving across a floor made up of many rectangles can get "caught" on the edges. The corner of the player sprite can get caught the corner of the floor sprite. To get around this, make sure the hit box for the bottom of the player sprite is rounded. Also, look into the possibility of merging horizontal rows of sprites.

9.2 Open a Window

To begin with, let's start with a program that will use Arcade to open a blank window. It also has stubs for methods we'll fill in later. Try this code and make sure you can run it. It should pop open a black window.

Listing 1: Starting Program

```
"""
Example of Pymunk Physics Engine Platformer
"""
```

(continues on next page)

(continued from previous page)

```python
import arcade

SCREEN_TITLE = "PyMunk Platformer"

# Size of screen to show, in pixels
SCREEN_WIDTH = 800
SCREEN_HEIGHT = 600


class GameWindow(arcade.Window):
    """ Main Window """

    def __init__(self, width, height, title):
        """ Create the variables """

        # Init the parent class
        super().__init__(width, height, title)

    def setup(self):
        """ Set up everything with the game """
        pass

    def on_key_press(self, key, modifiers):
        """Called whenever a key is pressed. """
        pass

    def on_key_release(self, key, modifiers):
        """Called when the user releases a key. """
        pass

    def on_update(self, delta_time):
        """ Movement and game logic """
        pass

    def on_draw(self):
        """ Draw everything """
        self.clear()


def main():
    """ Main function """
    window = GameWindow(SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_TITLE)
    window.setup()
    arcade.run()


if __name__ == "__main__":
    main()
```

9.3 Create Constants

Now let's set up the `import` statements, and define the constants we are going to use. In this case, we've got sprite tiles that are 128x128 pixels. They are scaled down to 50% of the width and 50% of the height (scale of 0.5). The screen size is set to 25x15 grid.

To keep things simple, this example will not scroll the screen with the player. See *Simple Platformer* or sprite_move_scrolling.

When you run this program, the screen should be larger.

Listing 2: Adding some constants

```python
"""
Example of Pymunk Physics Engine Platformer
"""
import math
from typing import Optional
import arcade

SCREEN_TITLE = "PyMunk Platformer"

# How big are our image tiles?
SPRITE_IMAGE_SIZE = 128

# Scale sprites up or down
SPRITE_SCALING_PLAYER = 0.5
SPRITE_SCALING_TILES = 0.5

# Scaled sprite size for tiles
SPRITE_SIZE = int(SPRITE_IMAGE_SIZE * SPRITE_SCALING_PLAYER)

# Size of grid to show on screen, in number of tiles
SCREEN_GRID_WIDTH = 25
SCREEN_GRID_HEIGHT = 15

# Size of screen to show, in pixels
SCREEN_WIDTH = SPRITE_SIZE * SCREEN_GRID_WIDTH
SCREEN_HEIGHT = SPRITE_SIZE * SCREEN_GRID_HEIGHT


class GameWindow(arcade.Window):
```

- pymunk_demo_platformer_02
- pymunk_demo_platformer_02_diff

9.4 Create Instance Variables

Next, let's create instance variables we are going to use, and set a background color that's green: `arcade.color.AMAZON`

If you aren't familiar with type-casting on Python, you might not be familiar with lines of code like this:

```
self.player_list: Optional[arcade.SpriteList] = None
```

This means the `player_list` attribute is going to be an instance of `SpriteList` or `None`. If you don't want to mess with typing, then this code also works just as well:

```
self.player_list = None
```

Running this program should show the same window, but with a green background.

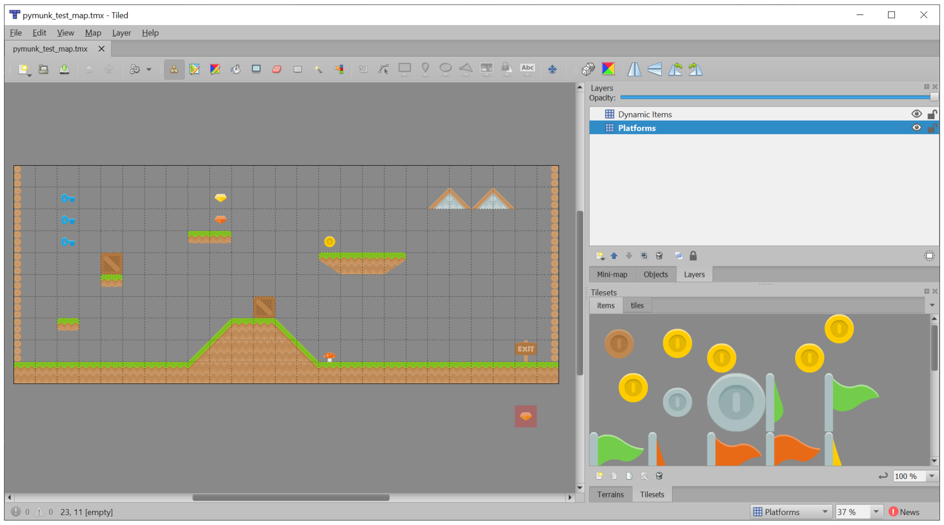
Listing 3: Create instance variables

```
class GameWindow(arcade.Window):
    """ Main Window """

    def __init__(self, width, height, title):
        """ Create the variables """

        # Init the parent class
        super().__init__(width, height, title)

        # Player sprite
        self.player_sprite: Optional[arcade.Sprite] = None

        # Sprite lists we need
        self.player_list: Optional[arcade.SpriteList] = None
        self.wall_list: Optional[arcade.SpriteList] = None
        self.bullet_list: Optional[arcade.SpriteList] = None
        self.item_list: Optional[arcade.SpriteList] = None

        # Track the current state of what key is pressed
        self.left_pressed: bool = False
        self.right_pressed: bool = False

        # Set background color
        self.background_color = arcade.color.AMAZON
```

- pymunk_demo_platformer_03
- pymunk_demo_platformer_03_diff

9.5 Load and Display Map

To get started, create a map with the Tiled Map Editor. Place items that you don't want to move, and to act as platforms in a layer named "Platforms". Place items you want to push around in a layer called "Dynamic Items". Name the file "pymunk_test_map.tmx" and place in the exact same directory as your code.



If you aren't sure how to use the Tiled Map Editor, see *Step 8 - Collecting Coins*.

Now, in the `setup` function, we are going add code to:

- Create instances of `SpriteList` for each group of sprites we are doing to work with.
- Create the player sprite.
- Read in the tiled map.
- Make sprites from the layers in the tiled map.

Note: When making sprites from the tiled map layer, the name of the layer you load must match **exactly** with the layer created in the tiled map editor. It is case-sensitive.

Listing 4: Creating our sprites

```python
    def setup(self):
        """ Set up everything with the game """

        # Create the sprite lists
        self.player_list = arcade.SpriteList()
        self.bullet_list = arcade.SpriteList()

        # Map name
        map_name = ":resources:/tiled_maps/pymunk_test_map.json"

        # Load in TileMap
        tile_map = arcade.load_tilemap(map_name, SPRITE_SCALING_TILES)

        # Pull the sprite layers out of the tile map
```

(continues on next page)

(continued from previous page)

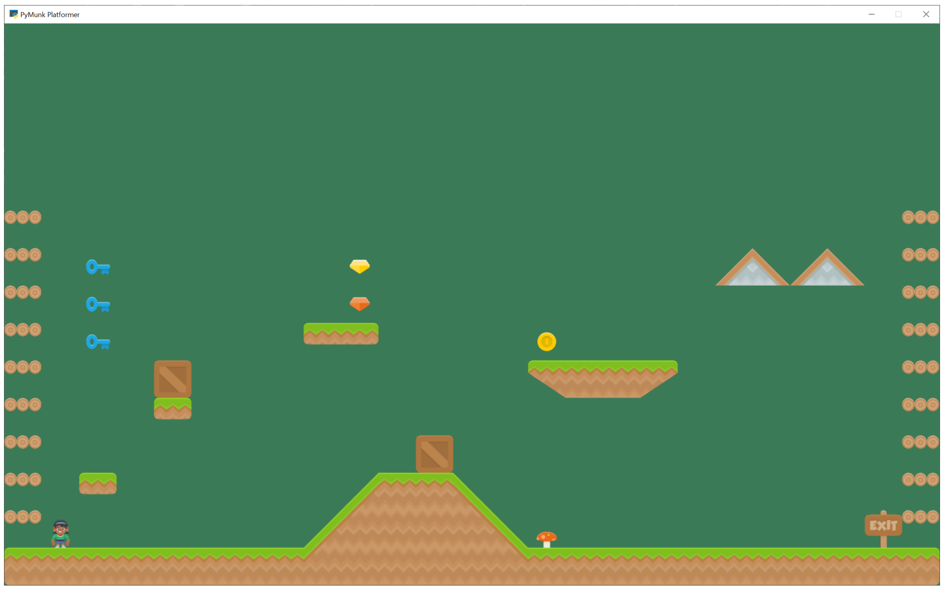
```python
        self.wall_list = tile_map.sprite_lists["Platforms"]
        self.item_list = tile_map.sprite_lists["Dynamic Items"]

        # Create player sprite
        self.player_sprite = arcade.Sprite(":resources:images/animated_characters/female_
↪person/femalePerson_idle.png",
                                           SPRITE_SCALING_PLAYER)
        # Set player location
        grid_x = 1
        grid_y = 1
        self.player_sprite.center_x = SPRITE_SIZE * grid_x + SPRITE_SIZE / 2
        self.player_sprite.center_y = SPRITE_SIZE * grid_y + SPRITE_SIZE / 2
        # Add to player sprite list
        self.player_list.append(self.player_sprite)
```

There's no point in having sprites if we don't draw them, so in the `on_draw` method, let's draw out sprite lists.

Listing 5: Drawing our sprites

```python
    def on_draw(self):
        """ Draw everything """
        self.clear()
        self.wall_list.draw()
        self.bullet_list.draw()
        self.item_list.draw()
        self.player_list.draw()
```

With the additions in the program below, running your program should show the tiled map you created:



- pymunk_demo_platformer_04
- pymunk_demo_platformer_04_diff

9.6 Add Physics Engine

The next step is to add in the physics engine.

First, add some constants for our physics. Here we are setting:

- A constant for the force of gravity.
- Values for "damping". A damping of 1.0 will cause an item to lose all it's velocity once a force no longer applies to it. A damping of 0.5 causes 50% of speed to be lost in 1 second. A value of 0 is free-fall.
- Values for friction. 0.0 is ice, 1.0 is like rubber.
- Mass. Item default to 1. We make the player 2, so she can push items around easier.
- Limits are the players horizontal and vertical speed. It is easier to play if the player is limited to a constant speed. And more realistic, because they aren't on wheels.

Listing 6: Add Constants for Physics

```
# --- Physics forces. Higher number, faster accelerating.

# Gravity
GRAVITY = 1500

# Damping - Amount of speed lost per second
DEFAULT_DAMPING = 1.0
PLAYER_DAMPING = 0.4

# Friction between objects
PLAYER_FRICTION = 1.0
WALL_FRICTION = 0.7
DYNAMIC_ITEM_FRICTION = 0.6

# Mass (defaults to 1)
PLAYER_MASS = 2.0

# Keep player from going too fast
PLAYER_MAX_HORIZONTAL_SPEED = 450
PLAYER_MAX_VERTICAL_SPEED = 1600
```

Second, add the following attributer in the `__init__` method to hold our physics engine:

Listing 7: Add Physics Engine Attribute

```
        # Physics engine
        self.physics_engine = Optional[arcade.PymunkPhysicsEngine]
```

Third, in the `setup` method we create the physics engine and add the sprites. The player, walls, and dynamic items all have different properties so they are added individually.

Listing 8: Add Sprites to Physics Engine in 'setup' Method

```
        # Add to player sprite list
        self.player_list.append(self.player_sprite)

        # --- Pymunk Physics Engine Setup ---
```

(continues on next page)

(continued from previous page)

```

        # The default damping for every object controls the percent of velocity
        # the object will keep each second. A value of 1.0 is no speed loss,
        # 0.9 is 10% per second, 0.1 is 90% per second.
        # For top-down games, this is basically the friction for moving objects.
        # For platformers with gravity, this should probably be set to 1.0.
        # Default value is 1.0 if not specified.
        damping = DEFAULT_DAMPING

        # Set the gravity. (0, 0) is good for outer space and top-down.
        gravity = (0, -GRAVITY)

        # Create the physics engine
        self.physics_engine = arcade.PymunkPhysicsEngine(damping=damping,
                                                         gravity=gravity)

        # Add the player.
        # For the player, we set the damping to a lower value, which increases
        # the damping rate. This prevents the character from traveling too far
        # after the player lets off the movement keys.
        # Setting the moment of inertia to PymunkPhysicsEngine.MOMENT_INF prevents it␣
↪from
        # rotating.
        # Friction normally goes between 0 (no friction) and 1.0 (high friction)
        # Friction is between two objects in contact. It is important to remember
        # in top-down games that friction moving along the 'floor' is controlled
        # by damping.
        self.physics_engine.add_sprite(self.player_sprite,
                                       friction=PLAYER_FRICTION,
                                       mass=PLAYER_MASS,
                                       moment_of_inertia=arcade.PymunkPhysicsEngine.
↪MOMENT_INF,
                                       collision_type="player",
                                       max_horizontal_velocity=PLAYER_MAX_HORIZONTAL_
↪SPEED,
                                       max_vertical_velocity=PLAYER_MAX_VERTICAL_SPEED)

        # Create the walls.
        # By setting the body type to PymunkPhysicsEngine.STATIC the walls can't
        # move.
        # Movable objects that respond to forces are PymunkPhysicsEngine.DYNAMIC
        # PymunkPhysicsEngine.KINEMATIC objects will move, but are assumed to be
        # repositioned by code and don't respond to physics forces.
        # Dynamic is default.
        self.physics_engine.add_sprite_list(self.wall_list,
                                            friction=WALL_FRICTION,
                                            collision_type="wall",
                                            body_type=arcade.PymunkPhysicsEngine.STATIC)

        # Create the items
```

Fourth, in the `on_update` method we call the physics engine's `step` method.

Listing 9: Add Sprites to Physics Engine in 'setup' Method

```
    def on_update(self, delta_time):
        """ Movement and game logic """
        self.physics_engine.step()
```

If you run the program, and you have dynamic items that are up in the air, you should see them fall when the game starts.

- pymunk_demo_platformer_05
- pymunk_demo_platformer_05_diff

9.7 Add Player Movement

Next step is to get the player moving. In this section we'll cover how to move left and right. In the next section we'll show how to jump.

The force that we will move the player is defined as `PLAYER_MOVE_FORCE_ON_GROUND`. We'll apply a different force later, if the player happens to be airborne.

Listing 10: Add Player Movement - Constants and Attributes

```
# Force applied while on the ground
PLAYER_MOVE_FORCE_ON_GROUND = 8000

class GameWindow(arcade.Window):
    """ Main Window """

    def __init__(self, width, height, title):
        """ Create the variables """

        # Init the parent class
        super().__init__(width, height, title)

        # Player sprite
        self.player_sprite: Optional[arcade.Sprite] = None

        # Sprite lists we need
        self.player_list: Optional[arcade.SpriteList] = None
        self.wall_list: Optional[arcade.SpriteList] = None
        self.bullet_list: Optional[arcade.SpriteList] = None
        self.item_list: Optional[arcade.SpriteList] = None

        # Track the current state of what key is pressed
        self.left_pressed: bool = False
        self.right_pressed: bool = False
```

We need to track if the left/right keys are held down. To do this we define instance variables `left_pressed` and `right_pressed`. These are set to appropriate values in the key press and release handlers.

Listing 11: Handle Key Up and Down Events

```python
    def on_key_press(self, key, modifiers):
        """Called whenever a key is pressed. """

        if key == arcade.key.LEFT:
            self.left_pressed = True
        elif key == arcade.key.RIGHT:
            self.right_pressed = True

    def on_key_release(self, key, modifiers):
        """Called when the user releases a key. """

        if key == arcade.key.LEFT:
            self.left_pressed = False
        elif key == arcade.key.RIGHT:
            self.right_pressed = False
```

Finally, we need to apply the correct force in `on_update`. Force is specified in a tuple with horizontal force first, and vertical force second.

We also set the friction when we are moving to zero, and when we are not moving to 1. This is important to get realistic movement.

Listing 12: Apply Force to Move Player

```python
    def on_update(self, delta_time):
        """ Movement and game logic """

        # Update player forces based on keys pressed
        if self.left_pressed and not self.right_pressed:
            # Create a force to the left. Apply it.
            force = (-PLAYER_MOVE_FORCE_ON_GROUND, 0)
            self.physics_engine.apply_force(self.player_sprite, force)
            # Set friction to zero for the player while moving
            self.physics_engine.set_friction(self.player_sprite, 0)
        elif self.right_pressed and not self.left_pressed:
            # Create a force to the right. Apply it.
            force = (PLAYER_MOVE_FORCE_ON_GROUND, 0)
            self.physics_engine.apply_force(self.player_sprite, force)
            # Set friction to zero for the player while moving
            self.physics_engine.set_friction(self.player_sprite, 0)
        else:
            # Player's feet are not moving. Therefore up the friction so we stop.
            self.physics_engine.set_friction(self.player_sprite, 1.0)

        # Move items in the physics engine
        self.physics_engine.step()
```

- pymunk_demo_platformer_06
- pymunk_demo_platformer_06_diff

9.8 Add Player Jumping

To get the player to jump we need to:

- Make sure the player is on the ground.
- Apply an impulse force to the player upward.
- Change the left/right force to the player while they are in the air.

We can see if a sprite has a sprite below it with the `is_on_ground` function. Otherwise we'll be able to jump while we are in the air. (Double-jumps would allow this once.)

If we don't allow the player to move left-right while in the air, they player will be very hard to control. If we allow them to move left/right with the same force as on the ground, that's typically too much. So we've got a different left/right force depending if we are in the air or not.

For the code changes, first we'll define some constants:

Listing 13: Add Player Jumping - Constants

```python
# Force applied when moving left/right in the air
PLAYER_MOVE_FORCE_IN_AIR = 900

# Strength of a jump
PLAYER_JUMP_IMPULSE = 1800
```

We'll add logic that will apply the impulse force when we jump:

Listing 14: Add Player Jumping - Jump Force

```python
    def on_key_press(self, key, modifiers):
        """Called whenever a key is pressed. """

        if key == arcade.key.LEFT:
            self.left_pressed = True
        elif key == arcade.key.RIGHT:
            self.right_pressed = True
        elif key == arcade.key.UP:
            # find out if player is standing on ground
            if self.physics_engine.is_on_ground(self.player_sprite):
                # She is! Go ahead and jump
                impulse = (0, PLAYER_JUMP_IMPULSE)
                self.physics_engine.apply_impulse(self.player_sprite, impulse)
```

Then we will adjust the left/right force depending on if we are grounded or not:

Listing 15: Add Player Jumping - Left/Right Force Selection

```python
    def on_update(self, delta_time):
        """ Movement and game logic """

        is_on_ground = self.physics_engine.is_on_ground(self.player_sprite)
        # Update player forces based on keys pressed
        if self.left_pressed and not self.right_pressed:
            # Create a force to the left. Apply it.
            if is_on_ground:
```

(continues on next page)

(continued from previous page)

```
                force = (-PLAYER_MOVE_FORCE_ON_GROUND, 0)
            else:
                force = (-PLAYER_MOVE_FORCE_IN_AIR, 0)
            self.physics_engine.apply_force(self.player_sprite, force)
            # Set friction to zero for the player while moving
            self.physics_engine.set_friction(self.player_sprite, 0)
        elif self.right_pressed and not self.left_pressed:
            # Create a force to the right. Apply it.
            if is_on_ground:
                force = (PLAYER_MOVE_FORCE_ON_GROUND, 0)
            else:
                force = (PLAYER_MOVE_FORCE_IN_AIR, 0)
            self.physics_engine.apply_force(self.player_sprite, force)
            # Set friction to zero for the player while moving
            self.physics_engine.set_friction(self.player_sprite, 0)
        else:
            # Player's feet are not moving. Therefore up the friction so we stop.
            self.physics_engine.set_friction(self.player_sprite, 1.0)

```

- pymunk_demo_platformer_07
- pymunk_demo_platformer_07_diff

9.9 Add Player Animation

To create a player animation, we make a custom child class of `Sprite`. We load each frame of animation that we need, including a mirror image of it.

We will flip the player to face left or right. If the player is in the air, we'll also change between a jump up and a falling graphics.

Because the physics engine works with small floating point numbers, it often flips above and below zero by small amounts. It is a good idea *not* to change the animation as the x and y float around zero. For that reason, in this code we have a "dead zone." We don't change the animation until it gets outside of that zone.

We also need to control how far the player moves before we change the walking animation, so that the feet appear in-sync with the ground.

Listing 16: Add Player Animation - Constants

```
DEAD_ZONE = 0.1

# Constants used to track if the player is facing left or right
RIGHT_FACING = 0
LEFT_FACING = 1

# How many pixels to move before we change the texture in the walking animation
DISTANCE_TO_CHANGE_TEXTURE = 20

```

Next, we create a `Player` class that is a child to `arcade.Sprite`. This class will update the player animation.

The `__init__` method loads all of the textures. Here we use Kenney.nl's Toon Characters 1 pack. It has six different characters you can choose from with the same layout, so it makes changing as simple as changing which line is enabled. There are eight textures for walking, and textures for idle, jumping, and falling.

As the character can face left or right, we use `arcade.load_texture_pair` which will load both a regular image, and one that's mirrored.

For the multi-frame walking animation, we use an "odometer." We need to move a certain number of pixels before changing the animation. If this value is too small our character moves her legs like Fred Flintstone, too large and it looks like you are ice skating. We keep track of the index of our current texture, 0-7 since there are eight of them.

Any sprite moved by the Pymunk engine will have its `pymunk_moved` method called. This can be used to update the animation.

Listing 17: Add Player Animation - Player Class

```python
class PlayerSprite(arcade.Sprite):
    """ Player Sprite """
    def __init__(self):
        """ Init """
        # Let parent initialize
        super().__init__()

        # Set our scale
        self.scale = SPRITE_SCALING_PLAYER

        # Images from Kenney.nl's Character pack
        # main_path = ":resources:images/animated_characters/female_adventurer/
↪femaleAdventurer"
        main_path = ":resources:images/animated_characters/female_person/femalePerson"
        # main_path = ":resources:images/animated_characters/male_person/malePerson"
        # main_path = ":resources:images/animated_characters/male_adventurer/
↪maleAdventurer"
        # main_path = ":resources:images/animated_characters/zombie/zombie"
        # main_path = ":resources:images/animated_characters/robot/robot"

        # Load textures for idle standing
        self.idle_texture_pair = arcade.load_texture_pair(f"{main_path}_idle.png")
        self.jump_texture_pair = arcade.load_texture_pair(f"{main_path}_jump.png")
        self.fall_texture_pair = arcade.load_texture_pair(f"{main_path}_fall.png")

        # Load textures for walking
        self.walk_textures = []
        for i in range(8):
            texture = arcade.load_texture_pair(f"{main_path}_walk{i}.png")
            self.walk_textures.append(texture)

        # Set the initial texture
        self.texture = self.idle_texture_pair[0]

        # Default to face-right
        self.character_face_direction = RIGHT_FACING

        # Index of our current texture
        self.cur_texture = 0
```

(continues on next page)

(continued from previous page)

```python

        # How far have we traveled horizontally since changing the texture
        self.x_odometer = 0

    def pymunk_moved(self, physics_engine, dx, dy, d_angle):
        """ Handle being moved by the pymunk engine """
        # Figure out if we need to face left or right
        if dx < -DEAD_ZONE and self.character_face_direction == RIGHT_FACING:
            self.character_face_direction = LEFT_FACING
        elif dx > DEAD_ZONE and self.character_face_direction == LEFT_FACING:
            self.character_face_direction = RIGHT_FACING

        # Are we on the ground?
        is_on_ground = physics_engine.is_on_ground(self)

        # Add to the odometer how far we've moved
        self.x_odometer += dx

        # Jumping animation
        if not is_on_ground:
            if dy > DEAD_ZONE:
                self.texture = self.jump_texture_pair[self.character_face_direction]
                return
            elif dy < -DEAD_ZONE:
                self.texture = self.fall_texture_pair[self.character_face_direction]
                return

        # Idle animation
        if abs(dx) <= DEAD_ZONE:
            self.texture = self.idle_texture_pair[self.character_face_direction]
            return

        # Have we moved far enough to change the texture?
        if abs(self.x_odometer) > DISTANCE_TO_CHANGE_TEXTURE:

            # Reset the odometer
            self.x_odometer = 0

            # Advance the walking animation
            self.cur_texture += 1
            if self.cur_texture > 7:
                self.cur_texture = 0
            self.texture = self.walk_textures[self.cur_texture][self.character_face_
↪direction]
```

Important! At this point, we are still creating an instance of `arcade.Sprite` and **not** `PlayerSprite`. We need to go back to the `setup` method and replace the line that creates the `player` instance with:

Listing 18: Add Player Animation - Creating the Player Class

```
        # Create player sprite
        self.player_sprite = PlayerSprite()
```

A really common mistake I've seen programmers make (and made myself) is to forget that last part. Then you can spend a lot of time looking at the player class when the error is in the setup.

We also need to go back and change the data type for the player sprite attribute in our `__init__` method:

Listing 19: Add Player Animation - Creating the Player Class

```
        # Player sprite
        self.player_sprite: Optional[PlayerSprite] = None
```

- pymunk_demo_platformer_08
- pymunk_demo_platformer_08_diff

9.10 Shoot Bullets

Getting the player to shoot something can add a lot to our game. To begin with we'll define a few constants to use. How much force to shoot the bullet with, the bullet's mass, and the gravity to use for the bullet.

If we use the same gravity for the bullet as everything else, it tends to drop too fast. We could set this to zero if we wanted it to not drop at all.

Listing 20: Shoot Bullets - Constants

```
# How much force to put on the bullet
BULLET_MOVE_FORCE = 4500

# Mass of the bullet
BULLET_MASS = 0.1

# Make bullet less affected by gravity
BULLET_GRAVITY = 300
```

Next, we'll put in a mouse press handler to put in the bullet shooting code.

We need to:

- Create the bullet sprite
- We need to calculate the angle from the player to the mouse click
- Create the bullet away from the player in the proper direction, as spawning it inside the player will confuse the physics engine
- Add the bullet to the physics engine
- Apply the force to the bullet to make if move. Note that as we angled the bullet we don't need to angle the force.

Warning: Does your platformer scroll?

If your window scrolls, you need to add in the coordinate off-set or else the angle calculation will be incorrect.

> **Warning:** Bullets don't disappear yet!
>
> If the bullet flies off-screen, it doesn't go away and the physics engine still has to track it.

Listing 21: Shoot Bullets - Mouse Press

```python
    def on_mouse_press(self, x, y, button, modifiers):
        """ Called whenever the mouse button is clicked. """

        bullet = arcade.SpriteSolidColor(width=20, height=5, color=arcade.color.DARK_
↪YELLOW)
        self.bullet_list.append(bullet)

        # Position the bullet at the player's current location
        start_x = self.player_sprite.center_x
        start_y = self.player_sprite.center_y
        bullet.position = self.player_sprite.position

        # Get from the mouse the destination location for the bullet
        # IMPORTANT! If you have a scrolling screen, you will also need
        # to add in self.view_bottom and self.view_left.
        dest_x = x
        dest_y = y

        # Do math to calculate how to get the bullet to the destination.
        # Calculation the angle in radians between the start points
        # and end points. This is the angle the bullet will travel.
        x_diff = dest_x - start_x
        y_diff = dest_y - start_y
        angle = math.atan2(y_diff, x_diff)

        # What is the 1/2 size of this sprite, so we can figure out how far
        # away to spawn the bullet
        size = max(self.player_sprite.width, self.player_sprite.height) / 2

        # Use angle to to spawn bullet away from player in proper direction
        bullet.center_x += size * math.cos(angle)
        bullet.center_y += size * math.sin(angle)

        # Set angle of bullet
        bullet.angle = math.degrees(angle)

        # Gravity to use for the bullet
        # If we don't use custom gravity, bullet drops too fast, or we have
        # to make it go too fast.
        # Force is in relation to bullet's angle.
        bullet_gravity = (0, -BULLET_GRAVITY)

        # Add the sprite. This needs to be done AFTER setting the fields above.
        self.physics_engine.add_sprite(bullet,
                                       mass=BULLET_MASS,
                                       damping=1.0,
```

(continues on next page)

(continued from previous page)

```
                                          friction=0.6,
                                          collision_type="bullet",
                                          gravity=bullet_gravity,
                                          elasticity=0.9)

        # Add force to bullet
        force = (BULLET_MOVE_FORCE, 0)
        self.physics_engine.apply_force(bullet, force)
```

- pymunk_demo_platformer_09
- pymunk_demo_platformer_09_diff

9.11 Destroy Bullets and Items

This section has two goals:

- Get rid of the bullet if it flies off-screen
- Handle collisions of the bullet and other items

9.11.1 Destroy Bullet If It Goes Off-Screen

First, we'll create a custom bullet class. This class will define the `pymunk_moved` method, and check our location each time the bullet moves. If our y value is too low, we'll remove the bullet.

Listing 22: Destroy Bullets - Bullet Sprite

```
class BulletSprite(arcade.SpriteSolidColor):
    """ Bullet Sprite """
    def pymunk_moved(self, physics_engine, dx, dy, d_angle):
        """ Handle when the sprite is moved by the physics engine. """
        # If the bullet falls below the screen, remove it
        if self.center_y < -100:
            self.remove_from_sprite_lists()
```

And, of course, once we create the bullet we have to update our code to use it instead of the plain `arcade.Sprite` class.

Listing 23: Destroy Bullets - Bullet Sprite

```
        bullet = BulletSprite(width=20, height=5, color=arcade.color.DARK_YELLOW)
        self.bullet_list.append(bullet)

        # Position the bullet at the player's current location
        start_x = self.player_sprite.center_x
        start_y = self.player_sprite.center_y
```

9.11.2 Handle Collisions

To handle collisions, we can add custom collision handler call-backs. If you'll remember when we added items to the physics engine, we gave each item a collision type, such as "wall" or "bullet" or "item". We can write a function and register it to handle all bullet/wall collisions.

In this case, bullets that hit a wall go away. Bullets that hit items cause both the item and the bullet to go away. We could also add code to track damage to a sprite, only removing it after so much damage was applied. Even changing the texture depending on its health.

Listing 24: Destroy Bullets - Collision Handlers

```
        def wall_hit_handler(bullet_sprite, _wall_sprite, _arbiter, _space, _data):
            """ Called for bullet/wall collision """
            bullet_sprite.remove_from_sprite_lists()

        self.physics_engine.add_collision_handler("bullet", "wall", post_handler=wall_
↪hit_handler)

        def item_hit_handler(bullet_sprite, item_sprite, _arbiter, _space, _data):
            """ Called for bullet/wall collision """
            bullet_sprite.remove_from_sprite_lists()
            item_sprite.remove_from_sprite_lists()

        self.physics_engine.add_collision_handler("bullet", "item", post_handler=item_
↪hit_handler)
```

- pymunk_demo_platformer_10
- pymunk_demo_platformer_10_diff

9.12 Add Moving Platforms

We can add support for moving platforms. Platforms can be added in an object layer. An object layer allows platforms to be placed anywhere, and not just on exact grid locations. Object layers also allow us to add custom properties for each tile we place.

Once we have the tile placed, we can add custom properties for it. Click the '+' icon and add properties for all or some of:

- `change_x`
- `change_y`
- `left_boundary`

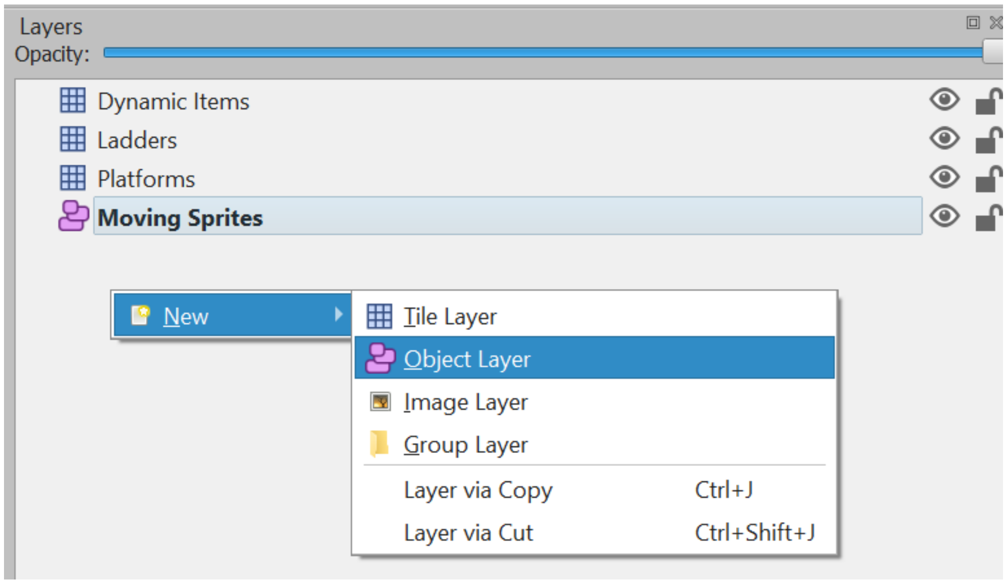


Fig. 1: Adding an object layer.

- `right_boundary`
- `top_boundary`
- `bottom_boundary`

If these are named exact matches, they'll automatically copy their values into the sprite attributes of the same name.

Now we need to update our code. In `GameWindow.__init__` add a line to create an attribute for `moving_sprites_list`:

Listing 25: Moving Platforms - Adding the sprite list

```
        self.moving_sprites_list: Optional[arcade.SpriteList] = None
```

In the `setup` method, load in the sprite list from the tmx layer.

Listing 26: Moving Platforms - Adding the sprite list

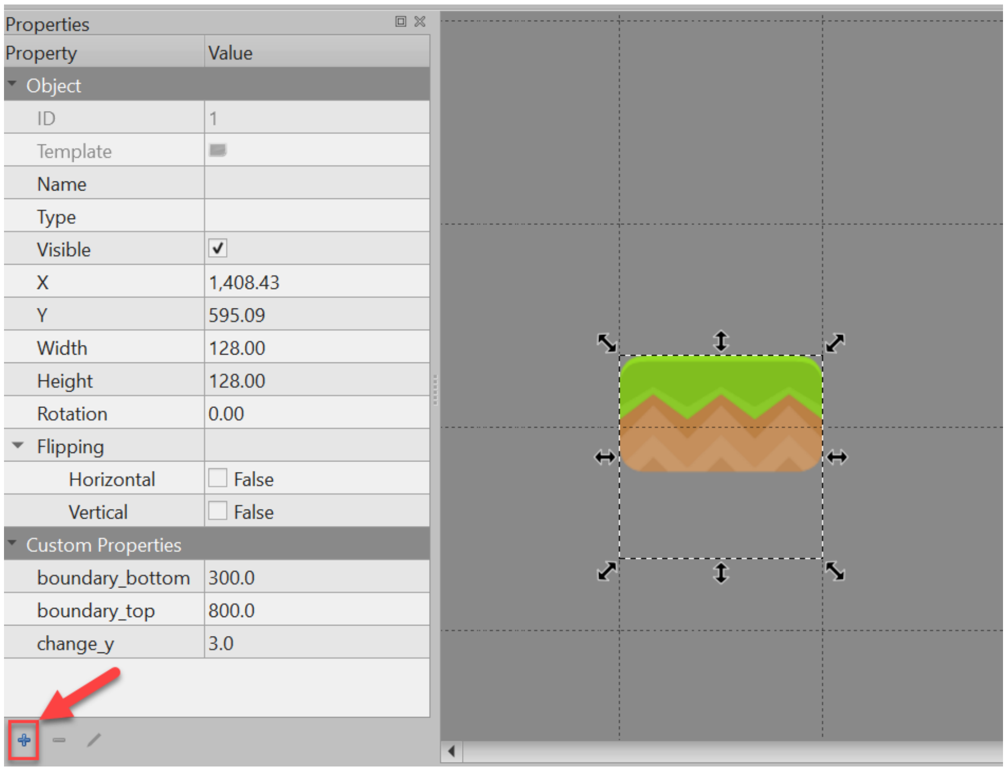
```
        self.moving_sprites_list = tile_map.sprite_lists['Moving Platforms']
```

Also in the `setup` method, we need to add these sprites to the physics engine. In this case we'll add the sprites as `KINEMATIC`. Static sprites don't move. Dynamic sprites move, and can have forces applied to them by other objects. Kinematic sprites do move, but aren't affected by other objects.

Listing 27: Moving Platforms - Loading the sprites

```
        # Add kinematic sprites
        self.physics_engine.add_sprite_list(self.moving_sprites_list,
                                            body_type=arcade.PymunkPhysicsEngine.
↪KINEMATIC)
```

We need to draw the moving platform sprites. After adding this line, you should be able to run the program and see the sprites from this layer, even if they don't move yet.



Properties

Property	Value
Object	
ID	1
Template	
Name	
Type	
Visible	✓
X	1,408.43
Y	595.09
Width	128.00
Height	128.00
Rotation	0.00
Flipping	
Horizontal	False
Vertical	False
Custom Properties	
boundary_bottom	300.0
boundary_top	800.0
change_y	3.0

Fig. 2: Adding custom properties.

Listing 28: Moving Platforms - Draw the sprites

```python
    def on_draw(self):
        """ Draw everything """
        self.clear()
        self.wall_list.draw()
        self.moving_sprites_list.draw()
        self.bullet_list.draw()
        self.item_list.draw()
        self.player_list.draw()
```

Next up, we need to get the sprites moving. First, we'll check to see if there are any boundaries set, and if we need to reverse our direction.

After that we'll create a velocity vector. Velocity is in pixels per second. In this case, I'm assuming the user set the velocity in pixels per frame in Tiled instead, so we'll convert.

Warning: Changing center_x and center_y will not move the sprite. If you want to change a sprite's position, use the physics engine's `set_position` method.

Also, setting an item's position "teleports" it there. The physics engine will happily move the object right into another object. Setting the item's velocity instead will cause the physics engine to move the item, pushing any dynamic items out of the way.

Listing 29: Moving Platforms - Moving the sprites

```
        # For each moving sprite, see if we've reached a boundary and need to
        # reverse course.
        for moving_sprite in self.moving_sprites_list:
            if moving_sprite.boundary_right and \
                    moving_sprite.change_x > 0 and \
                    moving_sprite.right > moving_sprite.boundary_right:
                moving_sprite.change_x *= -1
            elif moving_sprite.boundary_left and \
                    moving_sprite.change_x < 0 and \
                    moving_sprite.left > moving_sprite.boundary_left:
                moving_sprite.change_x *= -1
            if moving_sprite.boundary_top and \
                    moving_sprite.change_y > 0 and \
                    moving_sprite.top > moving_sprite.boundary_top:
                moving_sprite.change_y *= -1
            elif moving_sprite.boundary_bottom and \
                    moving_sprite.change_y < 0 and \
                    moving_sprite.bottom < moving_sprite.boundary_bottom:
                moving_sprite.change_y *= -1

            # Figure out and set our moving platform velocity.
            # Pymunk uses velocity is in pixels per second. If we instead have
            # pixels per frame, we need to convert.
            velocity = (moving_sprite.change_x * 1 / delta_time, moving_sprite.change_y␣
↪* 1 / delta_time)
            self.physics_engine.set_velocity(moving_sprite, velocity)
```

- pymunk_demo_platformer_11
- pymunk_demo_platformer_11_diff

9.13 Add Ladders

The first step to adding ladders to our platformer is modify the `__init__` to track some more items:

- Have a reference to a list of ladder sprites
- Add textures for a climbing animation
- Keep track of our movement in the y direction
- Add a boolean to track if we are on/off a ladder

Listing 30: Add Ladders - PlayerSprite class

```
    def __init__(self,
                 ladder_list: arcade.SpriteList,
                 hit_box_algorithm: arcade.hitbox.HitBoxAlgorithm):
        """ Init """
        # Let parent initialize
        super().__init__()

```

(continues on next page)

(continued from previous page)

```
        # Set our scale
        self.scale = SPRITE_SCALING_PLAYER

        # Images from Kenney.nl's Character pack
        # main_path = ":resources:images/animated_characters/female_adventurer/
↪femaleAdventurer"
        main_path = ":resources:images/animated_characters/female_person/femalePerson"
        # main_path = ":resources:images/animated_characters/male_person/malePerson"
        # main_path = ":resources:images/animated_characters/male_adventurer/
↪maleAdventurer"
        # main_path = ":resources:images/animated_characters/zombie/zombie"
        # main_path = ":resources:images/animated_characters/robot/robot"

        # Load textures for idle standing
        self.idle_texture_pair = arcade.load_texture_pair(f"{main_path}_idle.png",
                                                          hit_box_algorithm=hit_box_
↪algorithm)
        self.jump_texture_pair = arcade.load_texture_pair(f"{main_path}_jump.png")
        self.fall_texture_pair = arcade.load_texture_pair(f"{main_path}_fall.png")

        # Load textures for walking
        self.walk_textures = []
        for i in range(8):
            texture = arcade.load_texture_pair(f"{main_path}_walk{i}.png")
            self.walk_textures.append(texture)

        # Load textures for climbing
        self.climbing_textures = []
        texture = arcade.load_texture(f"{main_path}_climb0.png")
        self.climbing_textures.append(texture)
        texture = arcade.load_texture(f"{main_path}_climb1.png")
        self.climbing_textures.append(texture)

        # Set the initial texture
        self.texture = self.idle_texture_pair[0]

        # Default to face-right
        self.character_face_direction = RIGHT_FACING

        # Index of our current texture
        self.cur_texture = 0

        # How far have we traveled horizontally since changing the texture
        self.x_odometer = 0
        self.y_odometer = 0

        self.ladder_list = ladder_list
        self.is_on_ladder = False
```

Next, in our `pymunk_moved` method we need to change physics when we are on a ladder, and to update our player texture.

When we are on a ladder, we'll turn off gravity, turn up damping, and turn down our max vertical velocity. If we are off the ladder, reset those attributes.

When we are on a ladder, but not on the ground, we'll alternate between a couple climbing textures.

Listing 31: Add Ladders - PlayerSprite class

```python
    def pymunk_moved(self, physics_engine, dx, dy, d_angle):
        """ Handle being moved by the pymunk engine """
        # Figure out if we need to face left or right
        if dx < -DEAD_ZONE and self.character_face_direction == RIGHT_FACING:
            self.character_face_direction = LEFT_FACING
        elif dx > DEAD_ZONE and self.character_face_direction == LEFT_FACING:
            self.character_face_direction = RIGHT_FACING

        # Are we on the ground?
        is_on_ground = physics_engine.is_on_ground(self)

        # Are we on a ladder?
        if len(arcade.check_for_collision_with_list(self, self.ladder_list)) > 0:
            if not self.is_on_ladder:
                self.is_on_ladder = True
                self.pymunk.gravity = (0, 0)
                self.pymunk.damping = 0.0001
                self.pymunk.max_vertical_velocity = PLAYER_MAX_HORIZONTAL_SPEED
        else:
            if self.is_on_ladder:
                self.pymunk.damping = 1.0
                self.pymunk.max_vertical_velocity = PLAYER_MAX_VERTICAL_SPEED
                self.is_on_ladder = False
                self.pymunk.gravity = None

        # Add to the odometer how far we've moved
        self.x_odometer += dx
        self.y_odometer += dy

        if self.is_on_ladder and not is_on_ground:
            # Have we moved far enough to change the texture?
            if abs(self.y_odometer) > DISTANCE_TO_CHANGE_TEXTURE:

                # Reset the odometer
                self.y_odometer = 0

                # Advance the walking animation
                self.cur_texture += 1

            if self.cur_texture > 1:
                self.cur_texture = 0
            self.texture = self.climbing_textures[self.cur_texture]
            return

        # Jumping animation
        if not is_on_ground:
            if dy > DEAD_ZONE:
                self.texture = self.jump_texture_pair[self.character_face_direction]
```

(continues on next page)

(continued from previous page)

```
                return
            elif dy < -DEAD_ZONE:
                self.texture = self.fall_texture_pair[self.character_face_direction]
                return

        # Idle animation
        if abs(dx) <= DEAD_ZONE:
            self.texture = self.idle_texture_pair[self.character_face_direction]
            return

        # Have we moved far enough to change the texture?
        if abs(self.x_odometer) > DISTANCE_TO_CHANGE_TEXTURE:

            # Reset the odometer
            self.x_odometer = 0

            # Advance the walking animation
            self.cur_texture += 1
            if self.cur_texture > 7:
                self.cur_texture = 0
            self.texture = self.walk_textures[self.cur_texture][self.character_face_
↪direction]
```

Then we just need to add a few variables to the `__init__` to track ladders:

Listing 32: Add Ladders - Game Window Init

```
    def __init__(self, width, height, title):
        """ Create the variables """

        # Init the parent class
        super().__init__(width, height, title)

        # Player sprite
        self.player_sprite: Optional[PlayerSprite] = None

        # Sprite lists we need
        self.player_list: Optional[arcade.SpriteList] = None
        self.wall_list: Optional[arcade.SpriteList] = None
        self.bullet_list: Optional[arcade.SpriteList] = None
        self.item_list: Optional[arcade.SpriteList] = None
        self.moving_sprites_list: Optional[arcade.SpriteList] = None
        self.ladder_list: Optional[arcade.SpriteList] = None

        # Track the current state of what key is pressed
        self.left_pressed: bool = False
        self.right_pressed: bool = False
        self.up_pressed: bool = False
        self.down_pressed: bool = False

        # Physics engine
        self.physics_engine: Optional[arcade.PymunkPhysicsEngine] = None
```

(continues on next page)

(continued from previous page)

```python

        # Set background color
        self.background_color = arcade.color.AMAZON
```

Then load the ladder layer in `setup`:

Listing 33: Add Ladders - Game Window Setup

```python
        # Pull the sprite layers out of the tile map
        self.wall_list = tile_map.sprite_lists["Platforms"]
        self.item_list = tile_map.sprite_lists["Dynamic Items"]
        self.ladder_list = tile_map.sprite_lists["Ladders"]
        self.moving_sprites_list = tile_map.sprite_lists['Moving Platforms']
```

Also, pass the ladder list to the player class:

Listing 34: Add Ladders - Game Window Setup

```python
        # Create player sprite
        self.player_sprite = PlayerSprite(self.ladder_list, hit_box_algorithm=arcade.
↪hitbox.algo_detailed)
```

Then change the jump button so that we don't jump if we are on a ladder. Also, we want to track if the up key, or down key are pressed.

Listing 35: Add Ladders - Game Window Key Down

```python
    def on_key_press(self, key, modifiers):
        """Called whenever a key is pressed. """

        if key == arcade.key.LEFT:
            self.left_pressed = True
        elif key == arcade.key.RIGHT:
            self.right_pressed = True
        elif key == arcade.key.UP:
            self.up_pressed = True
            # find out if player is standing on ground, and not on a ladder
            if self.physics_engine.is_on_ground(self.player_sprite) \
                    and not self.player_sprite.is_on_ladder:
                # She is! Go ahead and jump
                impulse = (0, PLAYER_JUMP_IMPULSE)
                self.physics_engine.apply_impulse(self.player_sprite, impulse)
        elif key == arcade.key.DOWN:
            self.down_pressed = True
```

Add to the key up handler tracking for which key is pressed.

Listing 36: Add Ladders - Game Window Key Up

```python
    def on_key_release(self, key, modifiers):
        """Called when the user releases a key. """

        if key == arcade.key.LEFT:
```

(continues on next page)

(continued from previous page)

```
            self.left_pressed = False
        elif key == arcade.key.RIGHT:
            self.right_pressed = False
        elif key == arcade.key.UP:
            self.up_pressed = False
        elif key == arcade.key.DOWN:
            self.down_pressed = False
```

Next, change our update with new updates for the ladder.

Listing 37: Add Ladders - Game Window On Update

```
    def on_update(self, delta_time):
        """ Movement and game logic """

        is_on_ground = self.physics_engine.is_on_ground(self.player_sprite)
        # Update player forces based on keys pressed
        if self.left_pressed and not self.right_pressed:
            # Create a force to the left. Apply it.
            if is_on_ground or self.player_sprite.is_on_ladder:
                force = (-PLAYER_MOVE_FORCE_ON_GROUND, 0)
            else:
                force = (-PLAYER_MOVE_FORCE_IN_AIR, 0)
            self.physics_engine.apply_force(self.player_sprite, force)
            # Set friction to zero for the player while moving
            self.physics_engine.set_friction(self.player_sprite, 0)
        elif self.right_pressed and not self.left_pressed:
            # Create a force to the right. Apply it.
            if is_on_ground or self.player_sprite.is_on_ladder:
                force = (PLAYER_MOVE_FORCE_ON_GROUND, 0)
            else:
                force = (PLAYER_MOVE_FORCE_IN_AIR, 0)
            self.physics_engine.apply_force(self.player_sprite, force)
            # Set friction to zero for the player while moving
            self.physics_engine.set_friction(self.player_sprite, 0)
        elif self.up_pressed and not self.down_pressed:
            # Create a force to the right. Apply it.
            if self.player_sprite.is_on_ladder:
                force = (0, PLAYER_MOVE_FORCE_ON_GROUND)
                self.physics_engine.apply_force(self.player_sprite, force)
                # Set friction to zero for the player while moving
                self.physics_engine.set_friction(self.player_sprite, 0)
        elif self.down_pressed and not self.up_pressed:
            # Create a force to the right. Apply it.
            if self.player_sprite.is_on_ladder:
                force = (0, -PLAYER_MOVE_FORCE_ON_GROUND)
                self.physics_engine.apply_force(self.player_sprite, force)
                # Set friction to zero for the player while moving
                self.physics_engine.set_friction(self.player_sprite, 0)
```

And, of course, don't forget to draw the ladders:

Listing 38: Add Ladders - Game Window Key Down

```python
    def on_draw(self):
        """ Draw everything """
        self.clear()
        self.wall_list.draw()
        self.ladder_list.draw()
        self.moving_sprites_list.draw()
        self.bullet_list.draw()
        self.item_list.draw()
        self.player_list.draw()
```

- pymunk_demo_platformer_12
- pymunk_demo_platformer_12_diff

CHAPTER

TEN

USING VIEWS FOR START/END SCREENS

Views allow you to easily switch "views" for what you are showing on the window. You can use this to support adding screens such as:

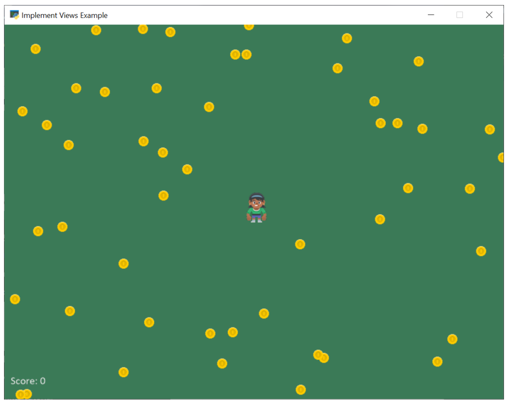
- Start screens
- Instruction screens
- Game over screens
- Pause screens

The `View` class is a lot like the `Window` class that you are already used to. The `View` class has methods for `on_update` and `on_draw` just like `Window`. We can change the current view to quickly change the code that is managing what is drawn on the window and handling user input.

If you know ahead of time you want to use views, you can build your code around the *View Management*. However, typically a programmer wants to add these items to a game that already exists.

This tutorial steps you through how to do just that.

10.1 Change Main Program to Use a View



First, we'll start with a simple collect coins example: 01_views

Then we'll move our game into a game view. Take the code where we define our window class:

```
class MyGame(arcade.Window):
```

Change it to derive from `arcade.View` instead of `arcade.Window`. I also suggest using "View" as part of the name:

```
class GameView(arcade.View):
```

This will require a couple other updates. The `View` class does not control the size of the window, so we'll need to take that out of the call to the parent class. Change:

```
super().__init__(SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_TITLE)
```

to:

```
super().__init__()
```

The `Window` class still controls if the mouse is visible or not, so to hide the mouse, we'll need to use the `window` attribute that is part of the `View` class. Change:

```
self.set_mouse_visible(False)
```

to:

```
self.window.set_mouse_visible(False)
```

Now in the `main` function, instead of just creating a window, we'll create a window, a view, and then show that view.

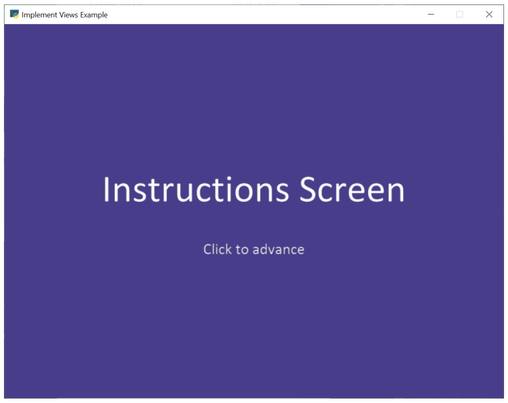
Listing 1: Add views - Main function

```
def main():
    """ Main function """

    window = arcade.Window(SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_TITLE)
    start_view = GameView()
    window.show_view(start_view)
    start_view.setup()
    arcade.run()
```

At this point, run your game and make sure that it still operates properly. It should run just like it did before, but now we are set up to add additional views.

- 02_views ← Full listing of where we are right now
- 02_views_diff ← What we changed to get here

10.2 Add Instruction Screen



Now we are ready to add in our instruction screen as a view. Create a class for it:

```python
class InstructionView(arcade.View):
```

Then we need to define the `on_show_view` method that will be run once when we switch to this view. In this case, we don't need to do much, just set the background color. If the game is one that scrolls, we'll also need to reset the viewport so that (0, 0) is back to the lower-left coordinate.

Listing 2: Add views - on_show_view

```python
    def on_show_view(self):
        """ This is run once when we switch to this view """
        self.window.background_color = arcade.csscolor.DARK_SLATE_BLUE

        # Reset the viewport, necessary if we have a scrolling game and we need
        # to reset the viewport back to the start so we can see what we draw.
        arcade.set_viewport(0, self.window.width, 0, self.window.height)
```

The `on_draw` method works just like the window class's method, but it will only be called when this view is active.

In this case, we'll just draw some text for the instruction screen. Another alternative is to make a graphic in a paint program, and show that image. We'll do that below where we show the Game Over screen.

Listing 3: Add views - on_draw

```python
    def on_draw(self):
        """ Draw this view """
        self.clear()
        arcade.draw_text("Instructions Screen", self.window.width / 2, self.window.
↪height / 2,
                         arcade.color.WHITE, font_size=50, anchor_x="center")
        arcade.draw_text("Click to advance", self.window.width / 2, self.window.height /␣
↪2-75,
                         arcade.color.WHITE, font_size=20, anchor_x="center")
```

Then we'll put in a method to respond to a mouse click. Here we'll create our `GameView` and call the setup method.

Listing 4: Add views - on_mouse_press

```
    def on_mouse_press(self, _x, _y, _button, _modifiers):
        """ If the user presses the mouse button, start the game. """
        game_view = GameView()
        game_view.setup()
        self.window.show_view(game_view)
```

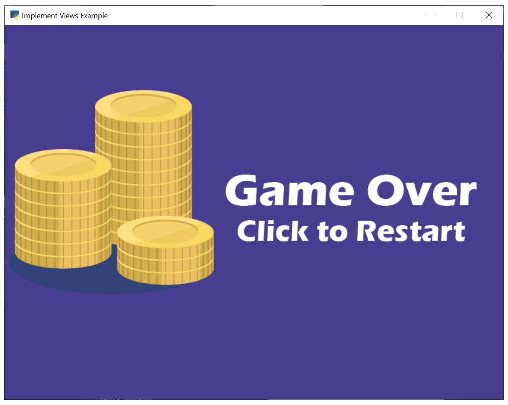
Now we need to go back to the main function. Instead of creating a GameView it needs to now create an InstructionView.

Listing 5: Add views - Main function

```
def main():
    """ Main function """

    window = arcade.Window(SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_TITLE)
    start_view = InstructionView()
    window.show_view(start_view)
    arcade.run()
```

- 03_views ← Full listing of where we are right now
- 03_views_diff ← What we changed to get here

10.3 Game Over Screen



Another way of doing instruction, pause, and game over screens is with a graphic. In this example, we've created a separate image with the same size as our window (800x600) and saved it as game_over.png. You can use the Windows "Paint" app or get an app for your Mac to make images in order to do this yourself.

The new GameOverView view that we are adding loads in the game over screen image as a texture in its __init__. The on_draw method draws that texture to the screen. By using an image, we can fancy up the game over screen using an image editor as much as we want, while keeping the code simple.

When the user clicks the mouse button, we just start the game over.

Listing 6: Add views - Game Over View

```python
class GameOverView(arcade.View):
    """ View to show when game is over """

    def __init__(self):
        """ This is run once when we switch to this view """
        super().__init__()
        self.texture = arcade.load_texture("game_over.png")

        # Reset the viewport, necessary if we have a scrolling game and we need
        # to reset the viewport back to the start so we can see what we draw.
        arcade.set_viewport(0, SCREEN_WIDTH - 1, 0, SCREEN_HEIGHT - 1)

    def on_draw(self):
        """ Draw this view """
        self.clear()
        self.texture.draw_sized(SCREEN_WIDTH / 2, SCREEN_HEIGHT / 2,
                                SCREEN_WIDTH, SCREEN_HEIGHT)

    def on_mouse_press(self, _x, _y, _button, _modifiers):
        """ If the user presses the mouse button, re-start the game. """
        game_view = GameView()
        game_view.setup()
        self.window.show_view(game_view)
```

The last thing we need, is to trigger the "Game Over" view. In our `GameView.on_update` method, we can check the list length. As soon as it hits zero, we'll change our view.

Listing 7: Add views - Game Over View

```python
    def on_update(self, delta_time):
        """ Movement and game logic """

        # Call update on all sprites (The sprites don't do much in this
        # example though.)
        self.coin_list.update()

        # Generate a list of all sprites that collided with the player.
        coins_hit_list = arcade.check_for_collision_with_list(self.player_sprite, self.
↪coin_list)

        # Loop through each colliding sprite, remove it, and add to the score.
        for coin in coins_hit_list:
            coin.remove_from_sprite_lists()
            self.score += 1

        # Check length of coin list. If it is zero, flip to the
        # game over view.
        if len(self.coin_list) == 0:
            view = GameOverView()
            self.window.show_view(view)
```

- 04_views ← Full listing of where we are right now

- 04_views_diff ← What we changed to get here

CHAPTER

ELEVEN

SOLITAIRE

This solitaire tutorial takes you though the basics of creating a card game, and doing extensive drag/drop work.

11.1 Open a Window

To begin with, let's start with a program that will use Arcade to open a blank window. The listing below also has stubs for methods we'll fill in later.

Get started with this code and make sure you can run it. It should pop open a green window.

Listing 1: Starting Program

```python
"""
Solitaire clone.
"""
import arcade

# Screen title and size
SCREEN_WIDTH = 1024
SCREEN_HEIGHT = 768
SCREEN_TITLE = "Drag and Drop Cards"


class MyGame(arcade.Window):
    """ Main application class. """

    def __init__(self):
        super().__init__(SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_TITLE)

        self.background_color = arcade.color.AMAZON

```

(continues on next page)

(continued from previous page)

```
    def setup(self):
        """ Set up the game here. Call this function to restart the game. """
        pass

    def on_draw(self):
        """ Render the screen. """
        # Clear the screen
        self.clear()

    def on_mouse_press(self, x, y, button, key_modifiers):
        """ Called when the user presses a mouse button. """
        pass

    def on_mouse_release(self, x: float, y: float, button: int,
                         modifiers: int):
        """ Called when the user presses a mouse button. """
        pass

    def on_mouse_motion(self, x: float, y: float, dx: float, dy: float):
        """ User moves mouse """
        pass


def main():
    """ Main function """
    window = MyGame()
    window.setup()
    arcade.run()


if __name__ == "__main__":
    main()
```

11.2 Create Card Sprites

Our next step is the create a bunch of sprites, one for each card.

11.2.1 Constants

First, we'll create some constants used in positioning the cards, and keeping track of what card is which.

We could just hard-code numbers, but I like to calculate things out. The "mat" will eventually be a square slightly larger than each card that tracks where we can put cards. (A mat where we can put a pile of cards on.)

Listing 2: Create constants for positioning

```
# Constants for sizing
CARD_SCALE = 0.6

# How big are the cards?
```

(continues on next page)

(continued from previous page)

```python
CARD_WIDTH = 140 * CARD_SCALE
CARD_HEIGHT = 190 * CARD_SCALE

# How big is the mat we'll place the card on?
MAT_PERCENT_OVERSIZE = 1.25
MAT_HEIGHT = int(CARD_HEIGHT * MAT_PERCENT_OVERSIZE)
MAT_WIDTH = int(CARD_WIDTH * MAT_PERCENT_OVERSIZE)

# How much space do we leave as a gap between the mats?
# Done as a percent of the mat size.
VERTICAL_MARGIN_PERCENT = 0.10
HORIZONTAL_MARGIN_PERCENT = 0.10

# The Y of the bottom row (2 piles)
BOTTOM_Y = MAT_HEIGHT / 2 + MAT_HEIGHT * VERTICAL_MARGIN_PERCENT

# The X of where to start putting things on the left side
START_X = MAT_WIDTH / 2 + MAT_WIDTH * HORIZONTAL_MARGIN_PERCENT

# Card constants
CARD_VALUES = ["A", "2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K"]
CARD_SUITS = ["Clubs", "Hearts", "Spades", "Diamonds"]
```

11.2.2 Card Class

Next up, we'll create a card class. The card class is a subclass of `arcade.Sprite`. It will have attributes for the suit and value of the card, and auto-load the image for the card based on that.

We'll use the entire image as the hit box, so we don't need to go through the time consuming hit box calculation. Therefore we turn that off. Otherwise loading the sprites would take a long time.

Listing 3: Create card sprites

```python
class Card(arcade.Sprite):
    """ Card sprite """

    def __init__(self, suit, value, scale=1):
        """ Card constructor """

        # Attributes for suit and value
        self.suit = suit
        self.value = value

        # Image to use for the sprite when face up
        self.image_file_name = f":resources:images/cards/card{self.suit}{self.value}.png"

        # Call the parent
        super().__init__(self.image_file_name, scale, hit_box_algorithm="None")
```

11.2.3 Creating Cards

We'll start by creating an attribute for the `SpriteList` that will hold all the cards in the game.

Listing 4: Create card sprites

```python
    def __init__(self):
        super().__init__(SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_TITLE)

        # Sprite list with all the cards, no matter what pile they are in.
        self.card_list = None

        self.background_color = arcade.color.AMAZON
```

In `setup` we'll create the list and the cards. We don't do this in `__init__` because by separating the creation into its own method, we can easily restart the game by calling `setup`.

Listing 5: Create card sprites

```python
    def setup(self):
        """ Set up the game here. Call this function to restart the game. """

        # Sprite list with all the cards, no matter what pile they are in.
        self.card_list = arcade.SpriteList()

        # Create every card
        for card_suit in CARD_SUITS:
            for card_value in CARD_VALUES:
                card = Card(card_suit, card_value, CARD_SCALE)
                card.position = START_X, BOTTOM_Y
                self.card_list.append(card)
```

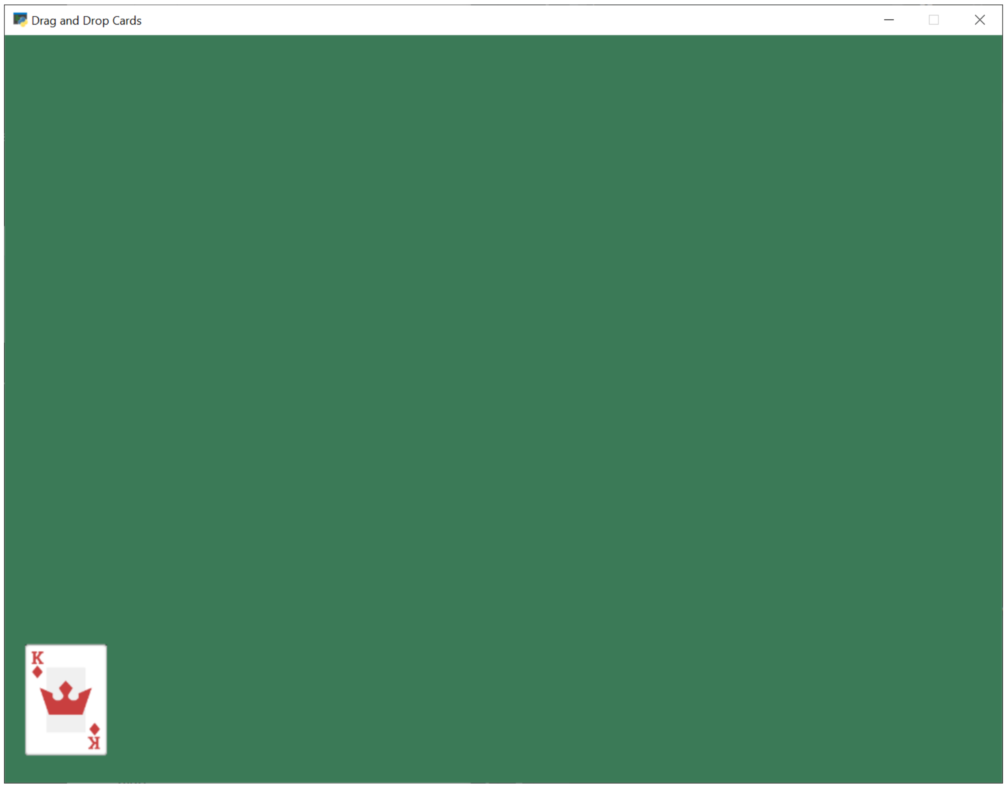
11.2.4 Drawing Cards

Finally, draw the cards:

Listing 6: Create card sprites

```python
    def on_draw(self):
        """ Render the screen. """
        # Clear the screen
        self.clear()

        # Draw the cards
        self.card_list.draw()
```

You should end up with all the cards stacked in the lower-left corner:



- solitaire_02 ← Full listing of where we are right now
- solitaire_02_diff ← What we changed to get here

11.3 Implement Drag and Drop

Next up, let's add the ability to pick up, drag, and drop the cards.

11.3.1 Track the Cards

First, let's add attributes to track what cards we are moving. Because we can move multiple cards, we'll keep this as a list. If the user drops the card in an illegal spot, we'll need to reset the card to its original position. So we'll also track that.

Create the attributes:

Listing 7: Add attributes to __init__

```
    def __init__(self):
        super().__init__(SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_TITLE)

        # Sprite list with all the cards, no matter what pile they are in.
        self.card_list = None

        self.background_color = arcade.color.AMAZON

        # List of cards we are dragging with the mouse
```

(continues on next page)

(continued from previous page)

```
        self.held_cards = None

        # Original location of cards we are dragging with the mouse in case
        # they have to go back.
        self.held_cards_original_position = None
```

Set the initial values (an empty list):

Listing 8: Create empty list attributes

```
    def setup(self):
        """ Set up the game here. Call this function to restart the game. """

        # List of cards we are dragging with the mouse
        self.held_cards = []

        # Original location of cards we are dragging with the mouse in case
        # they have to go back.
        self.held_cards_original_position = []

        # Sprite list with all the cards, no matter what pile they are in.
        self.card_list = arcade.SpriteList()

        # Create every card
        for card_suit in CARD_SUITS:
            for card_value in CARD_VALUES:
                card = Card(card_suit, card_value, CARD_SCALE)
                card.position = START_X, BOTTOM_Y
                self.card_list.append(card)
```

11.3.2 Pull Card to Top of Draw Order

When we click on the card, we'll want it to be the last card drawn, so it appears on top of all the other cards. Otherwise we might drag a card underneath another card, which would look odd.

Listing 9: Pull card to top

```
    def pull_to_top(self, card: arcade.Sprite):
        """ Pull card to top of rendering order (last to render, looks on-top) """

        # Remove, and append to the end
        self.card_list.remove(card)
        self.card_list.append(card)
```

11.3.3 Mouse Button Pressed

When the user presses the mouse button, we will:

- See if they clicked on a card
- If so, put that card in our held cards list
- Save the original position of the card
- Pull it to the top of the draw order

Listing 10: Pull card to top

```
    def on_mouse_press(self, x, y, button, key_modifiers):
        """ Called when the user presses a mouse button. """

        # Get list of cards we've clicked on
        cards = arcade.get_sprites_at_point((x, y), self.card_list)

        # Have we clicked on a card?
        if len(cards) > 0:

            # Might be a stack of cards, get the top one
            primary_card = cards[-1]

            # All other cases, grab the face-up card we are clicking on
            self.held_cards = [primary_card]
            # Save the position
            self.held_cards_original_position = [self.held_cards[0].position]
            # Put on top in drawing order
            self.pull_to_top(self.held_cards[0])
```

11.3.4 Mouse Moved

If the user moves the mouse, we'll move any held cards with it.

Listing 11: Pull card to top

```
    def on_mouse_motion(self, x: float, y: float, dx: float, dy: float):
        """ User moves mouse """

        # If we are holding cards, move them with the mouse
        for card in self.held_cards:
```

(continues on next page)

(continued from previous page)

```
            card.center_x += dx
            card.center_y += dy
```

11.3.5 Mouse Released

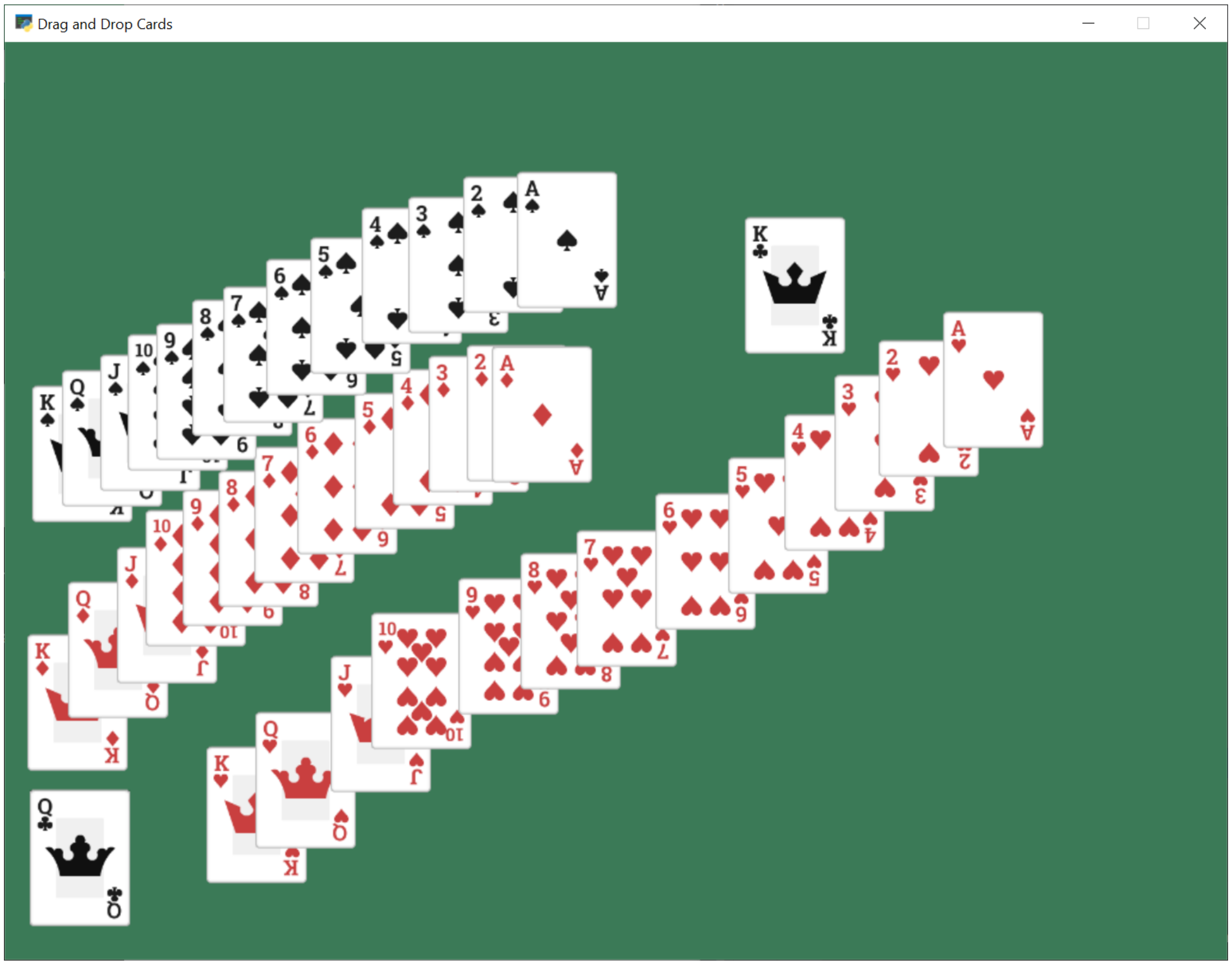
When the user releases the mouse button, we'll clear the held card list.

Listing 12: Pull card to top

```
    def on_mouse_release(self, x: float, y: float, button: int,
                         modifiers: int):
        """ Called when the user presses a mouse button. """

        # If we don't have any cards, who cares
        if len(self.held_cards) == 0:
            return

        # We are no longer holding cards
        self.held_cards = []
```

11.3.6 Test the Program

You should now be able to pick up and move cards around the screen. Try it out!



- solitaire_03 ← Full listing of where we are right now
- solitaire_03_diff ← What we changed to get here

11.4 Draw Pile Mats

Next, we'll create sprites that will act as guides to where the piles of cards go in our game. We'll create these as sprites, so we can use collision detection to figure out of we are dropping a card on them or not.

11.4.1 Create Constants

First, we'll create constants for the middle row of seven piles, and for the top row of four piles. We'll also create a constant for how far apart each pile should be.

Again, we could hard-code numbers, but I like calculating them so I can change the scale easily.

Listing 13: Add constants

```
# The Y of the top row (4 piles)
TOP_Y = SCREEN_HEIGHT - MAT_HEIGHT / 2 - MAT_HEIGHT * VERTICAL_MARGIN_PERCENT

```

(continues on next page)

(continued from previous page)

```python
# The Y of the middle row (7 piles)
MIDDLE_Y = TOP_Y - MAT_HEIGHT - MAT_HEIGHT * VERTICAL_MARGIN_PERCENT

# How far apart each pile goes
X_SPACING = MAT_WIDTH + MAT_WIDTH * HORIZONTAL_MARGIN_PERCENT
```

11.4.2 Create Mat Sprites

Create an attribute for the mat sprite list:

Listing 14: Create the mat sprites

```python
    def __init__(self):
        super().__init__(SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_TITLE)

        # Sprite list with all the cards, no matter what pile they are in.
        self.card_list = None

        self.background_color = arcade.color.AMAZON

        # List of cards we are dragging with the mouse
        self.held_cards = None

        # Original location of cards we are dragging with the mouse in case
        # they have to go back.
        self.held_cards_original_position = None

        # Sprite list with all the mats tha cards lay on.
        self.pile_mat_list = None
```

Then create the mat sprites in the `setup` method

Listing 15: Create the mat sprites

```python
    def setup(self):
        """ Set up the game here. Call this function to restart the game. """

        # List of cards we are dragging with the mouse
        self.held_cards = []

        # Original location of cards we are dragging with the mouse in case
        # they have to go back.
        self.held_cards_original_position = []

        # ---  Create the mats the cards go on.

        # Sprite list with all the mats tha cards lay on.
        self.pile_mat_list: arcade.SpriteList = arcade.SpriteList()

        # Create the mats for the bottom face down and face up piles
        pile = arcade.SpriteSolidColor(MAT_WIDTH, MAT_HEIGHT, arcade.csscolor.DARK_OLIVE_
```

(continues on next page)

(continued from previous page)

```
↪GREEN)
        pile.position = START_X, BOTTOM_Y
        self.pile_mat_list.append(pile)

        pile = arcade.SpriteSolidColor(MAT_WIDTH, MAT_HEIGHT, arcade.csscolor.DARK_OLIVE_
↪GREEN)
        pile.position = START_X + X_SPACING, BOTTOM_Y
        self.pile_mat_list.append(pile)

        # Create the seven middle piles
        for i in range(7):
            pile = arcade.SpriteSolidColor(MAT_WIDTH, MAT_HEIGHT, arcade.csscolor.DARK_
↪OLIVE_GREEN)
            pile.position = START_X + i * X_SPACING, MIDDLE_Y
            self.pile_mat_list.append(pile)

        # Create the top "play" piles
        for i in range(4):
            pile = arcade.SpriteSolidColor(MAT_WIDTH, MAT_HEIGHT, arcade.csscolor.DARK_
↪OLIVE_GREEN)
            pile.position = START_X + i * X_SPACING, TOP_Y
            self.pile_mat_list.append(pile)

        # Sprite list with all the cards, no matter what pile they are in.
        self.card_list = arcade.SpriteList()

        # Create every card
        for card_suit in CARD_SUITS:
            for card_value in CARD_VALUES:
                card = Card(card_suit, card_value, CARD_SCALE)
                card.position = START_X, BOTTOM_Y
                self.card_list.append(card)
```

11.4.3 Draw Mat Sprites

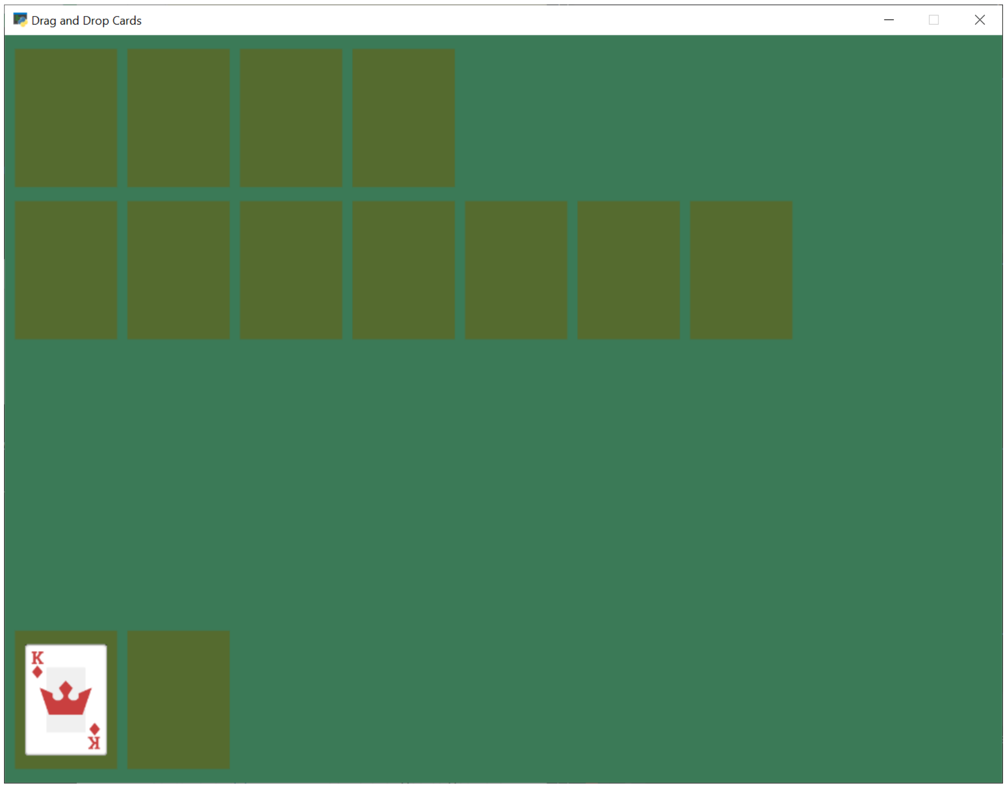
Finally, the mats aren't going to display if we don't draw them:

Listing 16: Draw the mat sprites

```python
    def on_draw(self):
        """ Render the screen. """
        # Clear the screen
        self.clear()

        # Draw the mats the cards go on to
        self.pile_mat_list.draw()

        # Draw the cards
        self.card_list.draw()
```

11.4.4 Test the Program

Run the program, and see if the mats appear:



- solitaire_04 ← Full listing of where we are right now
- solitaire_04_diff ← What we changed to get here

11.5 Snap Cards to Piles

Right now, you can drag the cards anywhere. They don't have to go onto a pile. Let's add code that "snaps" the card onto a pile. If we don't drop on a pile, let's reset back to the original location.

Listing 17: Snap to nearest pile

```python
    def on_mouse_release(self, x: float, y: float, button: int,
                         modifiers: int):
        """ Called when the user presses a mouse button. """

        # If we don't have any cards, who cares
        if len(self.held_cards) == 0:
            return

        # Find the closest pile, in case we are in contact with more than one
        pile, distance = arcade.get_closest_sprite(self.held_cards[0], self.pile_mat_
↪list)
        reset_position = True

        # See if we are in contact with the closest pile
        if arcade.check_for_collision(self.held_cards[0], pile):

            # For each held card, move it to the pile we dropped on
            for i, dropped_card in enumerate(self.held_cards):
                # Move cards to proper position
                dropped_card.position = pile.center_x, pile.center_y

            # Success, don't reset position of cards
            reset_position = False

            # Release on top play pile? And only one card held?
        if reset_position:
            # Where-ever we were dropped, it wasn't valid. Reset the each card's position
            # to its original spot.
            for pile_index, card in enumerate(self.held_cards):
                card.position = self.held_cards_original_position[pile_index]

        # We are no longer holding cards
        self.held_cards = []
```

- solitaire_05 ← Full listing of where we are right now
- solitaire_05_diff ← What we changed to get here

11.6 Shuffle the Cards

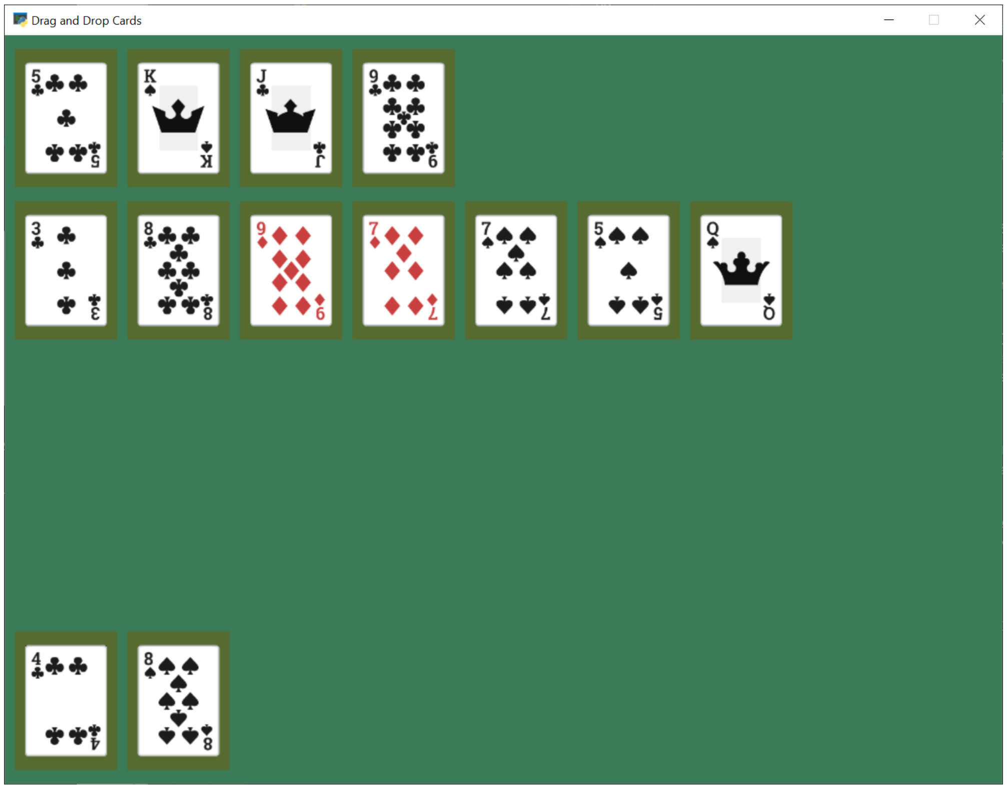
Having all the cards in order is boring. Let's shuffle them in the `setup` method:

Listing 18: Shuffle Cards

```python
        # Shuffle the cards
        for pos1 in range(len(self.card_list)):
            pos2 = random.randrange(len(self.card_list))
            self.card_list.swap(pos1, pos2)
```

Don't forget to `import random` at the top.

Run your program and make sure you can move cards around.



- solitaire_06 ← Full listing of where we are right now
- solitaire_06_diff ← What we changed to get here

11.7 Track Card Piles

Right now we are moving the cards around. But it isn't easy to figure out what card is in which pile. We could check by position, but then we start fanning the cards out, that will be very difficult.

Therefore we will keep a separate list for each pile of cards. When we move a card we need to move the position, and switch which list it is in.

11.7.1 Add New Constants

To start with, let's add some constants for each pile:

Listing 19: New Constants

```python
# If we fan out cards stacked on each other, how far apart to fan them?
CARD_VERTICAL_OFFSET = CARD_HEIGHT * CARD_SCALE * 0.3

# Constants that represent "what pile is what" for the game
PILE_COUNT = 13
BOTTOM_FACE_DOWN_PILE = 0
BOTTOM_FACE_UP_PILE = 1
PLAY_PILE_1 = 2
PLAY_PILE_2 = 3
PLAY_PILE_3 = 4
PLAY_PILE_4 = 5
PLAY_PILE_5 = 6
PLAY_PILE_6 = 7
PLAY_PILE_7 = 8
TOP_PILE_1 = 9
TOP_PILE_2 = 10
TOP_PILE_3 = 11
TOP_PILE_4 = 12
```

11.7.2 Create the Pile Lists

Then in our `__init__` add a variable to track the piles:

Listing 20: Init Method Additions

```python
        # Create a list of lists, each holds a pile of cards.
        self.piles = None
```

In the `setup` method, create a list for each pile. Then, add all the cards to the face-down deal pile. (Later, we'll add support for face-down cards. Yes, right now all the cards in the face down pile are up.)

Listing 21: Setup Method Additions

```
        # Create a list of lists, each holds a pile of cards.
        self.piles = [[] for _ in range(PILE_COUNT)]

        # Put all the cards in the bottom face-down pile
        for card in self.card_list:
            self.piles[BOTTOM_FACE_DOWN_PILE].append(card)
```

11.7.3 Card Pile Management Methods

Next, we need some convenience methods we'll use elsewhere.

First, given a card, return the index of which pile that card belongs to:

Listing 22: get_pile_for_card method

```
    def get_pile_for_card(self, card):
        """ What pile is this card in? """
        for index, pile in enumerate(self.piles):
            if card in pile:
                return index
```

Next, remove a card from whatever pile it happens to be in.

Listing 23: remove_card_from_pile method

```
    def remove_card_from_pile(self, card):
        """ Remove card from whatever pile it was in. """
        for pile in self.piles:
            if card in pile:
                pile.remove(card)
                break
```

Finally, move a card from one pile to another.

Listing 24: move_card_to_new_pile method

```
    def move_card_to_new_pile(self, card, pile_index):
        """ Move the card to a new pile """
        self.remove_card_from_pile(card)
        self.piles[pile_index].append(card)
```

11.7.4 Dropping the Card

Next, we need to modify what happens when we release the mouse.

First, see if we release it onto the same pile it came from. If so, just reset the card back to its original location.

Listing 25: on_mouse_release method

```
    def on_mouse_release(self, x: float, y: float, button: int,
                         modifiers: int):
        """ Called when the user presses a mouse button. """

        # If we don't have any cards, who cares
        if len(self.held_cards) == 0:
            return

        # Find the closest pile, in case we are in contact with more than one
        pile, distance = arcade.get_closest_sprite(self.held_cards[0], self.pile_mat_
↪list)
        reset_position = True

        # See if we are in contact with the closest pile
        if arcade.check_for_collision(self.held_cards[0], pile):

            # What pile is it?
            pile_index = self.pile_mat_list.index(pile)

            #  Is it the same pile we came from?
            if pile_index == self.get_pile_for_card(self.held_cards[0]):
                # If so, who cares. We'll just reset our position.
                pass
```

What if it is on a middle play pile? Ugh, that's a bit complicated. If the mat is empty, we need to place it in the middle of the mat. If there are cards on the mat, we need to offset the card so we can see a spread of cards.

While we can only pick up one card at a time right now, we need to support dropping multiple cards for once we support multiple card carries.

Listing 26: on_mouse_release method

```
            # Is it on a middle play pile?
            elif PLAY_PILE_1 <= pile_index <= PLAY_PILE_7:
                # Are there already cards there?
                if len(self.piles[pile_index]) > 0:
                    # Move cards to proper position
                    top_card = self.piles[pile_index][-1]
```

(continues on next page)

(continued from previous page)

```
                    for i, dropped_card in enumerate(self.held_cards):
                        dropped_card.position = top_card.center_x, \
                                                top_card.center_y - CARD_VERTICAL_OFFSET␣
↪* (i + 1)
                else:
                    # Are there no cards in the middle play pile?
                    for i, dropped_card in enumerate(self.held_cards):
                        # Move cards to proper position
                        dropped_card.position = pile.center_x, \
                                                pile.center_y - CARD_VERTICAL_OFFSET * i

                for card in self.held_cards:
                    # Cards are in the right position, but we need to move them to the␣
↪right list
                    self.move_card_to_new_pile(card, pile_index)

                # Success, don't reset position of cards
                reset_position = False
```

What if it is released on a top play pile? Make sure that we only have one card we are holding. We don't want to drop a stack up top. Then move the card to that pile.

Listing 27: on_mouse_release method

```
            # Release on top play pile? And only one card held?
            elif TOP_PILE_1 <= pile_index <= TOP_PILE_4 and len(self.held_cards) == 1:
                # Move position of card to pile
                self.held_cards[0].position = pile.position
                # Move card to card list
                for card in self.held_cards:
                    self.move_card_to_new_pile(card, pile_index)

                reset_position = False
```

If the move is invalid, we need to reset all held cards to their initial location.

Listing 28: on_mouse_release method

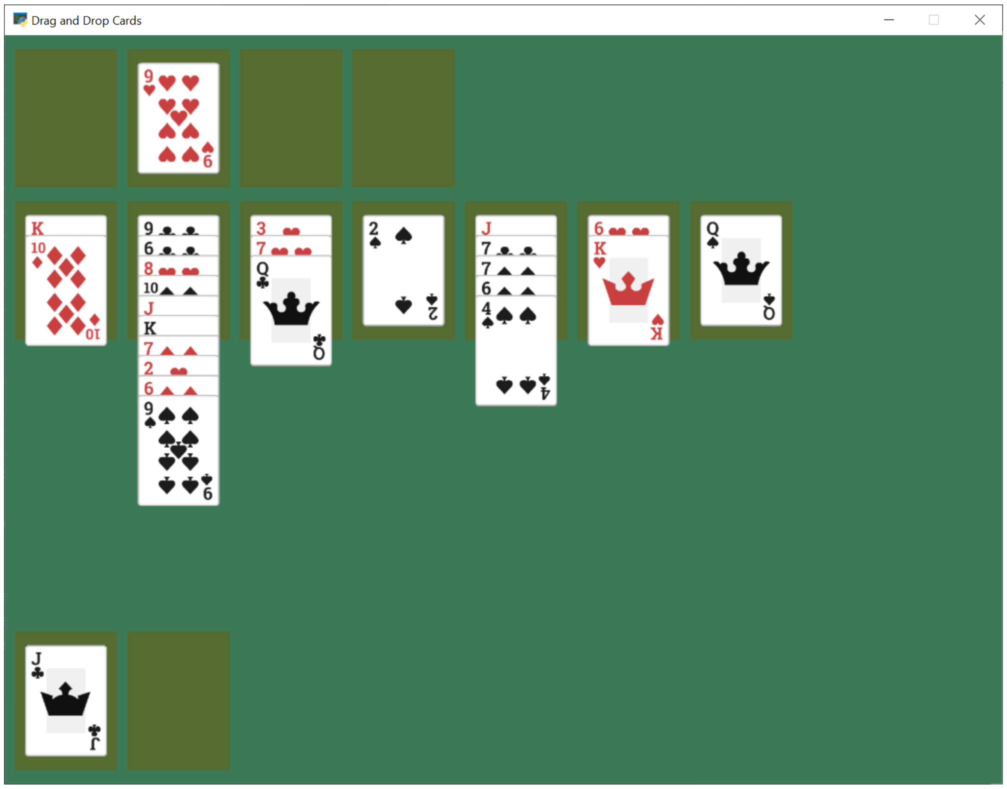
```
        if reset_position:
            # Where-ever we were dropped, it wasn't valid. Reset the each card's position
            # to its original spot.
            for pile_index, card in enumerate(self.held_cards):
                card.position = self.held_cards_original_position[pile_index]

        # We are no longer holding cards
        self.held_cards = []
```

11.7.5 Test

Test out your program, and see if the cards are being fanned out properly.

Note: The code isn't enforcing any game rules. You can stack cards in any order. Also, with long stacks of cards, you still have to drop the card on the mat. This is counter-intuitive when the stack of cards extends downwards past the mat.

We leave the solutions to these issues as an exercise for the reader.



- solitaire_07 ← Full listing of where we are right now
- solitaire_07_diff ← What we changed to get here

11.8 Pick Up Card Stacks

How do we pick up a whole stack of cards? When the mouse is pressed, we need to figure out what pile the card is in.

Next, look at where in the pile the card is that we clicked on. If there are any cards later on on the pile, we want to pick up those cards too. Add them to the list.

Listing 29: on_mouse_release method

```
    def on_mouse_press(self, x, y, button, key_modifiers):
        """ Called when the user presses a mouse button. """

        # Get list of cards we've clicked on
        cards = arcade.get_sprites_at_point((x, y), self.card_list)

        # Have we clicked on a card?
        if len(cards) > 0:

            # Might be a stack of cards, get the top one
            primary_card = cards[-1]
            # Figure out what pile the card is in
            pile_index = self.get_pile_for_card(primary_card)

            # All other cases, grab the face-up card we are clicking on
            self.held_cards = [primary_card]
            # Save the position
            self.held_cards_original_position = [self.held_cards[0].position]
            # Put on top in drawing order
            self.pull_to_top(self.held_cards[0])

            # Is this a stack of cards? If so, grab the other cards too
            card_index = self.piles[pile_index].index(primary_card)
            for i in range(card_index + 1, len(self.piles[pile_index])):
                card = self.piles[pile_index][i]
                self.held_cards.append(card)
                self.held_cards_original_position.append(card.position)
                self.pull_to_top(card)
```

After this, you should be able to pick up a stack of cards from the middle piles with the mouse and move them around.

- solitaire_08 ← Full listing of where we are right now
- solitaire_08_diff ← What we changed to get here

11.9 Deal Out Cards

We can deal the cards into the seven middle piles by adding some code to the `setup` method. We need to change the list each card is part of, along with its position.

Listing 30: Setup Method Additions

```
        # - Pull from that pile into the middle piles, all face-down
        # Loop for each pile
```

(continues on next page)

(continued from previous page)

```
        for pile_no in range(PLAY_PILE_1, PLAY_PILE_7 + 1):
            # Deal proper number of cards for that pile
            for j in range(pile_no - PLAY_PILE_1 + 1):
                # Pop the card off the deck we are dealing from
                card = self.piles[BOTTOM_FACE_DOWN_PILE].pop()
                # Put in the proper pile
                self.piles[pile_no].append(card)
                # Move card to same position as pile we just put it in
                card.position = self.pile_mat_list[pile_no].position
                # Put on top in draw order
                self.pull_to_top(card)
```

- solitaire_09 ← Full listing of where we are right now
- solitaire_09_diff ← What we changed to get here

11.10 Face Down Cards

We don't play solitaire with all the cards facing up, so let's add face-down support to our game.

11.10.1 New Constants

First define a constant for what image to use when face-down.

Listing 31: Face Down Image Constant

```
# Face down image
FACE_DOWN_IMAGE = ":resources:images/cards/cardBack_red2.png"
```

11.10.2 Updates to Card Class

Next, default each card in the `Card` class to be face up. Also, let's add methods to flip the card up or down.

Listing 32: Updated Card Class

```
class Card(arcade.Sprite):
    """ Card sprite """

    def __init__(self, suit, value, scale=1):
        """ Card constructor """

        # Attributes for suit and value
        self.suit = suit
        self.value = value

        # Image to use for the sprite when face up
        self.image_file_name = f":resources:images/cards/card{self.suit}{self.value}.png"
        self.is_face_up = False
        super().__init__(FACE_DOWN_IMAGE, scale, hit_box_algorithm="None")
```

(continues on next page)

(continued from previous page)

```python

    def face_down(self):
        """ Turn card face-down """
        self.texture = arcade.load_texture(FACE_DOWN_IMAGE)
        self.is_face_up = False

    def face_up(self):
        """ Turn card face-up """
        self.texture = arcade.load_texture(self.image_file_name)
        self.is_face_up = True

    @property
    def is_face_down(self):
        """ Is this card face down? """
        return not self.is_face_up
```

11.10.3 Flip Up Cards On Middle Seven Piles

Right now every card is face down. Let's update the `setup` method so the top cards in the middle seven piles are face up.

Listing 33: Flip Up Cards

```python
        # Flip up the top cards
        for i in range(PLAY_PILE_1, PLAY_PILE_7 + 1):
            self.piles[i][-1].face_up()
```

11.10.4 Flip Up Cards When Clicked

When we click on a card that is face down, instead of picking it up, let's flip it over:

Listing 34: Flip Up Cards

```python
    def on_mouse_press(self, x, y, button, key_modifiers):
        """ Called when the user presses a mouse button. """

        # Get list of cards we've clicked on
        cards = arcade.get_sprites_at_point((x, y), self.card_list)

        # Have we clicked on a card?
        if len(cards) > 0:

            # Might be a stack of cards, get the top one
            primary_card = cards[-1]
            assert isinstance(primary_card, Card)

            # Figure out what pile the card is in
            pile_index = self.get_pile_for_card(primary_card)

            if primary_card.is_face_down:
```

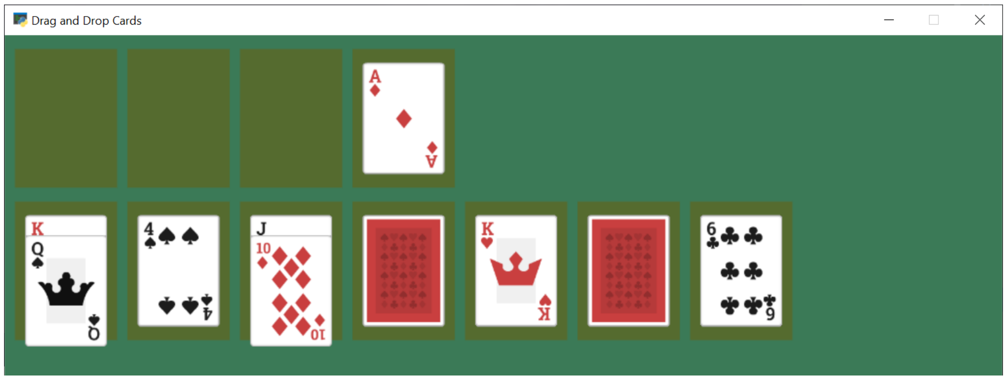
(continues on next page)

(continued from previous page)

```python
                # Is the card face down? In one of those middle 7 piles? Then flip up
                primary_card.face_up()
            else:
                # All other cases, grab the face-up card we are clicking on
                self.held_cards = [primary_card]
                # Save the position
                self.held_cards_original_position = [self.held_cards[0].position]
                # Put on top in drawing order
                self.pull_to_top(self.held_cards[0])

                # Is this a stack of cards? If so, grab the other cards too
                card_index = self.piles[pile_index].index(primary_card)
                for i in range(card_index + 1, len(self.piles[pile_index])):
                    card = self.piles[pile_index][i]
                    self.held_cards.append(card)
                    self.held_cards_original_position.append(card.position)
                    self.pull_to_top(card)
```

11.10.5 Test

Try out your program. As you move cards around, you should see face down cards as well, and be able to flip them over.



- solitaire_10 ← Full listing of where we are right now
- solitaire_10_diff ← What we changed to get here

11.11 Restart Game

We can add the ability to restart are game any type we press the 'R' key:

Listing 35: Flip Up Cards

```python
    def on_key_press(self, symbol: int, modifiers: int):
        """ User presses key """
        if symbol == arcade.key.R:
            # Restart
            self.setup()
```

11.12 Flip Three From Draw Pile

The draw pile at the bottom of our screen doesn't work right yet. When we click on it, we need it to flip three cards to the bottom-right pile. Also, if the have gone through all the cards in the pile, we need to reset the pile so we can go through it again.

Listing 36: Flipping of Bottom Deck

```python
    def on_mouse_press(self, x, y, button, key_modifiers):
        """ Called when the user presses a mouse button. """

        # Get list of cards we've clicked on
        cards = arcade.get_sprites_at_point((x, y), self.card_list)

        # Have we clicked on a card?
        if len(cards) > 0:

            # Might be a stack of cards, get the top one
            primary_card = cards[-1]
            assert isinstance(primary_card, Card)

            # Figure out what pile the card is in
            pile_index = self.get_pile_for_card(primary_card)

            # Are we clicking on the bottom deck, to flip three cards?
            if pile_index == BOTTOM_FACE_DOWN_PILE:
                # Flip three cards
                for i in range(3):
                    # If we ran out of cards, stop
                    if len(self.piles[BOTTOM_FACE_DOWN_PILE]) == 0:
                        break
                    # Get top card
                    card = self.piles[BOTTOM_FACE_DOWN_PILE][-1]
                    # Flip face up
                    card.face_up()
                    # Move card position to bottom-right face up pile
                    card.position = self.pile_mat_list[BOTTOM_FACE_UP_PILE].position
                    # Remove card from face down pile
                    self.piles[BOTTOM_FACE_DOWN_PILE].remove(card)
                    # Move card to face up list
                    self.piles[BOTTOM_FACE_UP_PILE].append(card)
                    # Put on top draw-order wise
                    self.pull_to_top(card)

            elif primary_card.is_face_down:
                # Is the card face down? In one of those middle 7 piles? Then flip up
                primary_card.face_up()
            else:
                # All other cases, grab the face-up card we are clicking on
                self.held_cards = [primary_card]
                # Save the position
                self.held_cards_original_position = [self.held_cards[0].position]
                # Put on top in drawing order
```

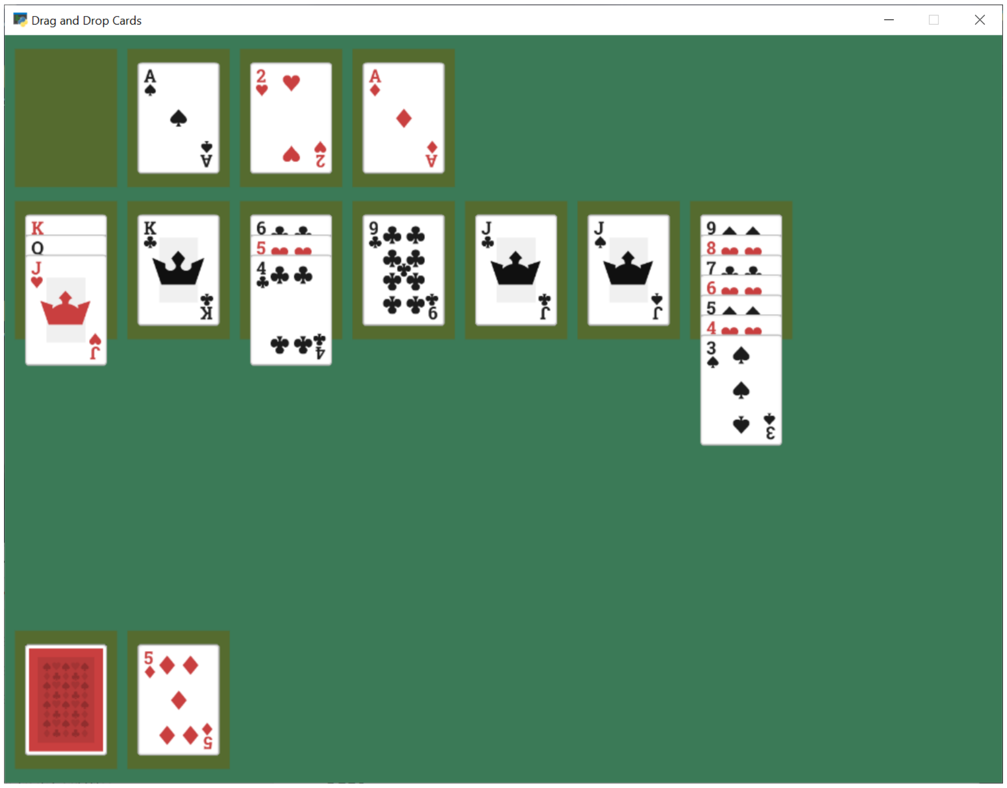
(continues on next page)

(continued from previous page)

```
                self.pull_to_top(self.held_cards[0])

                # Is this a stack of cards? If so, grab the other cards too
                card_index = self.piles[pile_index].index(primary_card)
                for i in range(card_index + 1, len(self.piles[pile_index])):
                    card = self.piles[pile_index][i]
                    self.held_cards.append(card)
                    self.held_cards_original_position.append(card.position)
                    self.pull_to_top(card)

        else:

            # Click on a mat instead of a card?
            mats = arcade.get_sprites_at_point((x, y), self.pile_mat_list)

            if len(mats) > 0:
                mat = mats[0]
                mat_index = self.pile_mat_list.index(mat)

                # Is it our turned over flip mat? and no cards on it?
                if mat_index == BOTTOM_FACE_DOWN_PILE and len(self.piles[BOTTOM_FACE_
↪DOWN_PILE]) == 0:
                    # Flip the deck back over so we can restart
                    temp_list = self.piles[BOTTOM_FACE_UP_PILE].copy()
                    for card in reversed(temp_list):
                        card.face_down()
                        self.piles[BOTTOM_FACE_UP_PILE].remove(card)
                        self.piles[BOTTOM_FACE_DOWN_PILE].append(card)
                        card.position = self.pile_mat_list[BOTTOM_FACE_DOWN_PILE].
↪position
```

11.12.1 Test

Now we've got a basic working solitaire game! Try it out!



- solitaire_11 ← Full listing of where we are right now
- solitaire_11_diff ← What we changed to get here

11.13 Conclusion

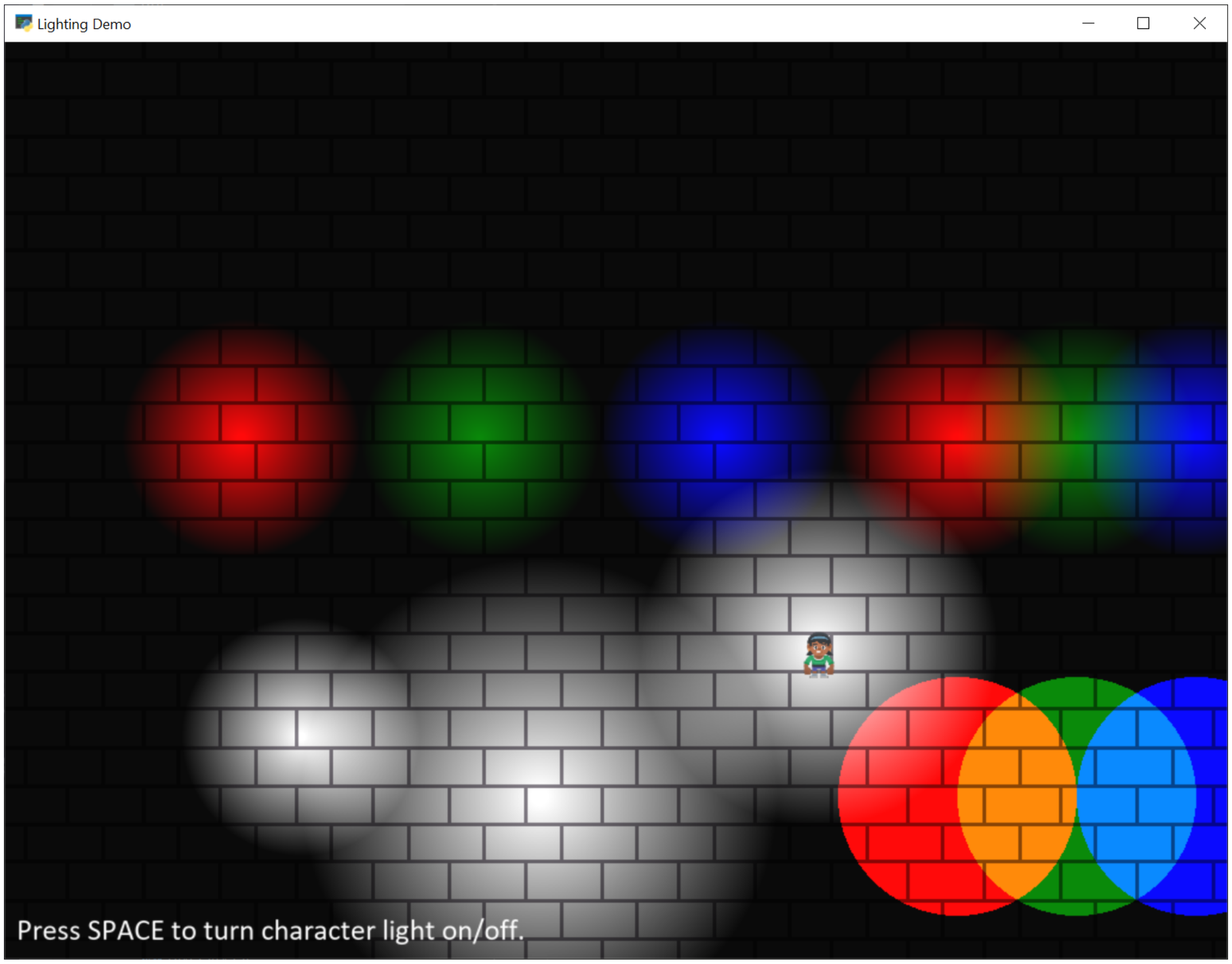
There's a lot more that could be added to this game, such as enforcing rules, adding animation to 'slide' a dropped card to its position, sound, better graphics, and more. Or this could be adapted to a different card game.

Hopefully this is enough to get you started on your own game.

CHAPTER

TWELVE

LIGHTS



Press SPACE to turn character light on/off.

This tutorial needs some documentation. Feel free to submit a PR to improve it!

Listing 1: light_demo.py

```
"""
Show how to use lights.

.. note:: This uses features from the upcoming version 2.4. The API for these
          functions may still change. To use, you will need to install one of the
```

(continues on next page)

(continued from previous page)

```python
          pre-release packages, or install via GitHub.

Artwork from http://kenney.nl

"""
import arcade
from arcade.experimental.lights import Light, LightLayer

SCREEN_WIDTH = 1024
SCREEN_HEIGHT = 768
SCREEN_TITLE = "Lighting Demo"
VIEWPORT_MARGIN = 200
MOVEMENT_SPEED = 5

# This is the color used for 'ambient light'. If you don't want any
# ambient light, set it to black.
AMBIENT_COLOR = (10, 10, 10)

class MyGame(arcade.Window):
    """ Main Game Window """

    def __init__(self, width, height, title):
        """ Set up the class. """
        super().__init__(width, height, title, resizable=True)

        # Sprite lists
        self.background_sprite_list = None
        self.player_list = None
        self.wall_list = None
        self.player_sprite = None

        # Physics engine
        self.physics_engine = None

        # Used for scrolling
        self.view_left = 0
        self.view_bottom = 0

        # --- Light related ---
        # List of all the lights
        self.light_layer = None
        # Individual light we move with player, and turn on/off
        self.player_light = None

    def setup(self):
        """ Create everything """

        # Create sprite lists
        self.background_sprite_list = arcade.SpriteList()
        self.player_list = arcade.SpriteList()
        self.wall_list = arcade.SpriteList()

```

(continues on next page)

(continued from previous page)

```
        # Create player sprite
        self.player_sprite = arcade.Sprite(":resources:images/animated_characters/female_
↪person/femalePerson_idle.png", 0.4)
        self.player_sprite.center_x = 64
        self.player_sprite.center_y = 270
        self.player_list.append(self.player_sprite)

        # --- Light related ---
        # Lights must shine on something. If there is no background sprite or color,
        # you will just see black. Therefore, we use a loop to create a whole bunch of␣
↪brick tiles to go in the
        # background.
        for x in range(-128, 2000, 128):
            for y in range(-128, 1000, 128):
                sprite = arcade.Sprite(":resources:images/tiles/brickTextureWhite.png")
                sprite.position = x, y
                self.background_sprite_list.append(sprite)

        # Create a light layer, used to render things to, then post-process and
        # add lights. This must match the screen size.
        self.light_layer = LightLayer(SCREEN_WIDTH, SCREEN_HEIGHT)
        # We can also set the background color that will be lit by lights,
        # but in this instance we just want a black background
        self.light_layer.set_background_color(arcade.color.BLACK)

        # Here we create a bunch of lights.

        # Create a small white light
        x = 100
        y = 200
        radius = 100
        mode = 'soft'
        color = arcade.csscolor.WHITE
        light = Light(x, y, radius, color, mode)
        self.light_layer.add(light)

        # Create an overlapping, large white light
        x = 300
        y = 150
        radius = 200
        color = arcade.csscolor.WHITE
        mode = 'soft'
        light = Light(x, y, radius, color, mode)
        self.light_layer.add(light)

        # Create three, non-overlapping RGB lights
        x = 50
        y = 450
        radius = 100
        mode = 'soft'
        color = arcade.csscolor.RED
        light = Light(x, y, radius, color, mode)
```

(continues on next page)

(continued from previous page)

```
        self.light_layer.add(light)

        x = 250
        y = 450
        radius = 100
        mode = 'soft'
        color = arcade.csscolor.GREEN
        light = Light(x, y, radius, color, mode)
        self.light_layer.add(light)

        x = 450
        y = 450
        radius = 100
        mode = 'soft'
        color = arcade.csscolor.BLUE
        light = Light(x, y, radius, color, mode)
        self.light_layer.add(light)

        # Create three, overlapping RGB lights
        x = 650
        y = 450
        radius = 100
        mode = 'soft'
        color = arcade.csscolor.RED
        light = Light(x, y, radius, color, mode)
        self.light_layer.add(light)

        x = 750
        y = 450
        radius = 100
        mode = 'soft'
        color = arcade.csscolor.GREEN
        light = Light(x, y, radius, color, mode)
        self.light_layer.add(light)

        x = 850
        y = 450
        radius = 100
        mode = 'soft'
        color = arcade.csscolor.BLUE
        light = Light(x, y, radius, color, mode)
        self.light_layer.add(light)

        # Create three, overlapping RGB lights
        # But 'hard' lights that don't fade out.
        x = 650
        y = 150
        radius = 100
        mode = 'hard'
        color = arcade.csscolor.RED
        light = Light(x, y, radius, color, mode)
        self.light_layer.add(light)
```

(continues on next page)

(continued from previous page)

```python

        x = 750
        y = 150
        radius = 100
        mode = 'hard'
        color = arcade.csscolor.GREEN
        light = Light(x, y, radius, color, mode)
        self.light_layer.add(light)

        x = 850
        y = 150
        radius = 100
        mode = 'hard'
        color = arcade.csscolor.BLUE
        light = Light(x, y, radius, color, mode)
        self.light_layer.add(light)

        # Create a light to follow the player around.
        # We'll position it later, when the player moves.
        # We'll only add it to the light layer when the player turns the light
        # on. We start with the light off.
        radius = 150
        mode = 'soft'
        color = arcade.csscolor.WHITE
        self.player_light = Light(0, 0, radius, color, mode)

        # Create the physics engine
        self.physics_engine = arcade.PhysicsEngineSimple(self.player_sprite, self.wall_
↪list)

        # Set the viewport boundaries
        # These numbers set where we have 'scrolled' to.
        self.view_left = 0
        self.view_bottom = 0

    def on_draw(self):
        """ Draw everything. """
        self.clear()

        # --- Light related ---
        # Everything that should be affected by lights gets rendered inside this
        # 'with' statement. Nothing is rendered to the screen yet, just the light
        # layer.
        with self.light_layer:
            self.background_sprite_list.draw()
            self.player_list.draw()

        # Draw the light layer to the screen.
        # This fills the entire screen with the lit version
        # of what we drew into the light layer above.
        self.light_layer.draw(ambient_color=AMBIENT_COLOR)

```

(continues on next page)

(continued from previous page)

```
        # Now draw anything that should NOT be affected by lighting.
        arcade.draw_text("Press SPACE to turn character light on/off.",
                         10 + self.view_left, 10 + self.view_bottom,
                         arcade.color.WHITE, 20)

    def on_resize(self, width, height):
        """ User resizes the screen. """

        # --- Light related ---
        # We need to resize the light layer to
        self.light_layer.resize(width, height)

        # Scroll the screen so the user is visible
        self.scroll_screen()

    def on_key_press(self, key, _):
        """Called whenever a key is pressed. """

        if key == arcade.key.UP:
            self.player_sprite.change_y = MOVEMENT_SPEED
        elif key == arcade.key.DOWN:
            self.player_sprite.change_y = -MOVEMENT_SPEED
        elif key == arcade.key.LEFT:
            self.player_sprite.change_x = -MOVEMENT_SPEED
        elif key == arcade.key.RIGHT:
            self.player_sprite.change_x = MOVEMENT_SPEED
        elif key == arcade.key.SPACE:
            # --- Light related ---
            # We can add/remove lights from the light layer. If they aren't
            # in the light layer, the light is off.
            if self.player_light in self.light_layer:
                self.light_layer.remove(self.player_light)
            else:
                self.light_layer.add(self.player_light)

    def on_key_release(self, key, _):
        """Called when the user releases a key. """

        if key == arcade.key.UP or key == arcade.key.DOWN:
            self.player_sprite.change_y = 0
        elif key == arcade.key.LEFT or key == arcade.key.RIGHT:
            self.player_sprite.change_x = 0

    def scroll_screen(self):
        """ Manage Scrolling """

        # Scroll left
        left_boundary = self.view_left + VIEWPORT_MARGIN
        if self.player_sprite.left < left_boundary:
            self.view_left -= left_boundary - self.player_sprite.left

        # Scroll right
```

(continues on next page)

(continued from previous page)

```python
        right_boundary = self.view_left + self.width - VIEWPORT_MARGIN
        if self.player_sprite.right > right_boundary:
            self.view_left += self.player_sprite.right - right_boundary

        # Scroll up
        top_boundary = self.view_bottom + self.height - VIEWPORT_MARGIN
        if self.player_sprite.top > top_boundary:
            self.view_bottom += self.player_sprite.top - top_boundary

        # Scroll down
        bottom_boundary = self.view_bottom + VIEWPORT_MARGIN
        if self.player_sprite.bottom < bottom_boundary:
            self.view_bottom -= bottom_boundary - self.player_sprite.bottom

        # Make sure our boundaries are integer values. While the viewport does
        # support floating point numbers, for this application we want every pixel
        # in the view port to map directly onto a pixel on the screen. We don't want
        # any rounding errors.
        self.view_left = int(self.view_left)
        self.view_bottom = int(self.view_bottom)

        arcade.set_viewport(self.view_left,
                            self.width + self.view_left,
                            self.view_bottom,
                            self.height + self.view_bottom)

    def on_update(self, delta_time):
        """ Movement and game logic """

        # Call update on all sprites (The sprites don't do much in this
        # example though.)
        self.physics_engine.update()

        # --- Light related ---
        # We can easily move the light by setting the position,
        # or by center_x, center_y.
        self.player_light.position = self.player_sprite.position

        # Scroll the screen so we can see the player
        self.scroll_screen()


if __name__ == "__main__":
    window = MyGame(SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_TITLE)
    window.setup()
    arcade.run()
```

CHAPTER

THIRTEEN

BUNDLING A GAME WITH PYINSTALLER

You've written your game using Arcade and it is a masterpiece! Congrats! Now you want to share it with others. That usually means helping people install Python, downloading the necessary modules, copying your code, and then getting it all working. Sharing is not an easy task. Well, PyInstaller can change all that!

PyInstaller is a tool for Python that lets you bundle up an entire Python application into a one-file executable bundle that you can easily share. Thankfully, it works great with Arcade!

We will be demonstrating usage with Windows, but everything should work exactly the same across Windows, Mac, and Linux. Note that you can only build for the system you are on. This means that in order to make a Windows build, you must be on a Windows machine, same thing for Linux and Mac.

13.1 Bundling a Simple Arcade Script

To demonstrate how PyInstaller works, we will:

- Install PyInstaller
- Create a simple example application that uses Arcade
- Bundle the application into a one-file executable
- Run the application

First, make sure both Arcade and PyInstaller are installed in your Python environment with:

```
pip install arcade pyinstaller
```

Then we need our game. In this case, we'll start simple. We need a one-file game that doesn't require any additional images or sounds. Once we have that working, we can get more complicated. Create a file called `main.py` that contains the following:

Listing 1: Sample game – main.py

```python
import arcade

arcade.open_window(400, 400, "My Game")

self.clear()
arcade.draw_circle_filled(200, 200, 100, arcade.color.BLUE)
arcade.finish_render()

arcade.run()
```

Now, create a one-file executable bundle file by running PyInstaller from the command-line:

```
pyinstaller main.py --onefile
```

PyInstaller generates the executable that is a bundle of your game. It puts it in the `dist\` folder under your current working directory. Look for a file named `main.exe` in `dist\`. Run this and see the example application start up!

You can copy this file wherever you want on your computer and run it. Or, share it with others. Everything your script needs is inside this executable file.

For simple games, this is all you need to know! But, if your game loads any kind of data files from disk, continue reading.

13.2 Handling Data Files

When creating a bundle, PyInstaller first examines your project and automatically identifies nearly everything your project needs (a Python interpreter, installed modules, etc). But, it can't automatically determine what data files your game is loading from disk (images, sounds, maps). So, you must explicitly tell PyInstaller about these files and where it should put them in the bundle. This is done with PyInstaller's `--add-data` flag:

```
pyinstaller main.py --add-data "stripes.jpg;."
```

The first item passed to `--add-data` is the "source" file or directory (ex: `stripes.jpg`) identifying what PyInstaller should include in the bundle. The item after the semicolon is the "destination" (ex: "`.`"), which specifies where files should be placed in the bundle, relative to the bundle's root. In the example above, the `stripes.jpg` image is copied to the root of the bundle ("`.`").

After instructing PyInstaller to include data files in a bundle, you must make sure your code loads the data files from the correct directory. When you share your game's bundle, you have no control over what directory the user will run your bundle from. This is complicated by the fact that a one-file PyInstaller bundle is uncompressed at runtime to a random temporary directory and then executed from there. This document describes one simple approach that allows your code to execute and load files when running in a PyInstaller bundle AND also be able to run when not bundled.

You need to do two things. First, the snippet below must be placed at the beginning of your script:

```
if getattr(sys, 'frozen', False) and hasattr(sys, '_MEIPASS'):
    os.chdir(sys._MEIPASS)
```

This snippet uses `sys.frozen` and `sys._MEIPASS`, which are both set by PyInstaller. The `sys.frozen` setting indicates whether code is running from a bundle ("frozen"). If the code is "frozen", the working directory is changed to the root of where the bundle has been uncompressed to (`sys._MEIPASS`). PyInstaller often uncompresses its one-file bundles to a directory named something like: `C:\Users\user\AppData\Local\Temp\_MEI123456`.

Second, once the code above has set the current working directory, all file paths in your code can be relative paths (ex: `resources\images\stripes.jpg`) as opposed to absolute paths (ex: `C:\projects\mygame\resources\images\stripes.jpg`). If you do these two things and add data files to your package as demonstrated below, your code will be able to run "normally" as well as running in a bundle.

Below are some examples that show a few common patterns of how data files can be included in a PyInstaller bundle. The examples first show a code snippet that demonstrates how data is loaded (relative path names), followed by the PyInstaller command to copy data files into the bundle. They all assume that the `os.chdir()` snippet of code listed above is being used.

13.2.1 One Data File

If you simply have one data file in the same directory as your script, refer to the data file using a relative path like this:

```
sprite = arcade.Sprite("stripes.jpg")
```

Then, you would use a PyInstaller command like this to include the data file in the bundled executable:

```
pyinstaller main.py --add-data "stripes.jpg;."
...or...
pyinstaller main.py --add-data "*.jpg;."
```

13.2.2 One Data Directory

If you have a directory of data files (such as `images`), refer to the data directory using a relative path like this:

```
sprite = arcade.Sprite("images/player.jpg")
sprite = arcade.Sprite("images/enemy.jpg")
```

Then, you would use a PyInstaller command like this to include the directory in the bundled executable:

```
pyinstaller main.py --add-data "images;images"
```

13.2.3 Multiple Data Files and Directories

You can use the `--add-data` flag multiple times to add multiple files and directories into the bundle:

```
pyinstaller main.py --add-data "player.jpg;." --add-data "enemy.jpg;." --add-data "music;
↪music"
```

13.2.4 One Directory for Everything

Although you can include every data file and directory with separate `--add-data` flags, it is suggested that you write your game so that all of your data files are under one root directory, often named `resources`. You can use subdirectories to help organize everything. An example directory tree could look like:

```
project/
|--- main.py
|--- resources/
     |--- images/
     |     |--- enemy.jpg
     |     |--- player.jpg
     |--- sound/
     |     |--- game_over.wav
     |     |--- laser.wav
     |--- text/
           |--- names.txt
```

With this approach, it becomes easy to bundle all your data with just a single `--add-data` flag. Your code would use relative pathnames to load resources, something like this:

```
sprite = arcade.Sprite("resources/images/player.jpg")
text = open("resources/text/names.txt").read()
```

And, you would include this entire directory tree into the bundle like this:

```
pyinstaller main.py --add-data "resources;resources"
```

It is worth spending a bit of time to plan out how you will layout and load your data files in order to keep the bundling process simple.

The technique of handling data files described above is just one approach. If you want more control and flexibility in handling data files, learn about the different path information that is available by reading the PyInstaller Run-Time Information documentation.

Now that you know how to install PyInstaller, include data files, and bundle your game into an executable, you have what you need to bundle your game and share it with your new fans!

13.3 Troubleshooting

13.3.1 Use a One-Folder Bundle for Troubleshooting

If you are having problems getting your bundle to work properly, it may help to temporarily omit the `--onefile` flag from the `pyinstaller` command. This will bundle your game into a one-folder bundle with an executable inside it. This allows you to inspect the contents of the folder and make sure all of the files are where you expect them to be. The one-file bundle produced by `--onefile` is simply a self-uncompressing archive of this one-folder bundle.

13.3.2 PyInstaller Not Bundling a Needed Module

In most cases, PyInstaller is able to analyze your project and automatically determine what modules to place in the bundle. But, if PyInstaller happens to miss a module, you can use the `--hidden-import MODULENAME` flag to explicitly instruct PyInstaller to include a module. See the PyInstaller documentation for more details.

13.4 Extra Details

- You will notice that after running `pyinstaller`, a `.spec` file will appear in your directory. This file is generated by PyInstaller and does not need to be saved or checked into your source code repo.
- Executable one-file bundles produced by PyInstaller's `--onefile` flag will start up slower than your original application or the one-folder bundle. This is expected because one-file bundles are ultimately just a compressed folder, so they must take time to uncompress themselves each time the bundle is run.
- By default, when PyInstaller creates a bundled application, the application opens a console window. You can suppress the creation of the console window by adding the `--windowed` flag to the `pyinstaller` command.
- See the PyInstaller documentation below for more details on the topics above, and much more.
- PyInstaller 4.x was used in this tutorial.

13.5 PyInstaller Documentation

PyInstaller is a flexible tool that can handle a wide variety of different situations. For further reading, here are links to the official PyInstaller documentation and GitHub page:

- PyInstaller Manual: https://pyinstaller.readthedocs.io/en/stable/
- PyInstaller GitHub: https://github.com/pyinstaller/pyinstaller

CHAPTER

FOURTEEN

COMPILING A GAME WITH NUITKA

So you have successfully written your dream game with Arcade and now, you want to share it with your friends and family. Good idea! But there is a *small* issue. Sadly, they are not a tech geek as big as you are and don't have any knowledge about Python and its working :(. Though *Bundling a Game with PyInstaller is* a good option, the executables it produces can sometime take up a good amount of space and antiviruses raise false positives almost every time. But Nuitka is here to solve all your problems!

Nuitka is a tool which compiles your Python code to machine code directly, and bundles your application's source code in dll files. This way, you get two benefits:

- The source code is safe in dll files.
- The application gets a performance boosts in many cases.
- The resulting executable's size is small.

We are using Windows for this tutorial, but most of the commands can be used as-it-is on other platforms including Linux and Mac.

Warning: Builds are platform dependent!

For example, a Windows build will not work out-of-the-box on a different OS. The same goes for Linux and Mac builds on other platforms.

You can use a Mac or a Linux system to compile your game for those platforms.

To compile for a different platform than your current one, you may be able to use a Virtual Machine or WINE/Proton. However, these options are not officially supported and are not covered in this tutorial.

14.1 Compiling a Simple Arcade Script

For this tutorial, we will use the code from *Simple Platformer*.

- First, we have to install Nuitka with the following command:

```
pip install nuitka
```

We will be using the code from this file.

Converting that code to a standalone executable is as easy as:

```
python -m nuitka 17_views.py --standalone --enable-plugin=numpy
```

Now sit back and relax. Might as well go and grab a cup of coffee since compilation takes time, sometimes maybe up to 2 hours, depending on your machine's specs. After the process is finished, two new folders named `17_views.py.dist` and `17_views.py.build` will popup. You can safely ignore the build folder for now. Just go to the dis folder and run `17_views.exe` file , present in there. If there are no errors, then the application should work perfectly.

Congratulations! You have successfully compiled your Python code to a standalone executable!

Note: If you want to compile the code to a single file instead of a folder, just remove the `standalone` flag and add the `onefile` flag!

14.2 But What About Data Files And Folders?

Sometimes, our application also uses custom data files which may include sound effects, fonts etc… In order to bundle them with the application, just use the `include-data-file` or `include-data-dir` flag:

```
python -m nuitka 17_views.py --standalone --enable-plugin=numpy --include-data-file=C:/
↪Users/Hunter/Desktop/my_game/my_image.png=.
```

This will copy the file named `my_image.png` at the specified location to the root of the executable.

To bundle a whole folder:

```
python -m nuitka 17_views.py --standalone --enable-plugin=numpy --include-data-dir=C:/
↪Users/Hunter/Desktop/my_game/assets=.
```

This will copy the whole folder named `assets` at the specified location to the root of the executable.

14.3 Removing The Console Window

You might have noticed that while opening the executable, a console window automatically opens. Even though it is helpful in debugging and errors, it does look ugly. You might think, is there a way to force the console output to a logs file? Well, thanks to Nuitka, this is also possible:

```
python -m nuitka 17_views.py --standalone --windows-force-stderr-spec=%PROGRAM%logs.txt -
↪-windows-force-stdout-spec=%PROGRAM%output.txt
```

This will automatically create two files, viz `logs.txt` and `output.txt` in the executable directory which will contain the stderr and stdout output respectively!

14.4 What About A Custom Taskbar Icon?

Nuitka provides us with the `windows-icon-from-ico` and `windows-icon-from-exe` flags (**varies for each OS**) to set custom icons. The first flag takes a `.png` or a `.ico` file and sets it as the app icon:

```
python -m nuitka 17_views.py --standalone --windows-icon-from-ico=icon.png
```

This will set the app icon to icon.png

```
python -m nuitka 17_views.py --standalone --windows-icon-from-exe=C:\Users\Hunter\
↪AppData\Local\Programs\Python\Python310/python.exe
```

This will set the app icon to Python's icon

14.5 Additional Information

- This tutorial was tested with Nutika 0.7.x. Later releases are likely to work.

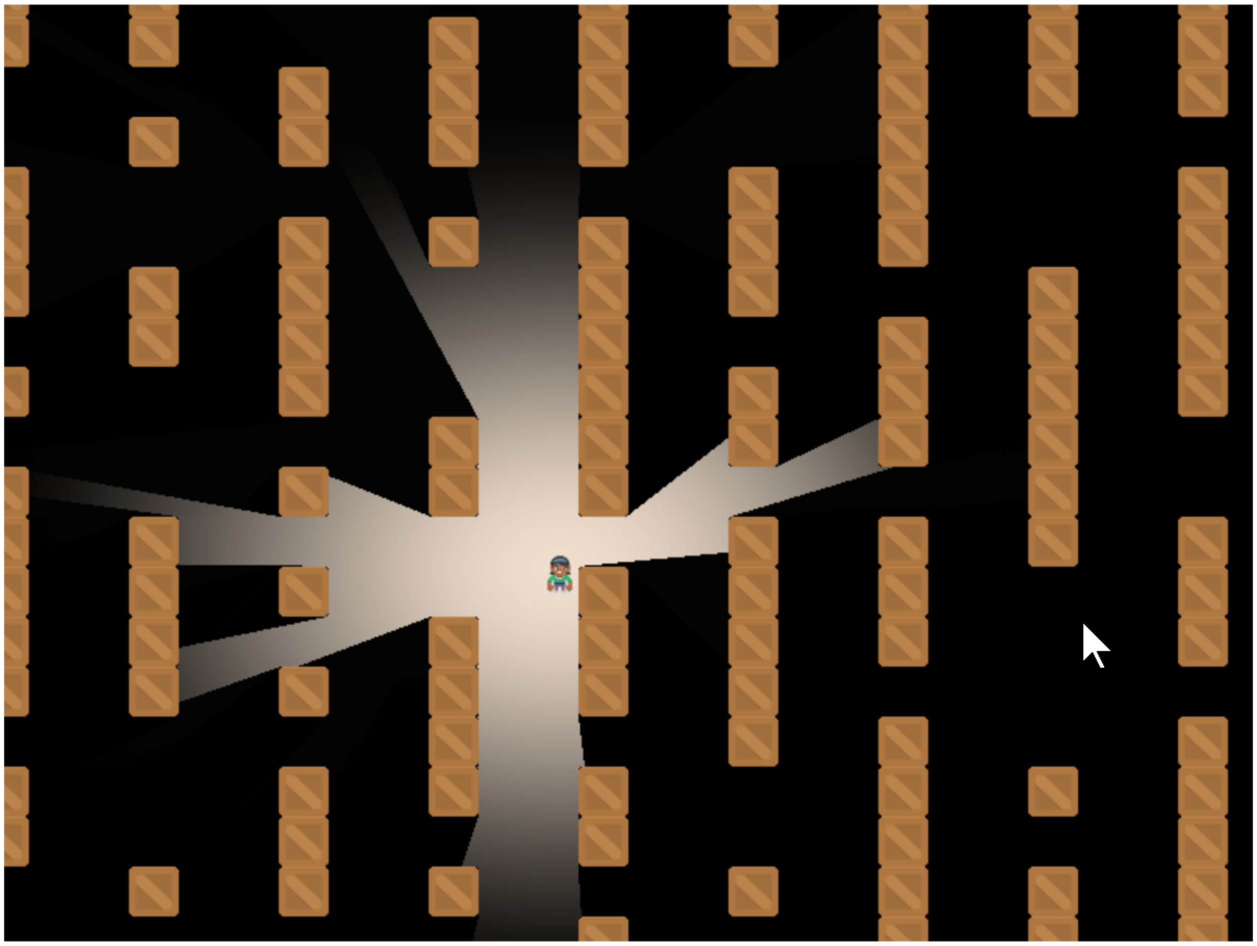
CHAPTER

FIFTEEN

SHADERS

Shaders are small programs which specify how graphics hardware should draw & shade objects. They offer power, flexibility, and efficiency far beyond what you could achieve using shapes or `Sprite` instances alone. The tutorials below serve as an introduction to shaders.

15.1 Ray-casting Shadows

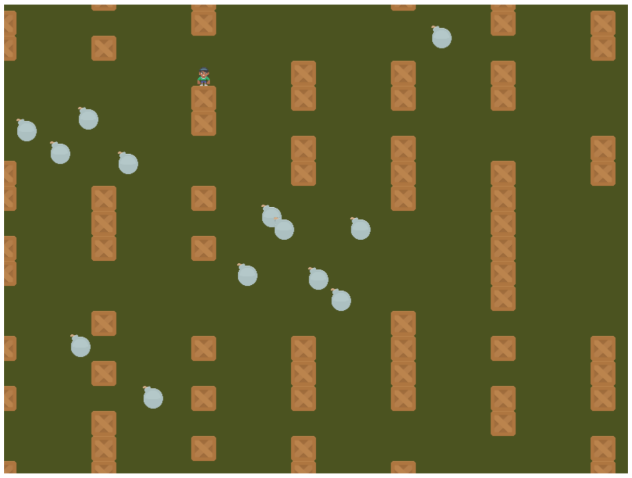


A common effect for many games is **ray-casting**. Having the user only be able to see what is directly in their line-of-sight.

This can be done quickly using **shaders**. These are small programs that run on the graphics card. They can take advantage of the **Graphics Processing Unit**. The GPU has a lot of mini-CPUs dedicated to processing graphics much faster than your main computer's CPU can.

15.1.1 Starting Program

Before we start adding shadows, we need a good starting program. Let's create some crates to block our vision, some bombs to hide in them, and a player character:



The listing for this starting program is available at raycasting_start.

15.1.2 Step 1: Add-In the Shadertoy

What is Shadertoy?

Where does the name Shadertoy come from? This class is designed to mimic the Shadertoy website. The website makes it easy to experiment with shaders, and those shaders can be run using the Arcade library.

Now, let's create a shader. We can program shaders using Arcade's `Shadertoy` class.

We'll modify our prior program to import the Shadertoy class:

Listing 1: Import Shadertoy

```python
from arcade.experimental import Shadertoy
```

Next, we'll need some shader-related variables. In addition to a variable to hold the shader, we are also going to need to keep track of a couple **frame buffer objects** (FBOs). You can store image data in an FBO and send it to the shader program. An FBO is held on the graphics card. Manipulating an FBO there is much faster than working with one in loaded into main memory.

Not just for images!

FBOs can hold more than just image-related data, but for now, just think of them as images.

Shadertoy has four built-in **channels** that our shader programs can work with. Channels can be mapped to FBOs. This allows us to pass image data to our shader program for it to process. The four channels are numbered 0 to 3.

We'll be using two channels to cast shadows. We will use the `channel0` variable to hold our barriers that can cast shadows. We will use the `channel1` variable to hold the ground, bombs, or anything we want to be hidden by shadows.

Listing 2: Create & initialize shader variables

```
    def __init__(self, width, height, title):
        super().__init__(width, height, title)

        # The shader toy and 'channels' we'll be using
        self.shadertoy = None
        self.channel0 = None
        self.channel1 = None
        self.load_shader()

        # Sprites and sprite lists
        self.player_sprite = None
        self.wall_list = arcade.SpriteList()
        self.player_list = arcade.SpriteList()
        self.bomb_list = arcade.SpriteList()
        self.physics_engine = None

        self.generate_sprites()
        self.background_color = arcade.color.ARMY_GREEN
```

These are just empty place-holders. We'll load our shader and create FBOs to hold the image data we send the shader in a `load_shader` method: This code creates the shader and the FBOs:

Listing 3: Create the shader, and the FBOs

```
    def load_shader(self):
        # Size of the window
        window_size = self.get_size()

        # Create the shader toy, passing in a path for the shader source
        self.shadertoy = Shadertoy.create_from_file(window_size, "step_01.glsl")

        # Create the channels 0 and 1 frame buffers.
        # Make the buffer the size of the window, with 4 channels (RGBA)
        self.channel0 = self.shadertoy.ctx.framebuffer(
            color_attachments=[self.shadertoy.ctx.texture(window_size, components=4)]
        )
        self.channel1 = self.shadertoy.ctx.framebuffer(
            color_attachments=[self.shadertoy.ctx.texture(window_size, components=4)]
        )

        # Assign the frame buffers to the channels
        self.shadertoy.channel_0 = self.channel0.color_attachments[0]
        self.shadertoy.channel_1 = self.channel1.color_attachments[0]
```

As you'll note, the method loads a "glsl" program from another file. Our ray-casting program will be made of two files. One file will hold our Python program, and one file will hold our Shader program. Shader programs are written in a language called OpenGL Shading Language (GLSL). This language's syntax is similar to C, Java, or C#.

Our first shader will be straight-forward. It will just take input from channel 0 and copy it to the output.

Listing 4: GLSL Program for Step 1

```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec2 normalizedFragCoord = fragCoord/iResolution.xy;
    fragColor = texture(iChannel0, normalizedFragCoord);
}
```

How does this shader work? For each point in our output, this `mainImage` function runs and calculates our output color. For a window that is 800x600 pixels, this function runs 480,000 times for each frame. Modern GPUs can have anywhere between 500-5,000 "cores" that can calculate these points in parallel for faster processing.

Our current coordinate we are calculating we've brought in as a parameter called `fragCoord`. The function needs to calculate a color for this coordinate and store it the output variable `fragColor`. You can see both the input and output variables in the parameters for the `mainImage` function. Note that the input data is labeled `in` and the output data is labeled `out`. This may be a bit different than what you are used to.

The `vec2` data type is an array of two numbers. Likewise there are `vec3` and `vec4` data types. These can be used to store coordinates, and also colors.

Or first step is to normalize the x, y coordinate to a number between 0.0 and 1.0. This normalized two-number x/y vector we store in `normalizedFragCoord`.

```
vec2 p = fragCoord/iResolution.xy;
```

We need to grab the color at this point `curPoint` from the channel 0 FBO. We can do this with the built-in `texture` function:

```
texture(iChannel0, curPoint)
```

Then we store it to our "out" `fragColor` variable and we are done:

```
fragColor = texture(iChannel0, normalizedCoord);
```

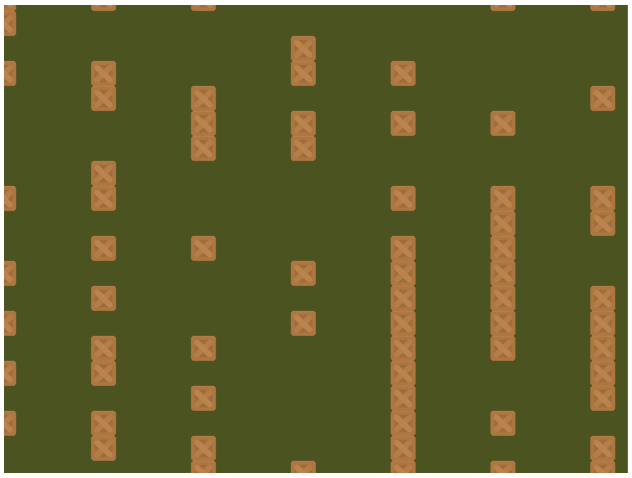
Now that we have our shader, a couple FBOs, and our initial GLSL program, we can flip back to our Python program and update the drawing code to use them:

Listing 5: Drawing using the shader

```
    def on_draw(self):
        # Select the channel 0 frame buffer to draw on
        self.channel0.use()
        self.channel0.clear()
        # Draw the walls
        self.wall_list.draw()

        # Select this window to draw on
        self.use()
        # Clear to background color
        self.clear()
        # Run the shader and render to the window
        self.shadertoy.render()
```

When we run `self.channel0.use()`, all subsequent drawing commands will draw not to the screen, but our FBO image buffer. When we run `self.use()` we'll go back to drawing on our window.

Running the program, our output should look like:



- raycasting_step_01 ← Full listing of where we are right now
- raycasting_step_01_diff ← What we changed to get here

15.1.3 Step 2: Simple Shader Experiment

How do we know our shader is really working? As it is just straight copying everything across, it is hard to tell.

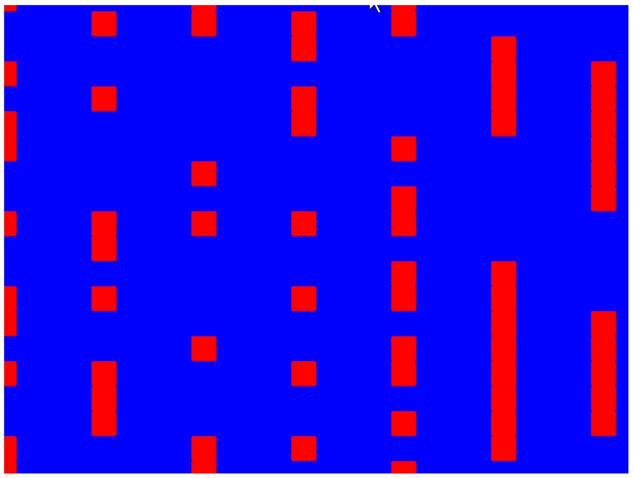
We can modify our shader to get the current texture color and store it in the variable `inColor`. A color has four components, red-green-blue and alpha. If the alpha is above zero, we can output a red color. If the alpha is zero, we output a blue color.

Note: Colors in OpenGL are specified in RGB or RGBA format. But instead of numbers going from 0-255, each component is a floating point number from 0.0 to 1.0.

Listing 6: GLSL Program for Step 2

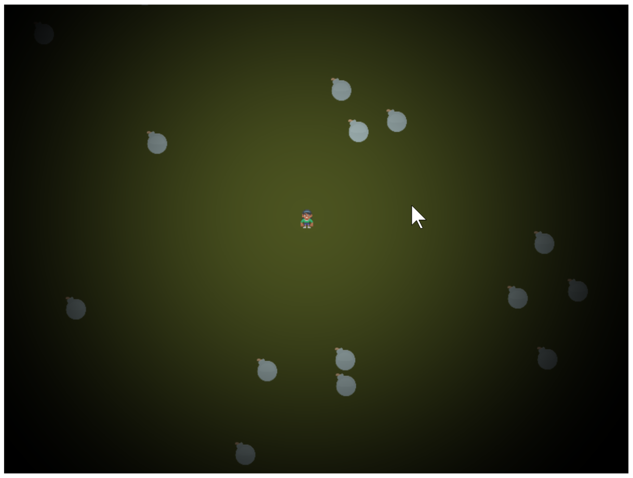
```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec2 normalizedFragCoord = fragCoord/iResolution.xy;
    vec4 inColor = texture(iChannel0, normalizedFragCoord);
    if (inColor.a > 0.0)
        // Set to a red color
        fragColor = vec4(1.0, 0.0, 0.0, 1.0);
    else
        // Set to a blue color
        fragColor = vec4(0.0, 0.0, 1.0, 1.0);
}
```

Giving us a resulting image that looks like:



15.1.4 Step 3: Creating a Light

Our next step is to create a light. We'll be fading between no light (black) and whatever we draw in Channel 1.



In this step, we won't worry about drawing the walls yet.

This step will require us to pass additional data into our shader. We'll do this using **uniforms**. We will pass in *where* the light is, and the light *size*.

We first declare and use the variables in our shader program.

Listing 7: GLSL Program for Step 3

```
// x, y position of the light
uniform vec2 lightPosition;
// Size of light in pixels
uniform float lightSize;
```

Next, we need to know how far away this point is from the light. We do that by subtracting this point from the light position. We can perform mathematical operations on vectors, so we just subtract. Then we use the build-in `length` function to get a floating point number of how long the length of this vector is.

Listing 8: GLSL Program for Step 3

```
    // Distance in pixels to the light
    float distanceToLight = length(lightPosition - fragCoord);
```

Next, we need to get the coordinate of the pixel we are calculating, but **normalized**. The coordinates will range from 0.0 to 1.0, with the left bottom of the window at (0,0), and the top right at (1,1). Normalized coordinates are used in shaders to make scaling up and down easy.

Listing 9: GLSL Program for Step 3

```
    // Normalize the fragment coordinate from (0.0, 0.0) to (1.0, 1.0)
    vec2 normalizedFragCoord = fragCoord/iResolution.xy;
```

Then we need to calculate how much light is falling on this coordinate. This number will also be normalized. A number of 0.0 will be in complete shadow, and 1.0 will be fully lit.

Linear or Squared?

The smoothstep function scales linearly. (Well, actually is uses Hermite interpolation, but mostly linear.) In reality, the intensity of light is inversely proportional to the square of the distance in reality. The implementation of this is left up to the reader.

We will use the built-in `smoothstep` function that will take how large our light size is, and how far we are from the light. Then scale it from a number 0.0 to 1.0.

If we are 0.0 pixels from the light, we'll get a 0.0 back. If we are halfway to the light we'll get 0.5. If we are at the light's edge, we'll get 1.0. If we are beyond the light's edge we'll get 1.0.

Unfortunately this is backwards from what we want. We want 1.0 at the center, and 0.0 outside the light. So a simple subtraction from 1.0 will solve this issue.

Listing 10: GLSL Program for Step 3

```
    // Start our mixing variable at 1.0
    float lightAmount = 1.0;

    // Find out how much light we have based on the distance to our light
    lightAmount *= 1.0 - smoothstep(0.0, lightSize, distanceToLight);
```

Next, we are going to use the built-in `mix` function and the `lightAmount` variable to alternate between whatever is in channel 1, and a black shadow color.

Listing 11: GLSL Program for Step 3

```
    // We'll alternate our display between black and whatever is in channel 1
    vec4 blackColor = vec4(0.0, 0.0, 0.0, 1.0);

    // Our fragment color will be somewhere between black and channel 1
    // dependent on the value of b.
    fragColor = mix(blackColor, texture(iChannel1, normalizedFragCoord), lightAmount);
```

Finally we'll go back to the Python program and update our `on_draw` method to:

- Draw the bombs into channel 1.
- Send the player position and the size of the light using the uniform.
- Draw the player character on the window.

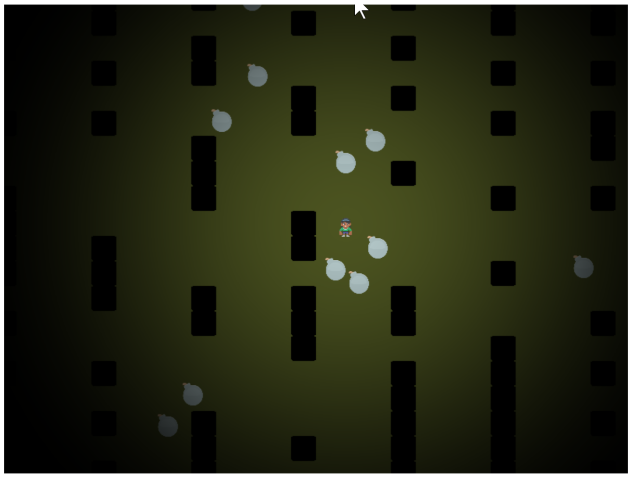
Listing 12: Drawing using the shader

```
    def on_draw(self):
        # Select the channel 0 frame buffer to draw on
        self.channel0.use()
        self.channel0.clear()
        # Draw the walls
        self.wall_list.draw()

        self.channel1.use()
        self.channel1.clear()
        # Draw the bombs
        self.bomb_list.draw()

        # Select this window to draw on
        self.use()
        # Clear to background color
        self.clear()
        # Run the shader and render to the window
        self.shadertoy.program['lightPosition'] = self.player_sprite.position
        self.shadertoy.program['lightSize'] = 300
        self.shadertoy.render()
        # Draw the player
        self.player_list.draw()
```

Note: If you set a uniform variable using `program`, that variable has to exist in the glsl program, *and be used* or you'll get an error. The glsl compiler will automatically drop unused variables, causing a confusing error when the program says a variable is missing even if you've declared it.

- raycasting_step_03 ← Full listing of where we are right now with the Python program
- raycasting_step_03_diff ← What we changed to get here
- raycasting_step_03_gl ← Full listing of where we are right now with the GLSL program
- raycasting_step_03_gl_diff ← What we changed to get here

15.1.5 Step 4: Make the Walls Shadowed



In addition to the light, we want the walls to show up in shadow for this step. We don't need to change our Python program at all for this, just the GLSL program.

First, we'll add to our GLSL program a `terrain` function. This will sample channel 0. If the pixel there has an alpha of 0.1 or greater (a barrier to our light), we'll use the `step` function and get 1.0. Otherwise we'll get 0.0. Then, since we want this reversed, (0.0 for barriers, 1.0 for no barrier) we'll subtract from 1.0:

Listing 13: GLSL Program for Step 4

```
float terrain(vec2 samplePoint)
{
    float samplePointAlpha = texture(iChannel0, samplePoint).a;
    float sampleStepped = step(0.1, samplePointAlpha);
    float returnValue = 1.0 - sampleStepped;

    return returnValue;
}
```

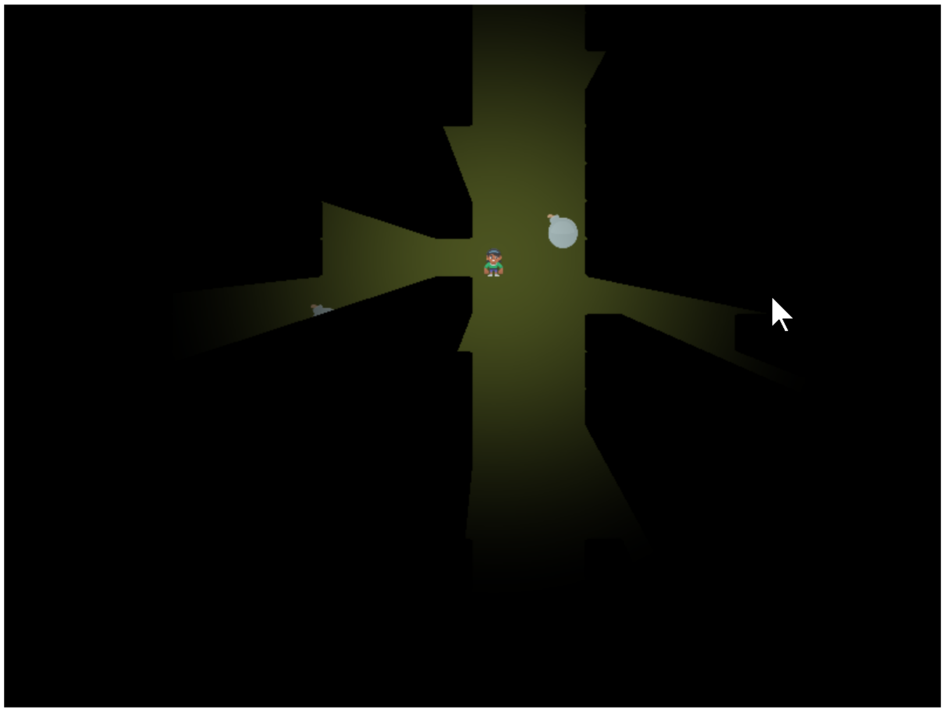
Next, we'll factor in this barrier to our light. So our light amount will be a combination of the distance from the light, and if there's a barrier object on this pixel.

Listing 14: GLSL Program for Step 4

```
    // Start our mixing variable at 1.0
    float lightAmount = 1.0;

    float shadowAmount = terrain(normalizedFragCoord);
    lightAmount *= shadowAmount;

    // Find out how much light we have based on the distance to our light
    lightAmount *= 1.0 - smoothstep(0.0, lightSize, distanceToLight);
```

- raycasting_step_04_gl ← Full listing of where we are right now with the GLSL program
- raycasting_step_04_gl_diff ← What we changed to get here

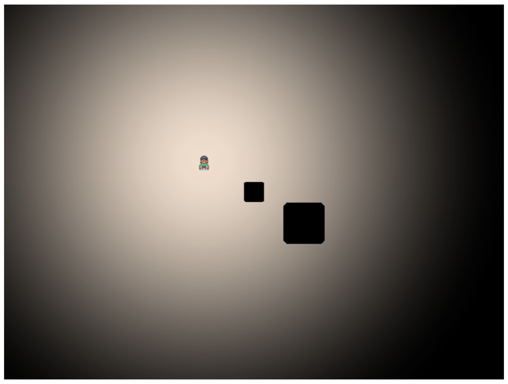
15.1.6 Step 5: Cast the Shadows



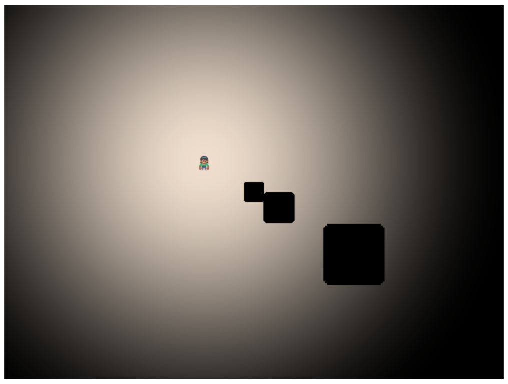
Now it is time to cast the shadows.

This involves a lot of "sampling". We start at our current point and draw a line to where the light is. We will sample "N" times along that line. If we spot a barrier, our coordinate must be in shadow.

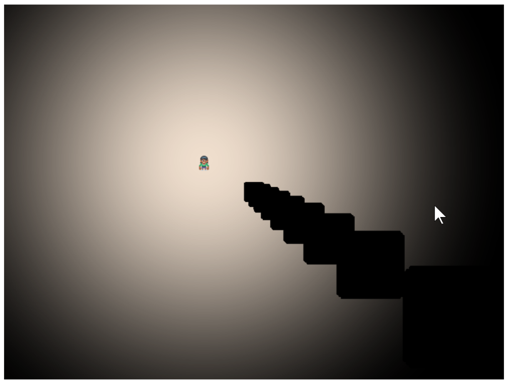
How many times do we sample? If we don't sample enough times, we miss barriers and end up with weird shadows. This first image is if we only sample twice. Once where we are, and once in the middle:



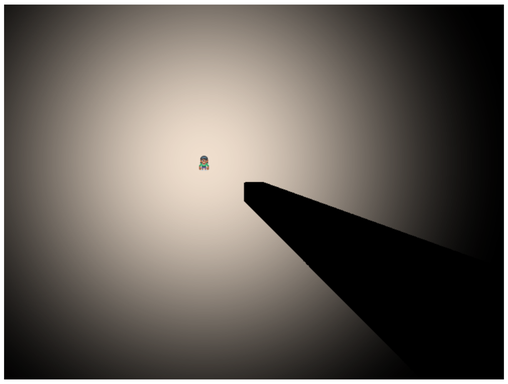
If N is three, we end up with three copies of the shadow:



With an N of 10:



We can use an N of 500 to get a good quality shadow. We might need more if your barriers are small, and the light range is large.

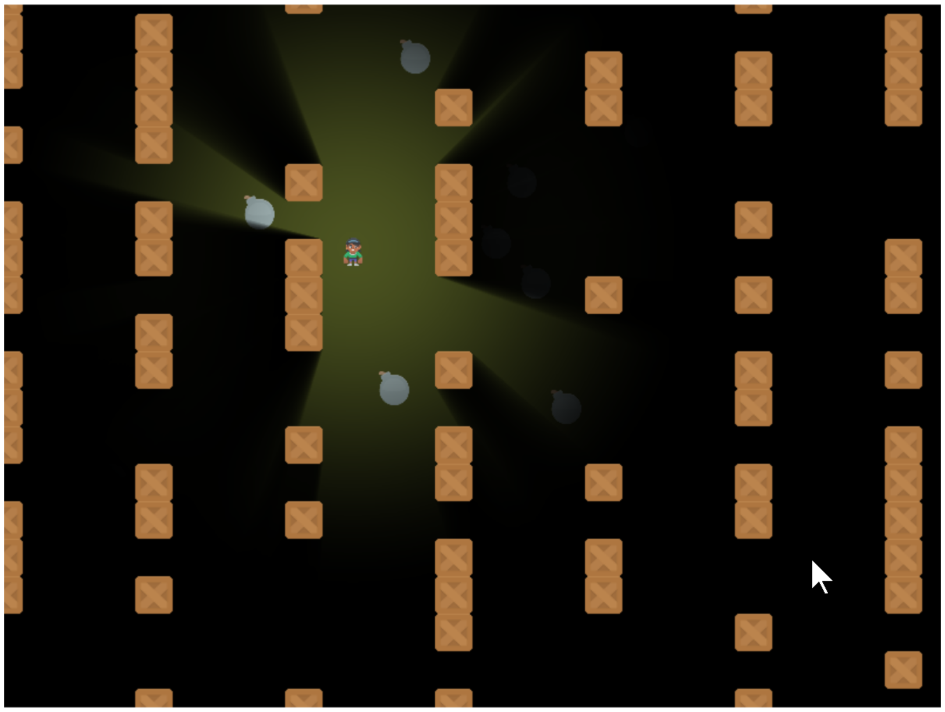


Keep in mind there is a speed trade-off. With 800x600 pixels, we have 480,000 pixels to calculate. If each of those pixels has a loop that does 500 samples, we are sampling 480,000x500 = 240,000 sample per frame, or 14.4 million samples per second, still very do-able with modern graphics cards.

But what if you scale up? A 4k monitor would need 247 billion samples per second! There are optimizations that would be done, such as exiting out of the `for` loop once we are in shadow, and not calculating for points beyond the light's range. We aren't covering that here, but even with 2D, it will be important to understand what the shader is doing to keep reasonable performance.

- raycasting_step_05_gl ← Full listing of where we are right now with the GLSL program
- raycasting_step_05_gl_diff ← What we changed to get here

15.1.7 Step 6: Soft Shadows and Wall Drawing



With one more line of code, we can soften up the shadows so they don't have such a "hard" edge to them.

To do this, modify the `terrain` function in our GLSL program. Rather than return 0.0 or 1.0, we'll return 0.0 or 0.98. This allows edges to only partially block the light.

Listing 15: GLSL Program for Step 6

```
float terrain(vec2 samplePoint)
{
    float samplePointAlpha = texture(iChannel0, samplePoint).a;
    float sampleStepped = step(0.1, samplePointAlpha);
    float returnValue = 1.0 - sampleStepped;

    // Soften the shadows. Comment out for hard shadows.
    // The closer the first number is to 1.0, the softer the shadows.
    returnValue = mix(0.98, 1.0, returnValue);
```

And then we can go ahead and draw the barriers back on the screen so we can see what is casting the shadows.

Listing 16: Step 6, Draw the Barriers

```
    def on_draw(self):
        # Select the channel 0 frame buffer to draw on
        self.channel0.use()
        self.channel0.clear()
        # Draw the walls
        self.wall_list.draw()

        self.channel1.use()
        self.channel1.clear()
        # Draw the bombs
        self.bomb_list.draw()

        # Select this window to draw on
        self.use()
        # Clear to background color
        self.clear()
        # Run the shader and render to the window
        self.shadertoy.program['lightPosition'] = self.player_sprite.position
        self.shadertoy.program['lightSize'] = 300
        self.shadertoy.render()

        # Draw the walls
        self.wall_list.draw()

        # Draw the player
        self.player_list.draw()
```

- raycasting_step_06 ← Full listing of where we are right now with the Python program
- raycasting_step_06_gl ← Full listing of where we are right now with the GLSL program
- raycasting_step_06_gl_diff ← What we changed to get here

15.1.8 Step 7 - Support window resizing

What if you need to resize the window? First enable resizing:

You'll need to enable resizing in the window's `__init__`:

Listing 17: Enable resizing

```
    def __init__(self, width, height, title):
        super().__init__(width, height, title, resizable=True)
```

Then we need to override the `Window.resize` method to also resize the shadertoy:

Listing 18: Resizing the window

```
    def on_resize(self, width: int, height: int):
        super().on_resize(width, height)
        self.shadertoy.resize((width, height))
```

- raycasting_step_07 ← Full listing of where we are right now with the Python program
- raycasting_step_07_diff ← What we changed to get here

15.1.9 Step 8 - Support scrolling

What if we want to scroll around the screen? Have a GUI that doesn't scroll?

First, we'll add a camera for the scrolling parts of the screen (sprites) and another camera for the non-scrolling GUI bits. Also, we'll create some text to toss on the screen as something for the GUI.

Listing 19: MyGame.__init__

```
    def __init__(self, width, height, title):
        super().__init__(width, height, title, resizable=True)

        # The shader toy and 'channels' we'll be using
        self.shadertoy = None
        self.channel0 = None
        self.channel1 = None
        self.load_shader()

        # Sprites and sprite lists
        self.player_sprite = None
        self.wall_list = arcade.SpriteList()
        self.player_list = arcade.SpriteList()
        self.bomb_list = arcade.SpriteList()
        self.physics_engine = None

        # Create cameras used for scrolling
        self.camera_sprites = arcade.SimpleCamera()
        self.camera_gui = arcade.SimpleCamera()

        self.generate_sprites()

        # Our sample GUI text
```

(continues on next page)

(continued from previous page)

```python
        self.score_text = arcade.Text("Score: 0", 10, 10, arcade.color.WHITE, 24)

        self.background_color = arcade.color.ARMY_GREEN
```

Next up, we need to draw and use the cameras. This complicates our shader as it doesn't care about the scrolling, so we have to pass it a position not affected by the camera position. Therefore, we subtract it out.

Listing 20: MyGame.on_draw

```python
    def on_draw(self):
        # Use our scrolled camera
        self.camera_sprites.use()

        # Select the channel 0 frame buffer to draw on
        self.channel0.use()
        self.channel0.clear()
        # Draw the walls
        self.wall_list.draw()

        self.channel1.use()
        self.channel1.clear()
        # Draw the bombs
        self.bomb_list.draw()

        # Select this window to draw on
        self.use()
        # Clear to background color
        self.clear()

        # Calculate the light position. We have to subtract the camera position
        # from the player position to get screen-relative coordinates.
        p = (self.player_sprite.position[0] - self.camera_sprites.position[0],
             self.player_sprite.position[1] - self.camera_sprites.position[1])

        # Set the uniform data
        self.shadertoy.program['lightPosition'] = p
        self.shadertoy.program['lightSize'] = 300

        # Run the shader and render to the window
        self.shadertoy.render()

        # Draw the walls
        self.wall_list.draw()

        # Draw the player
        self.player_list.draw()

        # Switch to the un-scrolled camera to draw the GUI with
        self.camera_gui.use()
        # Draw our sample GUI text
        self.score_text.draw()
```

When we update, we need to scroll the camera to where the user is:

Listing 21: MyGame.on_update

```python
    def on_update(self, delta_time):
        """ Movement and game logic """

        # Call update on all sprites (The sprites don't do much in this
        # example though.)
        self.physics_engine.update()
        # Scroll the screen to the player
        self.scroll_to_player()
```

We need to implement the `scroll_to_player` method ourselves.

First, we import pyglet's `Vec2` class to make the math faster to implement:

Listing 22: Import pyglet's 2D vector class to help with math

```python
import random
from pyglet.math import Vec2

import arcade
from arcade.experimental import Shadertoy
```

Then, we implement the `MyGame.scroll_to_player` method:

Listing 23: MyGame.scroll_to_player

```python
    def scroll_to_player(self, speed=CAMERA_SPEED):
        """
        Scroll the window to the player.

        if CAMERA_SPEED is 1, the camera will immediately move to the desired position.
        Anything between 0 and 1 will have the camera move to the location with a␣
↪smoother
        pan.
        """

        position = Vec2(self.player_sprite.center_x - self.width / 2,
                        self.player_sprite.center_y - self.height / 2)
        self.camera_sprites.move_to(position, speed)
```

Finally, when we resize the window, we have to resize our cameras:

Listing 24: MyGame.on_resize

```python
    def on_resize(self, width: int, height: int):
        super().on_resize(width, height)
        self.camera_sprites.resize(width, height)
        self.camera_gui.resize(width, height)
        self.shadertoy.resize((width, height))
```

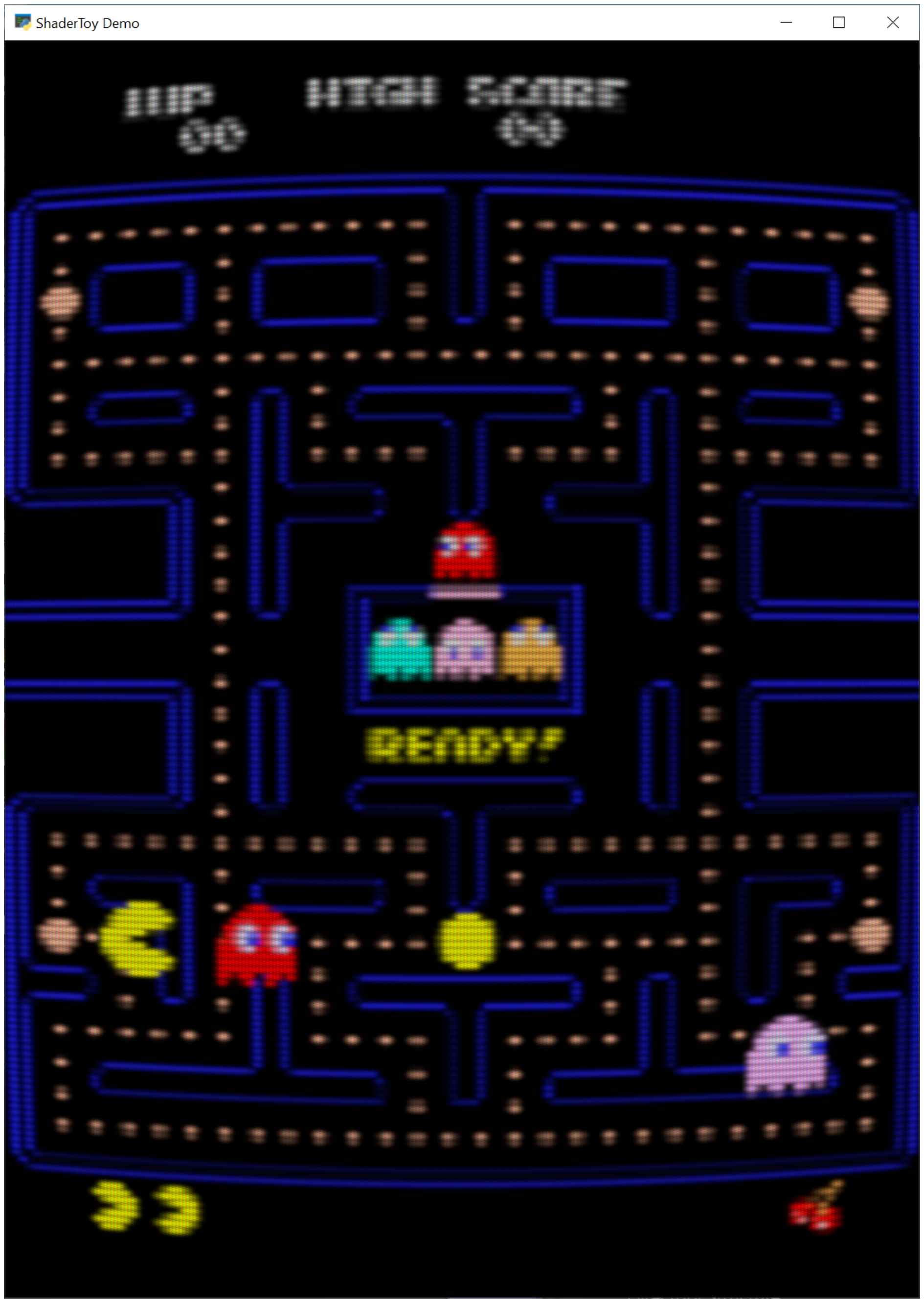
- raycasting_step_08 ← Full listing of where we are right now with the Python program
- raycasting_step_08_diff ← What we changed to get here

15.1.10 Bibliography

Before I wrote this tutorial I did not know how these shadows were made. I found the sample code Simple 2d Ray-Cast Shadow by jt which allowed me to very slowly figure out how to cast shadows.

15.2 CRT Filter

If you'd like an 80s feel to your games, you can use the built-in CRT filter.



You can create a CRT filter with code like this:

```
# Create the crt filter
self.crt_filter = CRTFilter(width, height,
                            resolution_down_scale=6.0,
                            hard_scan=-8.0,
                            hard_pix=-3.0,
                            display_warp = Vec2(1.0 / 32.0, 1.0 / 24.0),
                            mask_dark=0.5,
                            mask_light=1.5)
```

You can play around with the parameters to get an idea of what they do. For example:

Resolution Down Sampling

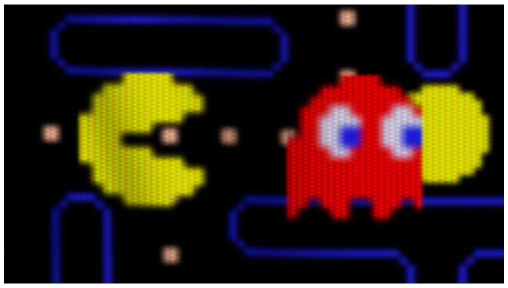


Fig. 1: resolution_down_scale = 1

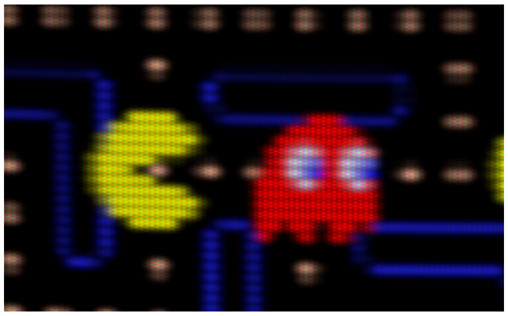


Fig. 2: resolution_down_scale = 6

To use the CRT Filter, your on_draw method should first draw everything to the CRT filter. At this point, nothing draws to the screen, we are just drawing to an internal frame buffer.

Then, once everything is drawn to the CRT filter, render that filter to the screen.

```
# Draw our stuff into the CRT filter instead of on screen
self.crt_filter.use()
self.crt_filter.clear()
self.sprite_list.draw()

# Next, switch back to the screen and dump the contents of the CRT filter
# to it.
self.use()
self.clear()
self.crt_filter.draw()
```

15.2.1 Full Example Code

The example code just animates a Pac-Man image. You can toggle the CRT filter on or off by hitting the space bar.

Images to run this example can be found here: https://github.com/pythonarcade/arcade/tree/development/doc/tutorials/crt_filter

```python
import arcade
from arcade.experimental.crt_filter import CRTFilter
from pyglet.math import Vec2


# Store our screen dimensions & title in a convenient place
SCREEN_WIDTH = 800
SCREEN_HEIGHT = 1100
SCREEN_TITLE = "ShaderToy Demo"


class MyGame(arcade.Window):

    def __init__(self, width, height, title):
        super().__init__(width, height, title, resizable=True)

        # Create the crt filter
        self.crt_filter = CRTFilter(width, height,
                                    resolution_down_scale=6.0,
                                    hard_scan=-8.0,
                                    hard_pix=-3.0,
                                    display_warp=Vec2(1.0 / 32.0, 1.0 / 24.0),
                                    mask_dark=0.5,
                                    mask_light=1.5)

        self.filter_on = True

        # Create some stuff to draw on the screen
        self.sprite_list = arcade.SpriteList()

        full = arcade.Sprite("Pac-man.png")
        full.center_x = width / 2
        full.center_y = height / 2
        full.scale = width / full.width
        self.sprite_list.append(full)

        my_sprite = arcade.Sprite(
            "pac_man_sprite_sheet.png",
            scale=5, image_x=4, image_y=65, image_width=13, image_height=15)
        my_sprite.change_x = 1
        self.sprite_list.append(my_sprite)
        my_sprite.center_x = 100
        my_sprite.center_y = 300

        my_sprite = arcade.Sprite(
            "pac_man_sprite_sheet.png",
            scale=5, image_x=4, image_y=81, image_width=13, image_height=15)
```

(continues on next page)

(continued from previous page)

```
        my_sprite.change_x = -1
        self.sprite_list.append(my_sprite)
        my_sprite.center_x = 800
        my_sprite.center_y = 200

        keyframes = []
        texture = arcade.load_texture("pac_man_sprite_sheet.png", x=4, y=1, width=13,␣
↪height=15)
        frame = arcade.TextureKeyframe(texture, duration=150)
        keyframes.append(frame)
        texture = arcade.load_texture("pac_man_sprite_sheet.png", x=20, y=1, width=13,␣
↪height=15)
        frame = arcade.TextureKeyframe(texture, duration=150)
        keyframes.append(frame)
        my_sprite = arcade.TextureAnimationSprite(animation=arcade.
↪TextureAnimation(keyframes))

        my_sprite.change_x = 1
        self.sprite_list.append(my_sprite)
        my_sprite.center_x = 0
        my_sprite.center_y = 300
        my_sprite.texture = texture
        my_sprite.scale = 5.0

    def on_draw(self):
        if self.filter_on:
            # Draw our stuff into the CRT filter instead of on screen
            self.crt_filter.use()
            self.crt_filter.clear()
            self.sprite_list.draw()

            # Next, switch back to the screen and dump the contents of
            # the CRT filter to it.
            self.use()
            self.clear()
            self.crt_filter.draw()
        else:
            # Draw our stuff into the screen
            self.use()
            self.clear()
            self.sprite_list.draw()

    def on_update(self, dt):
        # Keep track of elapsed time
        self.sprite_list.update()
        self.sprite_list.update_animation(dt)
        for sprite in self.sprite_list:
            if sprite.left > self.width and sprite.change_x > 0:
                sprite.right = 0
            if sprite.right < 0 and sprite.change_x < 0:
                sprite.left = self.width
```

(continues on next page)

(continued from previous page)

```
    def on_key_press(self, key, mod):
        if key == arcade.key.SPACE:
            self.filter_on = not self.filter_on


if __name__ == "__main__":
    MyGame(SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_TITLE)
    arcade.run()
```

15.3 Shader Toy - Glow

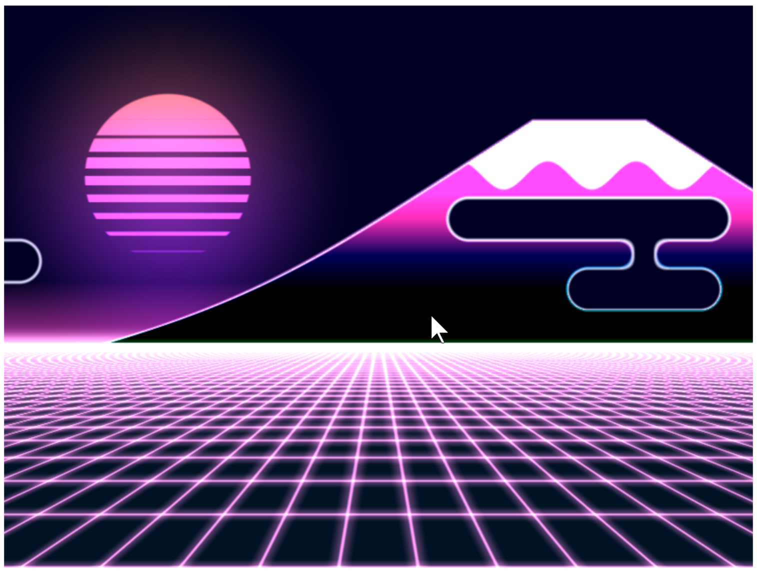


Fig. 3: cyber_fuji_2020

Graphics cards can run programs written in the C-like language OpenGL Shading Language, or GLSL for short. These programs can be easily parallelized and run across the processors of the graphics card GPU.

Shaders take a bit of set-up to write. The ShaderToy website has standardized some of these and made it easier to experiment with writing shaders. The website is at:

https://www.shadertoy.com/

Arcade includes additional code making it easier to run these ShaderToy shaders in an Arcade program. This tutorial helps you get started.

15.3.1 PyCon 2022 Slides

This tutorial is scheduled to be presented at 2022 PyCon US. Here are the slides for that presentation:

15.3.2 Step 1: Open a window

This is simple program that just opens a basic Arcade window. We'll add a shader in the next step.

Listing 25: Open a window

```python
import arcade

# Derive an application window from Arcade's parent Window class
class MyGame(arcade.Window):

    def __init__(self):
        # Call the parent constructor
        super().__init__(width=1920, height=1080)

    def on_draw(self):
        # Clear the screen
        self.clear()

if __name__ == "__main__":
    MyGame()
    arcade.run()
```

15.3.3 Step 2: Load a shader

This program will load a GLSL program and display it. We'll write our shader in the next step.

Listing 26: Run a shader

```python
import arcade
from arcade.experimental import Shadertoy


# Derive an application window from Arcade's parent Window class
class MyGame(arcade.Window):

    def __init__(self):
        # Call the parent constructor
        super().__init__(width=1920, height=1080)

        # Load a file and create a shader from it
        shader_file_path = "circle_1.glsl"
        window_size = self.get_size()
        self.shadertoy = Shadertoy.create_from_file(window_size, shader_file_path)
```

(continues on next page)

(continued from previous page)

```python

    def on_draw(self):
        # Run the GLSL code
        self.shadertoy.render()

if __name__ == "__main__":
    MyGame()
    arcade.run()
```

Note: The proper way to read in a file to a string is using a **with** statement. For clarity/brevity our code isn't doing that in the presentation. Here's the proper way to do it:

```python
file_name = "circle_1.glsl"
with open(file_name) as file:
    shader_source = file.read()
self.shadertoy = Shadertoy(size=self.get_size(),
                           main_source=shader_source)
```

15.3.4 Step 3: Write a shader

Next, let's create a simple first GLSL program. Our program will:

- Normalize the coordinates. Instead of 0 to 1024, we'll go 0.0 to 1.0. This is standard practice, and allows us to work independently of resolution. Resolution is already stored for us in a standardized variable named `iResolution`.
- Next, we'll use a white color as default. Colors are four floating point RGBA values, ranging from 0.0 to 1.0. To start with, we'll set just RGB and use 1.0 for alpha.
- If we are greater that 0.2 for our coordinate (20% of screen size) we'll use black instead.
- Set our output color, standardized with the variable name `fracColor`.

Listing 27: GLSL code for creating a shader.

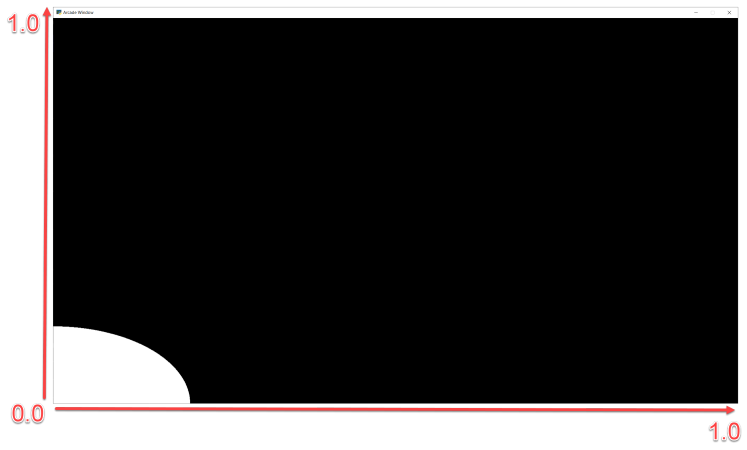
```glsl
void mainImage(out vec4 fragColor, in vec2 fragCoord) {

    // Normalized pixel coordinates (from 0 to 1)
    vec2 uv = fragCoord/iResolution.xy;

    // How far is the current pixel from the origin (0, 0)
    float distance = length(uv);

    // Are we are 20% of the screen away from the origin?
    if (distance > 0.2) {
        // Black
        fragColor = vec4(0.0, 0.0, 0.0, 1.0);
    } else {
        // White
        fragColor = vec4(1.0, 1.0, 1.0, 1.0);
```

(continues on next page)

(continued from previous page)

```
    }
}
```

The output of the program looks like this:



Other default variables you can use:

```
uniform vec3 iResolution;
uniform float iTime;
uniform float iTimeDelta;
uniform float iFrame;
uniform float iChannelTime[4];
uniform vec4 iMouse;
uniform vec4 iDate;
uniform float iSampleRate;
uniform vec3 iChannelResolution[4];
uniform samplerXX iChanneli;
```

"Uniform" means the data is the same for each pixel the GLSL program runs on.

15.3.5 Step 4: Move origin to center of screen, adjust for aspect

Next up, we'd like to center our circle, and adjust for the aspect ratio. This will give us a (0, 0) in the middle of the screen and a perfect circle.

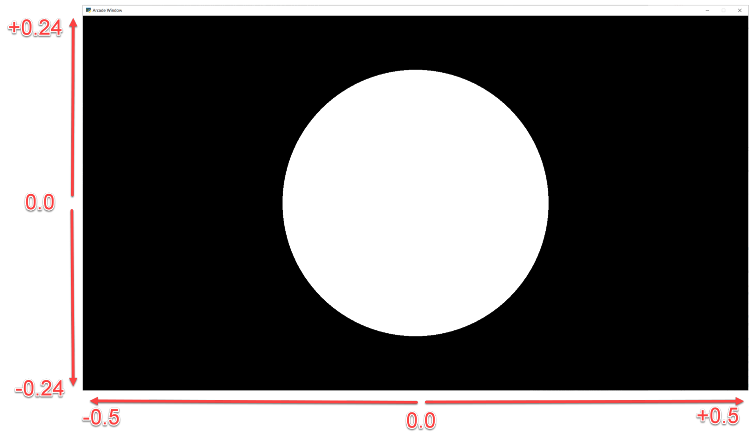
Listing 28: Center the origin

```
void mainImage(out vec4 fragColor, in vec2 fragCoord) {

    // Normalized pixel coordinates (from 0 to 1)
    vec2 uv = fragCoord/iResolution.xy;

    // Position of fragment relative to center of screen
    vec2 rpos = uv - 0.5;
    // Adjust y by aspect ratio
    rpos.y /= iResolution.x/iResolution.y;

    // How far is the current pixel from the origin (0, 0)
```

(continues on next page)

(continued from previous page)

```
    float distance = length(rpos);

    // Default our color to white
    vec3 color = vec3(1.0, 1.0, 1.0);

    // Are we are 20% of the screen away from the origin?
    if (distance > 0.2) {
        // Black
        fragColor = vec4(0.0, 0.0, 0.0, 1.0);
    } else {
        // White
        fragColor = vec4(1.0, 1.0, 1.0, 1.0);
    }
}
```



15.3.6 Step 5: Add a fade effect

We can take colors, like our white (1.0, 1.0, 1.0) and adjust their intensity by multiplying them times a float. Multiplying white times 0.5 will give us gray (0.5, 0.5, 0.5).

We can use this to create a fade effect around our circle. The inverse of the distance $\frac{1}{d}$ gives us a good curve. However the numbers are too large to adjust our white color. We can solve this by scaling it down. Run this, and adjust the scale value to see how it changes.

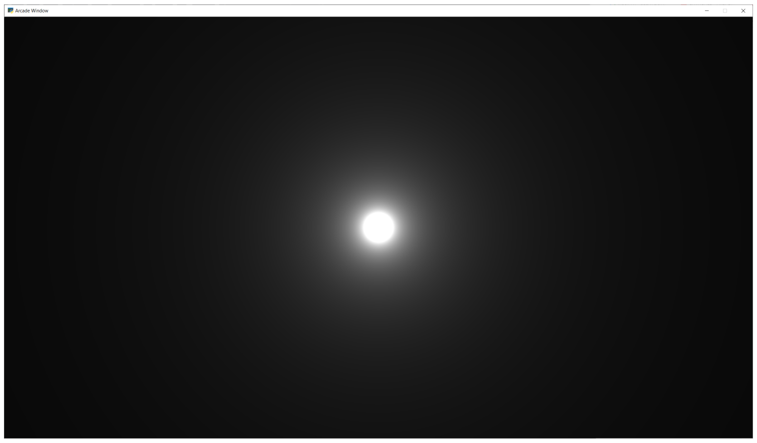
Listing 29: Add fade effect

```
void mainImage(out vec4 fragColor, in vec2 fragCoord) {

    // Normalized pixel coordinates (from 0 to 1)
    vec2 uv = fragCoord/iResolution.xy;

    // Position of fragment relative to center of screen
    vec2 rpos = uv - 0.5;
    // Adjust y by aspect ratio
    rpos.y /= iResolution.x/iResolution.y;

    // How far is the current pixel from the origin (0, 0)
```

(continues on next page)

(continued from previous page)

```
    float distance = length(rpos);
    // Use an inverse 1/distance to set the fade
    float scale = 0.02;
    float strength = 1.0 / distance * scale;

    // Fade our white color
    vec3 color = strength * vec3(1.0, 1.0, 1.0);

    // Output to the screen
    fragColor = vec4(color, 1.0);
}
```



15.3.7 Step 6: Adjust how fast we fade

We can use an exponent to adjust how steep or shallow that curve is. If we use 1.0 it will be the same, 0.5 will cause it to fade out slower, 1.5 will fade faster.

We can also change our color to orange.

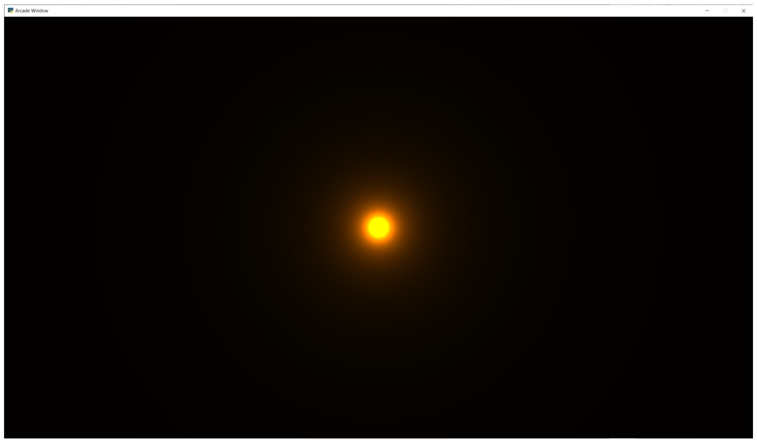
Listing 30: Adjusts fade speed

```
void mainImage(out vec4 fragColor, in vec2 fragCoord) {

    // Normalized pixel coordinates (from 0 to 1)
    vec2 uv = fragCoord/iResolution.xy;

    // Position of fragment relative to center of screen
    vec2 rpos = uv - 0.5;
    // Adjust y by aspect ratio
    rpos.y /= iResolution.x/iResolution.y;

    // How far is the current pixel from the origin (0, 0)
    float distance = length(rpos);
    // Use an inverse 1/distance to set the fade
    float scale = 0.02;
    float fade = 1.5;
    float strength = pow(1.0 / distance * scale, fade);
```

(continues on next page)

(continued from previous page)

```

    // Fade our orange color
    vec3 color = strength * vec3(1.0, 0.5, 0.0);

    // Output to the screen
    fragColor = vec4(color, 1.0);
}
```



15.3.8 Step 7: Tone mapping

Once we add color, the glow looks a bit off. We can do "tone mapping" with a bit of math if you like the look better.

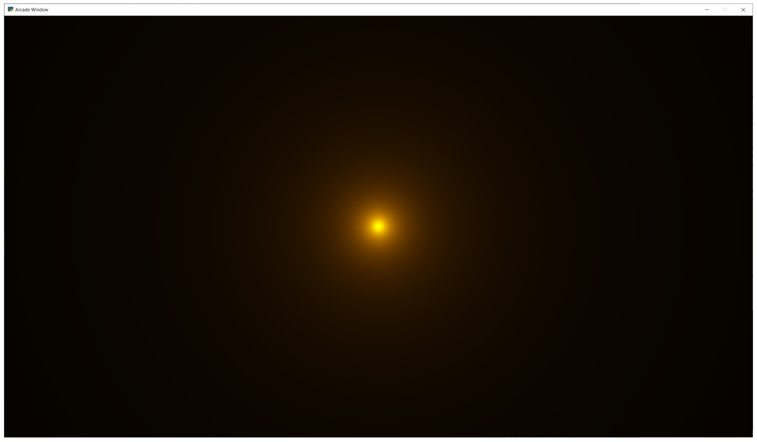
Listing 31: Tone mapping

```
void mainImage(out vec4 fragColor, in vec2 fragCoord) {

    // Normalized pixel coordinates (from 0 to 1)
    vec2 uv = fragCoord/iResolution.xy;

    // Position of fragment relative to center of screen
    vec2 rpos = uv - 0.5;
    // Adjust y by aspect ratio
    rpos.y /= iResolution.x/iResolution.y;

    // How far is the current pixel from the origin (0, 0)
    float distance = length(rpos);
    // Use an inverse 1/distance to set the fade
    float scale = 0.02;
    float fade = 1.1;
    float strength = pow(1.0 / distance * scale, fade);

    // Fade our orange color
    vec3 color = strength * vec3(1.0, 0.5, 0);

    // Tone mapping
    color = 1.0 - exp( -color );

```

(continues on next page)

(continued from previous page)

```
    // Output to the screen
    fragColor = vec4(color, 1.0);
}
```



15.3.9 Step 8: Positioning the glow

What if we want to position the glow at a certain spot? Send an x, y to center on? What if we want to control the color of the glow too?

We can send data to our shader using *uniforms*. The data we send will be the same (uniform) for each pixel rendered by the shader. The uniforms can easily be set in our Python program:

Listing 32: Run a shader

```python
import arcade
from arcade.experimental import Shadertoy


# Derive an application window from Arcade's parent Window class
class MyGame(arcade.Window):

    def __init__(self, width=1920, height=1080, glow_color=arcade.color.LIGHT_BLUE):
        # Call the parent constructor
        super().__init__(width=width, height=height)

        # Load a file and create a shader from it
        shader_file_path = "circle_6.glsl"
        window_size = self.get_size()
        self.shadertoy = Shadertoy.create_from_file(window_size, shader_file_path)
        # Set uniform light color data to send to the GLSL shader
        # from the normalized RGB components of the color.
        self.shadertoy.program['color'] = glow_color.normalized[:3]

    def on_draw(self):
        # Set uniform position data to send to the GLSL shader
        self.shadertoy.program['pos'] = self.mouse["x"], self.mouse["y"]
        # Run the GLSL code
```

(continues on next page)

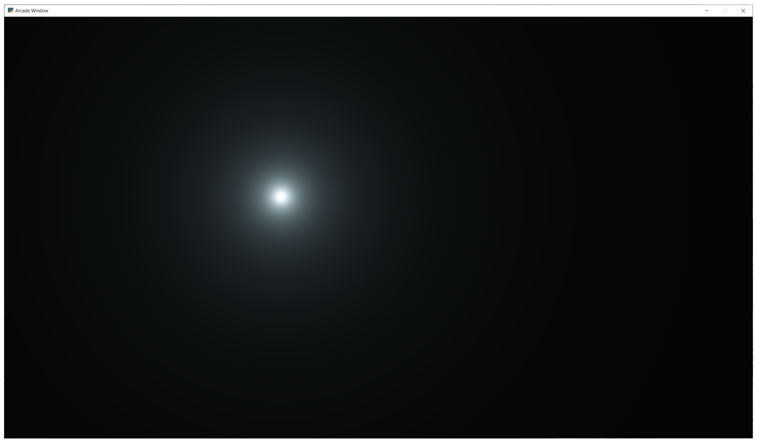
(continued from previous page)

```
        self.shadertoy.render()

if __name__ == "__main__":
    MyGame()
    arcade.run()
```

Then we can use those uniforms in our shader:

Listing 33: Glow follows mouse, and color can be changed.

```
uniform vec2 pos;
uniform vec3 color;

void mainImage(out vec4 fragColor, in vec2 fragCoord) {

    // Normalized pixel coordinates (from 0 to 1)
    vec2 uv = fragCoord/iResolution.xy;
    vec2 npos = pos/iResolution.xy;

    // Position of fragment relative to specified position
    vec2 rpos = npos - uv;
    // Adjust y by aspect ratio
    rpos.y /= iResolution.x/iResolution.y;

    // How far is the current pixel from the origin (0, 0)
    float distance = length(rpos);
    // Use an inverse 1/distance to set the fade
    float scale = 0.02;
    float fade = 1.1;
    float strength = pow(1.0 / distance * scale, fade);

    // Fade our orange color
    vec3 color = strength * color;

    // Tone mapping
    color = 1.0 - exp( -color );

    // Output to the screen
    fragColor = vec4(color, 1.0);
}
```



Note: Built-in Uniforms

Shadertoy assumes some built-in values. These can be set during the `Shadertoy.render()` call. In this example I'm not using those variables because I want to show how to send any value, not just built-in ones. The built-in values:

Python Variable	GLSL Variable
time	iTime
time_delta	iTimeDelta
mouse_position	iMouse
size	This is set by Shadertoy.resize()
frame	iFrame

An example of how they are set:

```
my_shader.render(time=self.time, mouse_position=mouse_position)
```

When resizing a window, make sure to always resize the shader as well.

15.3.10 Other examples

Here's another Python program that loads a GLSL file and displays it:

Listing 34: Shader Toy Demo

```python
import arcade
from arcade.experimental import Shadertoy


class MyGame(arcade.Window):

    def __init__(self):
        # Call the parent constructor
        super().__init__(width=1920, height=1080, title="Shader Demo", resizable=True)

        # Keep track of total run-time
        self.time = 0.0
```

(continues on next page)

(continued from previous page)

```python

        # File name of GLSL code
        # file_name = "fractal_pyramid.glsl"
        # file_name = "cyber_fuji_2020.glsl"
        file_name = "earth_planet_sky.glsl"
        # file_name = "flame.glsl"
        # file_name = "star_nest.glsl"

        # Create a shader from it
        self.shadertoy = Shadertoy(size=self.get_size(),
                                   main_source=open(file_name).read())

    def on_draw(self):
        self.clear()
        mouse_pos = self.mouse["x"], self.mouse["y"]
        self.shadertoy.render(time=self.time, mouse_position=mouse_pos)

    def on_update(self, dt):
        # Keep track of elapsed time
        self.time += dt


if __name__ == "__main__":
    MyGame()
    arcade.run()
```

You can use this demo with any of the sample code below. Click on the caption below the example shaders here to see the source code for the shader.

Some other sample shaders:

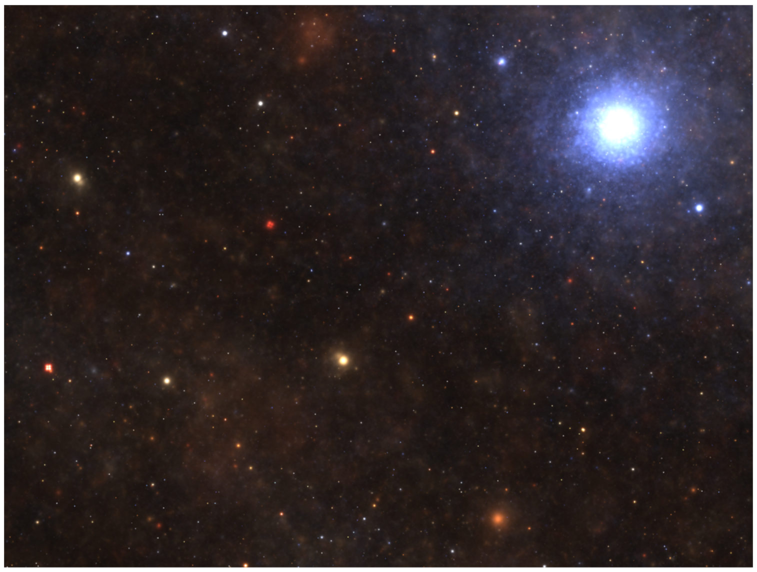


Fig. 4: star_nest

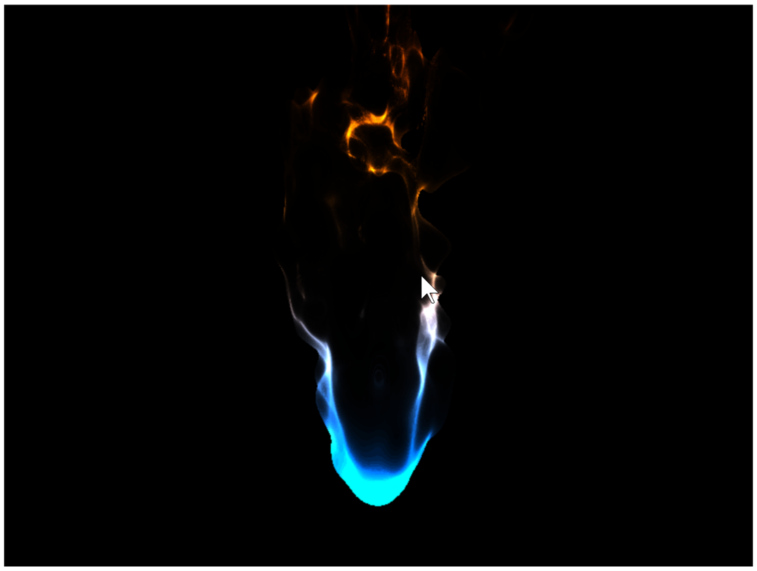


Fig. 5: flame

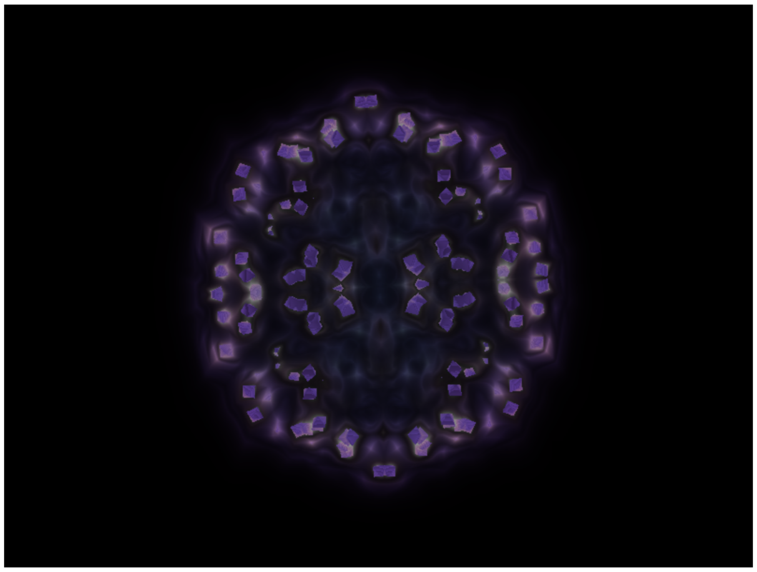


Fig. 6: fractal_pyramid

15.3.11 Additional learning

On this site:

- Learn a method of creating particles in *Shader Toy - Particles*.
- Learn how to ray-cast shadows in the *Ray-casting Shadows*.
- Make your screen look like an 80s monitor in *CRT Filter*.
- Read more about using OpenGL in Arcade with *OpenGL*.
- Learn to do a compute shader in *Compute Shader*.

On other sites:

- Here is a decent learn-by-example tutorial for making shaders: https://www.shadertoy.com/view/Md23DV
- Here's a video tutorial that steps through how to do an explosion: https://www.youtube.com/watch?v=xDxAnguEOn8

15.4 Shader Toy - Particles

Contents

This tutorial assumes you are already familiar with the material in *Shader Toy - Glow*. In this tutorial, we take a look at adding animated particles. These particles can be used for an explosion effect.

The "trick" to this example, is the use of pseudo-random numbers to generate each particle's angle and speed from the initial explosion point. Why "pseudo-random"? This allows each processor on the GPU to independently calculate each particle's position at any point and time. We can then allow the GPU to calculate in parallel.

15.4.1 Load the shader

First, we need a program that will load a shader. This program is also keeping track of how much time has elapsed. This is necessary for us to calculate how far along the animation sequence we are.

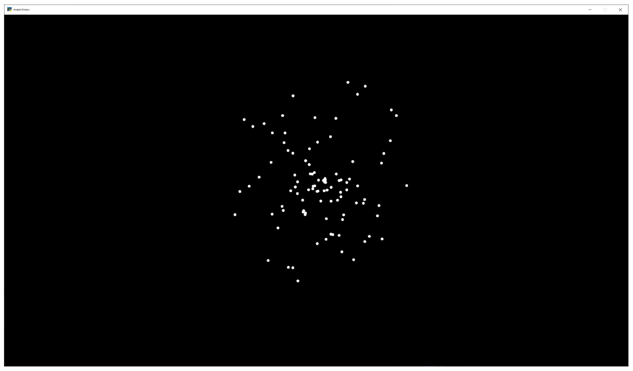
```python
import arcade
from arcade.experimental import Shadertoy


# Derive an application window from Arcade's parent Window class
class MyGame(arcade.Window):
```

(continues on next page)

(continued from previous page)

```python

    def __init__(self):
        # Call the parent constructor
        super().__init__(width=1920, height=1080)

        # Used to track run-time
        self.time = 0.0

        # Load a file and create a shader from it
        file_name = "explosion.glsl"
        self.shadertoy = Shadertoy(size=self.get_size(),
                                   main_source=open(file_name).read())

    def on_draw(self):
        self.clear()
        # Set uniform data to send to the GLSL shader
        self.shadertoy.program['pos'] = self.mouse["x"], self.mouse["y"]

        # Run the GLSL code
        self.shadertoy.render(time=self.time)

    def on_update(self, delta_time: float):
        # Track run time
        self.time += delta_time


if __name__ == "__main__":
    window = MyGame()
    window.center_window()
    arcade.run()
```

15.4.2 Initial shader with particles



```glsl
// Origin of the particles
uniform vec2 pos;

// Constants

```

(continues on next page)

(continued from previous page)

```
// Number of particles
const float PARTICLE_COUNT = 100.0;
// Max distance the particle can be from the position.
// Normalized. (So, 0.3 is 30% of the screen.)
const float MAX_PARTICLE_DISTANCE = 0.3;
// Size of each particle. Normalized.
const float PARTICLE_SIZE = 0.004;
const float TWOPI = 6.2832;

// This function will return two pseudo-random numbers given an input seed.
// The result is in polar coordinates, to make the points random in a circle
// rather than a rectangle.
vec2 Hash12_Polar(float t) {
  float angle = fract(sin(t * 674.3) * 453.2) * TWOPI;
  float distance = fract(sin((t + angle) * 724.3) * 341.2);
  return vec2(sin(angle), cos(angle)) * distance;
}

void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    // Normalized pixel coordinates (from 0 to 1)
    // Origin of the particles
    vec2 npos = (pos - .5 * iResolution.xy) / iResolution.y;
    // Position of current pixel we are drawing
    vec2 uv = (fragCoord- .5 * iResolution.xy) / iResolution.y;

    // Re-center based on input coordinates, rather than origin.
    uv -= npos;

    // Default alpha is transparent.
    float alpha = 0.0;

    // Loop for each particle
    for (float i= 0.; i < PARTICLE_COUNT; i++) {
        // Direction of particle + speed
        float seed = i + 1.0;
        vec2 dir = Hash12_Polar(seed);
        // Get position based on direction, magnitude, and explosion size
        vec2 particlePosition = dir * MAX_PARTICLE_DISTANCE;
        // Distance of this pixel from that particle
        float d = length(uv - particlePosition);
        // If we are within the particle size, set alpha to 1.0
        if (d < PARTICLE_SIZE)
            alpha = 1.0;
    }
    // Output to screen
    fragColor = vec4(1.0, 1.0, 1.0, alpha);
}
```

15.4.3 Add particle movement

```
// Origin of the particles
uniform vec2 pos;

// Constants

// Number of particles
const float PARTICLE_COUNT = 100.0;
// Max distance the particle can be from the position.
// Normalized. (So, 0.3 is 30% of the screen.)
const float MAX_PARTICLE_DISTANCE = 0.3;
// Size of each particle. Normalized.
const float PARTICLE_SIZE = 0.004;
// Time for each burst cycle, in seconds.
const float BURST_TIME = 2.0;
const float TWOPI = 6.2832;

// This function will return two pseudo-random numbers given an input seed.
// The result is in polar coordinates, to make the points random in a circle
// rather than a rectangle.
vec2 Hash12_Polar(float t) {
  float angle = fract(sin(t * 674.3) * 453.2) * TWOPI;
  float distance = fract(sin((t + angle) * 724.3) * 341.2);
  return vec2(sin(angle), cos(angle)) * distance;
}

void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    // Normalized pixel coordinates (from 0 to 1)
    // Origin of the particles
    vec2 npos = (pos - .5 * iResolution.xy) / iResolution.y;
    // Position of current pixel we are drawing
    vec2 uv = (fragCoord- .5 * iResolution.xy) / iResolution.y;

    // Re-center based on input coordinates, rather than origin.
    uv -= npos;

    // Default alpha is transparent.
    float alpha = 0.0;

    // 0.0 - 1.0 normalized fraction representing how far along in the explosion we are.
    // Auto resets if time goes beyond burst time. This causes the explosion to cycle.
    float timeFract = fract(iTime * 1 / BURST_TIME);

    // Loop for each particle
    for (float i= 0.; i < PARTICLE_COUNT; i++) {
        // Direction of particle + speed
        float seed = i + 1.0;
        vec2 dir = Hash12_Polar(seed);
        // Get position based on direction, magnitude, and explosion size
        // Adjust based on time scale. (0.0-1.0)
```

(continues on next page)

(continued from previous page)

```
        vec2 particlePosition = dir * MAX_PARTICLE_DISTANCE * timeFract;
        // Distance of this pixel from that particle
        float d = length(uv - particlePosition);
        // If we are within the particle size, set alpha to 1.0
        if (d < PARTICLE_SIZE)
            alpha = 1.0;
    }
    // Output to screen
    fragColor = vec4(1.0, 1.0, 1.0, alpha);
}
```

15.4.4 Fade-out

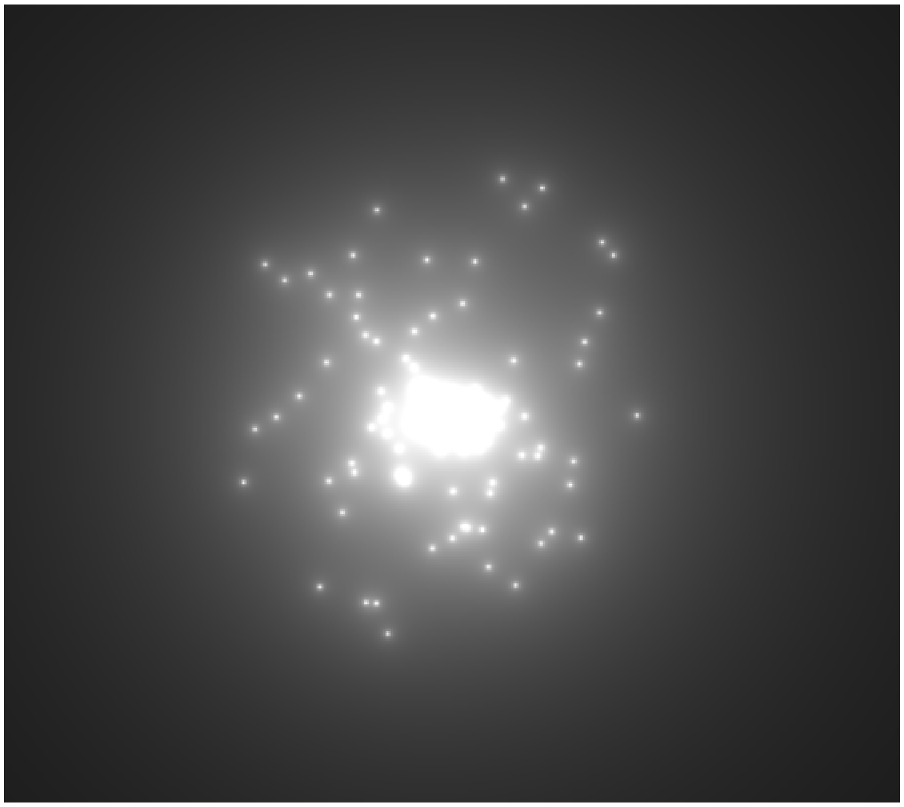
```
// Origin of the particles
uniform vec2 pos;

// Constants

// Number of particles
const float PARTICLE_COUNT = 100.0;
// Max distance the particle can be from the position.
// Normalized. (So, 0.3 is 30% of the screen.)
const float MAX_PARTICLE_DISTANCE = 0.3;
// Size of each particle. Normalized.
const float PARTICLE_SIZE = 0.004;
// Time for each burst cycle, in seconds.
const float BURST_TIME = 2.0;
const float TWOPI = 6.2832;

// This function will return two pseudo-random numbers given an input seed.
// The result is in polar coordinates, to make the points random in a circle
// rather than a rectangle.
vec2 Hash12_Polar(float t) {
  float angle = fract(sin(t * 674.3) * 453.2) * TWOPI;
  float distance = fract(sin((t + angle) * 724.3) * 341.2);
  return vec2(sin(angle), cos(angle)) * distance;
}

void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    // Normalized pixel coordinates (from 0 to 1)
    // Origin of the particles
    vec2 npos = (pos - .5 * iResolution.xy) / iResolution.y;
    // Position of current pixel we are drawing
    vec2 uv = (fragCoord- .5 * iResolution.xy) / iResolution.y;

    // Re-center based on input coordinates, rather than origin.
    uv -= npos;

    // Default alpha is transparent.
```

(continues on next page)

(continued from previous page)

```
    float alpha = 0.0;

    // 0.0 - 1.0 normalized fraction representing how far along in the explosion we are.
    // Auto resets if time goes beyond burst time. This causes the explosion to cycle.
    float timeFract = fract(iTime * 1 / BURST_TIME);

    // Loop for each particle
    for (float i= 0.; i < PARTICLE_COUNT; i++) {
        // Direction of particle + speed
        float seed = i + 1.0;
        vec2 dir = Hash12_Polar(seed);
        // Get position based on direction, magnitude, and explosion size
        // Adjust based on time scale. (0.0-1.0)
        vec2 particlePosition = dir * MAX_PARTICLE_DISTANCE * timeFract;
        // Distance of this pixel from that particle
        float d = length(uv - particlePosition);
        // If we are within the particle size, set alpha to 1.0
        if (d < PARTICLE_SIZE)
            alpha = 1.0;
    }
    // Output to screen
    fragColor = vec4(1.0, 1.0, 1.0, alpha * (1.0 - timeFract));
}
```

15.4.5 Glowing Particles



```
// Origin of the particles
uniform vec2 pos;

// Constants

// Number of particles
const float PARTICLE_COUNT = 100.0;
// Max distance the particle can be from the position.
// Normalized. (So, 0.3 is 30% of the screen.)
const float MAX_PARTICLE_DISTANCE = 0.3;
// Size of each particle. Normalized.
const float PARTICLE_SIZE = 0.004;
// Time for each burst cycle, in seconds.
const float BURST_TIME = 2.0;
// Particle brightness
const float DEFAULT_BRIGHTNESS = 0.0005;

const float TWOPI = 6.2832;

// This function will return two pseudo-random numbers given an input seed.
// The result is in polar coordinates, to make the points random in a circle
// rather than a rectangle.
vec2 Hash12_Polar(float t) {
  float angle = fract(sin(t * 674.3) * 453.2) * TWOPI;
  float distance = fract(sin((t + angle) * 724.3) * 341.2);
  return vec2(sin(angle), cos(angle)) * distance;
}

void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    // Normalized pixel coordinates (from 0 to 1)
    // Origin of the particles
    vec2 npos = (pos - .5 * iResolution.xy) / iResolution.y;
    // Position of current pixel we are drawing
    vec2 uv = (fragCoord- .5 * iResolution.xy) / iResolution.y;

    // Re-center based on input coordinates, rather than origin.
    uv -= npos;

    // Default alpha is transparent.
    float alpha = 0.0;

    // 0.0 - 1.0 normalized fraction representing how far along in the explosion we are.
    // Auto resets if time goes beyond burst time. This causes the explosion to cycle.
    float timeFract = fract(iTime * 1 / BURST_TIME);

    // Loop for each particle
    for (float i= 0.; i < PARTICLE_COUNT; i++) {
        // Direction of particle + speed
        float seed = i + 1.0;
        vec2 dir = Hash12_Polar(seed);
        // Get position based on direction, magnitude, and explosion size
        // Adjust based on time scale. (0.0-1.0)
```

(continues on next page)

(continued from previous page)

```
        vec2 particlePosition = dir * MAX_PARTICLE_DISTANCE * timeFract;
        // Distance of this pixel from that particle
        float d = length(uv - particlePosition);
        // Add glow based on distance
        alpha += DEFAULT_BRIGHTNESS / d;
    }
    // Output to screen
    fragColor = vec4(1.0, 1.0, 1.0, alpha * (1.0 - timeFract));
}
```

15.4.6 Twinkling Particles

```
// Origin of the particles
uniform vec2 pos;

// Constants

// Number of particles
const float PARTICLE_COUNT = 100.0;
// Max distance the particle can be from the position.
// Normalized. (So, 0.3 is 30% of the screen.)
const float MAX_PARTICLE_DISTANCE = 0.3;
// Size of each particle. Normalized.
const float PARTICLE_SIZE = 0.004;
// Time for each burst cycle, in seconds.
const float BURST_TIME = 2.0;
// Particle brightness
const float DEFAULT_BRIGHTNESS = 0.0005;
// How many times to the particles twinkle
const float TWINKLE_SPEED = 10.0;

const float TWOPI = 6.2832;

// This function will return two pseudo-random numbers given an input seed.
// The result is in polar coordinates, to make the points random in a circle
// rather than a rectangle.
vec2 Hash12_Polar(float t) {
  float angle = fract(sin(t * 674.3) * 453.2) * TWOPI;
  float distance = fract(sin((t + angle) * 724.3) * 341.2);
  return vec2(sin(angle), cos(angle)) * distance;
}

void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    // Normalized pixel coordinates (from 0 to 1)
    // Origin of the particles
    vec2 npos = (pos - .5 * iResolution.xy) / iResolution.y;
    // Position of current pixel we are drawing
    vec2 uv = (fragCoord- .5 * iResolution.xy) / iResolution.y;

```

(continues on next page)

(continued from previous page)

```
    // Re-center based on input coordinates, rather than origin.
    uv -= npos;

    // Default alpha is transparent.
    float alpha = 0.0;

    // 0.0 - 1.0 normalized fraction representing how far along in the explosion we are.
    // Auto resets if time goes beyond burst time. This causes the explosion to cycle.
    float timeFract = fract(iTime * 1 / BURST_TIME);

    // Loop for each particle
    for (float i= 0.; i < PARTICLE_COUNT; i++) {
        // Direction of particle + speed
        float seed = i + 1.0;
        vec2 dir = Hash12_Polar(seed);
        // Get position based on direction, magnitude, and explosion size
        // Adjust based on time scale. (0.0-1.0)
        vec2 particlePosition = dir * MAX_PARTICLE_DISTANCE * timeFract;
        // Distance of this pixel from that particle
        float d = length(uv - particlePosition);
        // Add glow based on distance
        float brightness = DEFAULT_BRIGHTNESS * (sin(timeFract * TWINKLE_SPEED + i) * .5␣
↪+ .5);
        alpha += brightness / d;
    }
    // Output to screen
    fragColor = vec4(1.0, 1.0, 1.0, alpha * (1.0 - timeFract));
}
```

15.5 Compute Shader

For certain types of calculations, compute shaders on the GPU can be thousands of times faster than on the CPU alone.

In this tutorial, we will simulate a star field using an ‘N-Body simulation’. Each star is affected by the gravity of every other star. For 1,000 stars, this means we have 1,000 x 1,000 = 1,000,000 million calculations to perform for each frame. The video has 65,000 stars, requiring 4.2 billion gravity force calculations per frame. On high-end hardware it can still run at 60 fps!

How does this work? There are three major parts to this program:

- The Python code, which allocates buffers & glues everything together
- The visualization shaders, which let us see the data in the buffers
- The compute shader, which moves everything

15.5.1 Buffers

We need a place to store the data we'll visualize. To do so, we'll create two **Shader Storage Buffer Objects** (SSBOs) of floating point numbers from within our Python code. One will hold the previous frame's star positions, and the other will be used to store calculate the next frame's positions.

Each buffer must be able to store the following for each star:

1. The x, y, and radius of each star stored
2. The velocity of the star, which will be unused by the visualization
3. The floating point RGBA color of the star

Generating Aligned Data

To avoid issues with GPU memory alignment quirks, we'll use the function below to generate well-aligned data ready to load into the SSBO. The docstrings & comments explain why in greater detail:

Listing 35: Generating Well-Aligned Data to Load onto the GPU

```python
def gen_initial_data(
        screen_size: Tuple[int, int],
        num_stars: int = NUM_STARS,
        use_color: bool = False
) -> array:
    """
    Generate an :py:class:`~array.array` of randomly positioned star data.

    Some of this data is wasted as padding because:

    1. GPUs expect SSBO data to be aligned to multiples of 4
    2. GLSL's vec3 is actually a vec4 with compiler-side restrictions,
       so we have to use 4-length vectors anyway.

    :param screen_size: A (width, height) of the area to generate stars in
    :param num_stars: How many stars to generate
    :param use_color: Whether to generate white or randomized pastel stars
    :return: an array of star position data
    """
    width, height = screen_size
    color_channel_min = 0.5 if use_color else 1.0

    def _data_generator() -> Generator[float, None, None]:
        """Inner generator function used to illustrate memory layout"""

        for i in range(num_stars):
            # Position/radius
            yield random.randrange(0, width)
            yield random.randrange(0, height)
            yield 0.0  # z (padding, unused by shaders)
            yield 6.0

            # Velocity (unused by visualization shaders)
            yield 0.0
```

(continues on next page)

(continued from previous page)

```
            yield 0.0
            yield 0.0  # vz (padding, unused by shaders)
            yield 0.0  # vw (padding, unused by shaders)

            # Color
            yield random.uniform(color_channel_min, 1.0)  # r
            yield random.uniform(color_channel_min, 1.0)  # g
            yield random.uniform(color_channel_min, 1.0)  # b
            yield 1.0  # a

    # Use the generator function to fill an array in RAM
    return array('f', _data_generator())
```

Allocating the Buffers

Listing 36: Allocating the Buffers & Loading the Data onto the GPU

```
        # --- Create buffers

        # Create pairs of buffers for the compute & visualization shaders.
        # We will swap which buffer instance is the initial value and
        # which is used as the current value to write to.

        # ssbo = shader storage buffer object
        initial_data = gen_initial_data(self.get_size(), use_color=USE_COLORED_STARS)
        self.ssbo_previous = self.ctx.buffer(data=initial_data)
        self.ssbo_current = self.ctx.buffer(data=initial_data)

        # vao = vertex array object
        # Format string describing how to interpret the SSBO buffer data.
        # 4f = position and size -> x, y, z, radius
        # 4x4 = Four floats used for calculating velocity. Not needed for visualization.
        # 4f = color -> rgba
        buffer_format = "4f 4x4 4f"

        # Attribute variable names for the vertex shader
        attributes = ["in_vertex", "in_color"]

        self.vao_previous = self.ctx.geometry(
            [BufferDescription(self.ssbo_previous, buffer_format, attributes)],
            mode=self.ctx.POINTS,
        )
        self.vao_current = self.ctx.geometry(
            [BufferDescription(self.ssbo_current, buffer_format, attributes)],
            mode=self.ctx.POINTS,
        )
```

15.5.2 Visualization Shaders

Now that we have the data, we need to be able to visualize it. We'll do it by applying vertex, geometry, and fragment shaders to convert the data in the SSBO into pixels. For each star's 12 floats in the array, the following flow of data will take place:

Vertex Shader

In this tutorial, the vertex shader will be run for each star's 12 float long stretch of raw padded data in `self.ssbo_current`. Each execution will output clean typed data to an instance of the geometry shader.

Data is read in as follows:

- The x, y, and radius of each star are accessed via `in_vertex`
- The floating point RGBA color of the star, via `in_color`

Listing 37: shaders/vertex_shader.glsl

```glsl
#version 330

in vec4 in_vertex;
in vec4 in_color;

out vec2 vertex_pos;
out float vertex_radius;
out vec4 vertex_color;

void main()
{
    vertex_pos = in_vertex.xy;
    vertex_radius = in_vertex.w;
    vertex_color = in_color;
}
```

The variables below are then passed as inputs to the geometry shader:

- `vertex_pos`
- `vertex_radius`
- `vertex_color`

Geometry Shader

The **geometry shader** converts a single point into a quad, in this case a square, which the GPU can render. It does this by emitting four points centered on the input point.

Listing 38: shaders/geometry_shader.glsl

```glsl
#version 330

layout (points) in;
layout (triangle_strip, max_vertices = 4) out;

```

(continues on next page)

(continued from previous page)

```
// Use arcade's global projection UBO
uniform Projection {
    uniform mat4 matrix;
} proj;


// The outputs from the vertex shader are used as inputs
in vec2 vertex_pos[];
in float vertex_radius[];
in vec4 vertex_color[];

// These are used with EmitVertex to generate four points of
// a quad centered around vertex_pos[0].
out vec2 g_uv;
out vec3 g_color;

void main() {
    vec2 center = vertex_pos[0];
    vec2 hsize = vec2(vertex_radius[0]);

    g_color = vertex_color[0].rgb;

    gl_Position = proj.matrix * vec4(vec2(-hsize.x, hsize.y) + center, 0.0, 1.0);
    g_uv = vec2(0, 1);
    EmitVertex();

    gl_Position = proj.matrix * vec4(vec2(-hsize.x, -hsize.y) + center, 0.0, 1.0);
    g_uv = vec2(0, 0);
    EmitVertex();

    gl_Position = proj.matrix * vec4(vec2(hsize.x, hsize.y) + center, 0.0, 1.0);
    g_uv = vec2(1, 1);
    EmitVertex();

    gl_Position = proj.matrix * vec4(vec2(hsize.x, -hsize.y) + center, 0.0, 1.0);
    g_uv = vec2(1, 0);
    EmitVertex();

    // End geometry emmission
    EndPrimitive();
}
```

Fragment Shader

A **fragment shader** runs for each pixel in a quad. It converts a UV coordinate within the quad to a float RGBA value. In this tutorial's case, the shader produces the soft glowing circle on the surface of each star's quad.

Listing 39: shaders/fragment_shader.glsl

```glsl
#version 330

in vec2 g_uv;
in vec3 g_color;

out vec4 out_color;

void main()
{
    float l = length(vec2(0.5, 0.5) - g_uv.xy);
    if ( l > 0.5)
    {
        discard;
    }
    float alpha;
    if (l == 0.0)
        alpha = 1.0;
    else
        alpha = min(1.0, .60-l * 2);

    vec3 c = g_color.rgb;
    // c.xy += v_uv.xy * 0.05;
    // c.xy += v_pos.xy * 0.75;
    out_color = vec4(c, alpha);
}
```

15.5.3 Compute Shader

Now that we have a way to display data, we should update it.

We created pairs of buffers earlier. We will use one SSBO as an **input buffer** holding the previous frame's data, and another as our **output** buffer to write results to.

We then swap our buffers each frame after drawing, using the output as the input of the next frame, and repeat the process until the program stops running.

Listing 40: shaders/compute_shader.glsl

```glsl
#version 430

// Set up our compute groups.
// The COMPUTE_SIZE_X and COMPUTE_SIZE_Y values will be replaced
// by the Python code with actual values. This does not happen
// automatically, and must be called manually.
layout(local_size_x=COMPUTE_SIZE_X, local_size_y=COMPUTE_SIZE_Y) in;

// Input uniforms would go here if you need them.
```

(continues on next page)

(continued from previous page)

```
// The examples below match the ones commented out in main.py
//uniform vec2 screen_size;
//uniform float frame_time;

// Structure of the star data
struct Star
{
    vec4 pos;
    vec4 vel;
    vec4 color;
};

// Input buffer
layout(std430, binding=0) buffer stars_in
{
    Star stars[];
} In;

// Output buffer
layout(std430, binding=1) buffer stars_out
{
    Star stars[];
} Out;

void main()
{
    int curStarIndex = int(gl_GlobalInvocationID);

    Star in_star = In.stars[curStarIndex];

    vec4 p = in_star.pos.xyzw;
    vec4 v = in_star.vel.xyzw;

    // Move the star according to the current force
    p.xy += v.xy;

    // Calculate the new force based on all the other bodies
    for (int i=0; i < In.stars.length(); i++) {
        // If enabled, this will keep the star from calculating gravity on itself
        // However, it does slow down the calcluations do do this check.
        //  if (i == x)
        //      continue;

        // Calculate distance squared
        float dist = distance(In.stars[i].pos.xyzw.xy, p.xy);
        float distanceSquared = dist * dist;

        // If distance is too small, extremely high forces can result and
        // fling the star into escape velocity and forever off the screen.
        // Using a reasonable minimum distance to prevents this.
        float minDistance = 0.02;
        float gravityStrength = 0.3;
```

(continues on next page)

(continued from previous page)

```
        float simulationSpeed = 0.002;
        float force = min(minDistance, gravityStrength / distanceSquared) * -
↪simulationSpeed;

        vec2 diff = p.xy - In.stars[i].pos.xyzw.xy;
        // We should normalize this I think, but it doesn't work.
        //  diff = normalize(diff);
        vec2 delta_v = diff * force;
        v.xy += delta_v;
    }


    Star out_star;
    out_star.pos.xyzw = p.xyzw;
    out_star.vel.xyzw = v.xyzw;

    vec4 c = in_star.color.xyzw;
    out_star.color.xyzw = c.xyzw;

    Out.stars[curStarIndex] = out_star;
}
```

15.5.4 The Finished Python Program

The code includes thorough docstrings and annotations explaining how it works.

Listing 41: main.py

```
"""
N-Body Gravity with Compute Shaders & Buffers
"""
import random
from array import array
from pathlib import Path
from typing import Generator, Tuple

import arcade
from arcade.gl import BufferDescription

# Window dimensions in pixels
WINDOW_WIDTH = 800
WINDOW_HEIGHT = 600

# Size of performance graphs in pixels
GRAPH_WIDTH = 200
GRAPH_HEIGHT = 120
GRAPH_MARGIN = 5

NUM_STARS: int = 4000
USE_COLORED_STARS: bool = True

```

(continues on next page)

(continued from previous page)

```python

def gen_initial_data(
        screen_size: Tuple[int, int],
        num_stars: int = NUM_STARS,
        use_color: bool = False
) -> array:
    """
    Generate an :py:class:`~array.array` of randomly positioned star data.

    Some of this data is wasted as padding because:

    1. GPUs expect SSBO data to be aligned to multiples of 4
    2. GLSL's vec3 is actually a vec4 with compiler-side restrictions,
       so we have to use 4-length vectors anyway.

    :param screen_size: A (width, height) of the area to generate stars in
    :param num_stars: How many stars to generate
    :param use_color: Whether to generate white or randomized pastel stars
    :return: an array of star position data
    """
    width, height = screen_size
    color_channel_min = 0.5 if use_color else 1.0

    def _data_generator() -> Generator[float, None, None]:
        """Inner generator function used to illustrate memory layout"""

        for i in range(num_stars):
            # Position/radius
            yield random.randrange(0, width)
            yield random.randrange(0, height)
            yield 0.0  # z (padding, unused by shaders)
            yield 6.0

            # Velocity (unused by visualization shaders)
            yield 0.0
            yield 0.0
            yield 0.0  # vz (padding, unused by shaders)
            yield 0.0  # vw (padding, unused by shaders)

            # Color
            yield random.uniform(color_channel_min, 1.0)  # r
            yield random.uniform(color_channel_min, 1.0)  # g
            yield random.uniform(color_channel_min, 1.0)  # b
            yield 1.0  # a

    # Use the generator function to fill an array in RAM
    return array('f', _data_generator())


class NBodyGravityWindow(arcade.Window):

    def __init__(self):
```

(continues on next page)

(continued from previous page)

```
        # Ask for OpenGL context supporting version 4.3 or greater when
        # calling the parent initializer to make sure we have compute shader
        # support.
        super().__init__(
            WINDOW_WIDTH, WINDOW_HEIGHT,
            "N-Body Gravity with Compute Shaders & Buffers",
            gl_version=(4, 3),
            resizable=False
        )
        # Attempt to put the window in the center of the screen.
        self.center_window()

        # --- Create buffers

        # Create pairs of buffers for the compute & visualization shaders.
        # We will swap which buffer instance is the initial value and
        # which is used as the current value to write to.

        # ssbo = shader storage buffer object
        initial_data = gen_initial_data(self.get_size(), use_color=USE_COLORED_STARS)
        self.ssbo_previous = self.ctx.buffer(data=initial_data)
        self.ssbo_current = self.ctx.buffer(data=initial_data)

        # vao = vertex array object
        # Format string describing how to interpret the SSBO buffer data.
        # 4f = position and size -> x, y, z, radius
        # 4x4 = Four floats used for calculating velocity. Not needed for visualization.
        # 4f = color -> rgba
        buffer_format = "4f 4x4 4f"

        # Attribute variable names for the vertex shader
        attributes = ["in_vertex", "in_color"]

        self.vao_previous = self.ctx.geometry(
            [BufferDescription(self.ssbo_previous, buffer_format, attributes)],
            mode=self.ctx.POINTS,
        )
        self.vao_current = self.ctx.geometry(
            [BufferDescription(self.ssbo_current, buffer_format, attributes)],
            mode=self.ctx.POINTS,
        )

        # --- Create the visualization shaders

        vertex_shader_source = Path("shaders/vertex_shader.glsl").read_text()
        fragment_shader_source = Path("shaders/fragment_shader.glsl").read_text()
        geometry_shader_source = Path("shaders/geometry_shader.glsl").read_text()

        # Create the complete shader program which will draw the stars
        self.program = self.ctx.program(
            vertex_shader=vertex_shader_source,
            geometry_shader=geometry_shader_source,
```

(continues on next page)

(continued from previous page)

```python
            fragment_shader=fragment_shader_source,
        )

        # --- Create our compute shader

        # Load in the raw source code safely & auto-close the file
        compute_shader_source = Path("shaders/compute_shader.glsl").read_text()

        # Compute shaders use groups to parallelize execution.
        # You don't need to understand how this works yet, but the
        # values below should serve as reasonable defaults. Later, we'll
        # preprocess the shader source by replacing the templating token
        # with its corresponding value.
        self.group_x = 256
        self.group_y = 1

        self.compute_shader_defines = {
            "COMPUTE_SIZE_X": self.group_x,
            "COMPUTE_SIZE_Y": self.group_y
        }

        # Preprocess the source by replacing each define with its value as a string
        for templating_token, value in self.compute_shader_defines.items():
            compute_shader_source = compute_shader_source.replace(templating_token,␣
↪str(value))

        self.compute_shader = self.ctx.compute_shader(source=compute_shader_source)

        # --- Create the FPS graph

        # Enable timings for the performance graph
        arcade.enable_timings()

        # Create a sprite list to put the performance graph into
        self.perf_graph_list = arcade.SpriteList()

        # Create the FPS performance graph
        graph = arcade.PerfGraph(GRAPH_WIDTH, GRAPH_HEIGHT, graph_data="FPS")
        graph.position = GRAPH_WIDTH / 2, self.height - GRAPH_HEIGHT / 2
        self.perf_graph_list.append(graph)

    def on_draw(self):
        # Clear the screen
        self.clear()
        # Enable blending so our alpha channel works
        self.ctx.enable(self.ctx.BLEND)

        # Bind buffers
        self.ssbo_previous.bind_to_storage_buffer(binding=0)
        self.ssbo_current.bind_to_storage_buffer(binding=1)

        # If you wanted, you could set input variables for compute shader
```

(continues on next page)

(continued from previous page)

```
        # as in the lines commented out below. You would have to add or
        # uncomment corresponding lines in compute_shader.glsl
        # self.compute_shader["screen_size"] = self.get_size()
        # self.compute_shader["frame_time"] = self.frame_time

        # Run compute shader to calculate new positions for this frame
        self.compute_shader.run(group_x=self.group_x, group_y=self.group_y)

        # Draw the current star positions
        self.vao_current.render(self.program)

        # Swap the buffer pairs.
        # The buffers for the current state become the initial state,
        # and the data of this frame's initial state will be overwritten.
        self.ssbo_previous, self.ssbo_current = self.ssbo_current, self.ssbo_previous
        self.vao_previous, self.vao_current = self.vao_current, self.vao_previous

        # Draw the graphs
        self.perf_graph_list.draw()



if __name__ == "__main__":
    app = NBodyGravityWindow()
    arcade.run()
```

An expanded version of this tutorial whith support for 3D is available at: https://github.com/pvcraven/n-body

15.6 GPU Particle Burst

In this example, we show how to create explosions using particles. The particles are tracked by the GPU, significantly improving the performance.

15.6.1 Step 1: Open a Blank Window

First, let's start with a blank window.

Listing 42: gpu_particle_burst_01.py

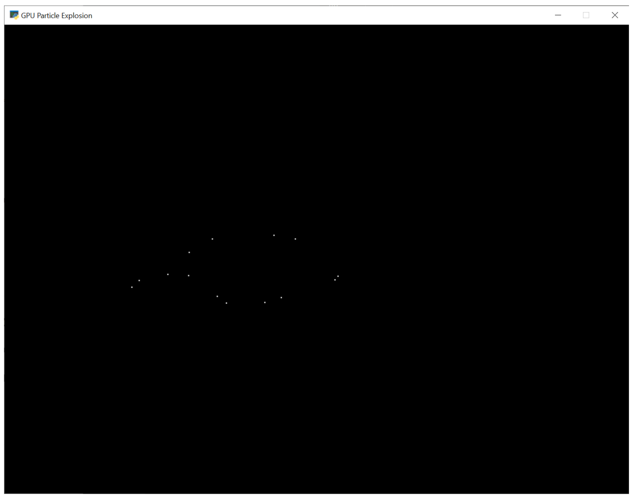
```
"""
Example showing how to create particle explosions via the GPU.
"""
import arcade

SCREEN_WIDTH = 1024
SCREEN_HEIGHT = 768
SCREEN_TITLE = "GPU Particle Explosion"


```

(continues on next page)

(continued from previous page)

```python
class MyWindow(arcade.Window):
    """ Main window"""
    def __init__(self):
        super().__init__(SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_TITLE)

    def on_draw(self):
        """ Draw everything """
        self.clear()

    def on_update(self, dt):
        """ Update everything """
        pass

    def on_mouse_press(self, x: float, y: float, button: int, modifiers: int):
        """ User clicks mouse """
        pass


if __name__ == "__main__":
    window = MyWindow()
    window.center_window()
    arcade.run()
```

15.6.2 Step 2: Create One Particle For Each Click



For this next section, we are going to draw a dot each time the user clicks their mouse on the screen.

For each click, we are going to create an instance of a `Burst` class that will eventually be turned into a full explosion. Each burst instance will be added to a list.

Imports

First, we'll import some more items for our program:

```python
from array import array
from dataclasses import dataclass

import arcade
import arcade.gl
```

Burst Dataclass

Next, we'll create a dataclass to track our data for each burst. For each burst we need to track a Vertex Array Object (VAO) which stores information about our burst. Inside of that, we'll have a Vertex Buffer Object (VBO) which will be a high-speed memory buffer where we'll store locations, colors, velocity, etc.

```python
@dataclass
class Burst:
    """ Track for each burst. """
    buffer: arcade.gl.Buffer
    vao: arcade.gl.Geometry
```

Init method

Next, we'll create an empty list attribute called `burst_list`. We'll also create our OpenGL shader program. The program will be a collection of two shader programs. These will be stored in separate files, saved in the same directory.

Note: In addition to loading the program via the *load_program()* method of *ArcadeContext* shown, it is also possible to keep the GLSL programs in triple- quoted string by using *program()* of *Context*.

Listing 43: MyWindow.__init__

```
    def __init__(self):
        super().__init__(SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_TITLE)
        self.burst_list = []

        # Program to visualize the points
        self.program = self.ctx.load_program(
            vertex_shader="vertex_shader_v1.glsl",
            fragment_shader="fragment_shader.glsl",
        )

        self.ctx.enable_only()
```

OpenGL Shaders

The OpenGL Shading Language (GLSL) is C-style language that runs on your graphics card (GPU) rather than your CPU. Unfortunately a full explanation of the language is beyond the scope of this tutorial. I hope, however, the tutorial can get you started understanding how it works.

We'll have two shaders. A **vertex shader**, and a **fragment shader**. A vertex shader runs for each vertex point of the geometry we are rendering, and a fragment shader runs for each pixel. For example, vertex shader might run four times for each point on a rectangle, and the fragment shader would run for each pixel on the screen.

The vertex shader takes in the position of our vertex. We'll set `in_pos` in our Python program, and pass that data to this shader.

The vertex shader outputs the color of our vertex. Colors are in Red-Green-Blue-Alpha (RGBA) format, with floating-point numbers ranging from 0 to 1. In our program below case, we set the color to (1, 1, 1) which is white, and the fourth 1 for completely opaque.

Listing 44: vertex_shader_v1.glsl

```
#version 330

// (x, y) position passed in
in vec2 in_pos;

// Output the color to the fragment shader
out vec4 color;

void main() {

    // Set the RGBA color
    color = vec4(1, 1, 1, 1);

    // Set the position. (x, y, z, w)
    gl_Position = vec4(in_pos, 0.0, 1);
}
```

There's not much to the fragment shader, it just takes in `color` from the vertex shader and passes it back out as the pixel color. We'll use the same fragment shader for every version in this tutorial.

Listing 45: fragment_shader.glsl

```glsl
#version 330

// Color passed in from the vertex shader
in vec4 color;

// The pixel we are writing to in the framebuffer
out vec4 fragColor;

void main() {

    // Fill the point
    fragColor = vec4(color);
}
```

Mouse Pressed

Each time we press the mouse button, we are going to create a burst at that location.

The data for that burst will be stored in an instance of the `Burst` class.

The `Burst` class needs our data buffer. The data buffer contains information about each particle. In this case, we just have one particle and only need to store the x, y of that particle in the buffer. However, eventually we'll have hundreds of particles, each with a position, velocity, color, and fade rate. To accommodate creating that data, we have made a generator function `_gen_initial_data`. It is totally overkill at this point, but we'll add on to it in this tutorial.

The `buffer_description` says that each vertex has two floating data points (`2f`) and those data points will come into the shader with the reference name `in_pos` which we defined above in our *OpenGL Shaders*

Listing 46: MyWindow.on_mouse_press

```python
    def on_mouse_press(self, x: float, y: float, button: int, modifiers: int):
        """ User clicks mouse """

        def _gen_initial_data(initial_x, initial_y):
            """ Generate data for each particle """
            yield initial_x
            yield initial_y

        # Recalculate the coordinates from pixels to the OpenGL system with
        # 0, 0 at the center.
        x2 = x / self.width * 2. - 1.
        y2 = y / self.height * 2. - 1.

        # Get initial particle data
        initial_data = _gen_initial_data(x2, y2)

        # Create a buffer with that data
        buffer = self.ctx.buffer(data=array('f', initial_data))

        # Create a buffer description specifying the buffer's data format
        buffer_description = arcade.gl.BufferDescription(
```

(continues on next page)

(continued from previous page)

```
            buffer,
            '2f',
            ['in_pos'])

        # Create our Vertex Attribute Object
        vao = self.ctx.geometry([buffer_description])

        # Create the Burst object and add it to the list of bursts
        burst = Burst(buffer=buffer, vao=vao)
        self.burst_list.append(burst)
```

Drawing

Finally, draw it.

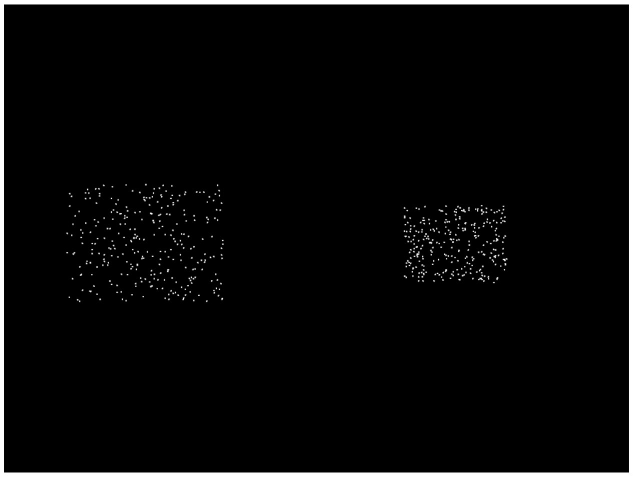
Listing 47: MyWindow.on_draw

```
    def on_draw(self):
        """ Draw everything """
        self.clear()

        # Set the particle size
        self.ctx.point_size = 2 * self.get_pixel_ratio()

        # Loop through each burst
        for burst in self.burst_list:

            # Render the burst
            burst.vao.render(self.program, mode=self.ctx.POINTS)
```

Program Listings

- fragment_shader ← Where we are right now
- vertex_shader_v1 ← Where we are right now
- gpu_particle_burst_02 ← Where we are right now
- gpu_particle_burst_02_diff ← What we changed to get here

15.6.3 Step 3: Multiple Moving Particles



Next step is to have more than one particle, and to have the particles move. We'll do this by creating the particles, and calculating where they should be based on the time since creation. This is a bit different than the way we move sprites, as they are manually repositioned bit-by-bit during each update call.

Imports

First, we'll add imports for both the `random` and `time` libraries:

```
import random
import time
```

Constants

Then we need to create a constant that contains the number of particles to create:

```
PARTICLE_COUNT = 300
```

Burst Dataclass

We'll need to add a time to our burst data. This will be a floating point number that represents the start-time of when the burst was created.

```
@dataclass
class Burst:
    """ Track for each burst. """
    buffer: arcade.gl.Buffer
    vao: arcade.gl.Geometry
    start_time: float
```

Update Burst Creation

Now when we create a burst, we need multiple particles, and each particle also needs a velocity. In `_gen_initial_data` we add a loop for each particle, and also output a delta x and y.

Note: Because of how we set delta x and delta y, the particles will expand into a rectangle rather than a circle. We'll fix that on a later step.

Because we added a velocity, our buffer now needs two pairs of floats `2f 2f` named `in_pos` and `in_vel`. We'll update our shader in a bit to work with the new values.

Finally, our burst object needs to track the time we created the burst.

```python
    def on_mouse_press(self, x: float, y: float, button: int, modifiers: int):
        """ User clicks mouse """

        def _gen_initial_data(initial_x, initial_y):
            """ Generate data for each particle """
            for i in range(PARTICLE_COUNT):
                dx = random.uniform(-.2, .2)
                dy = random.uniform(-.2, .2)
                yield initial_x
                yield initial_y
                yield dx
                yield dy

        # Recalculate the coordinates from pixels to the OpenGL system with
        # 0, 0 at the center.
        x2 = x / self.width * 2. - 1.
        y2 = y / self.height * 2. - 1.

        # Get initial particle data
        initial_data = _gen_initial_data(x2, y2)

        # Create a buffer with that data
        buffer = self.ctx.buffer(data=array('f', initial_data))

        # Create a buffer description specifying the buffer's data format
        buffer_description = arcade.gl.BufferDescription(
            buffer,
            '2f 2f',
            ['in_pos', 'in_vel'])

        # Create our Vertex Attribute Object
        vao = self.ctx.geometry([buffer_description])

        # Create the Burst object and add it to the list of bursts
        burst = Burst(buffer=buffer, vao=vao, start_time=time.time())
        self.burst_list.append(burst)
```

Set Time in on_draw

When we draw, we need to set "uniform data" (data that is the same for all points) that says how many seconds it has been since the burst started. The shader will use this to calculate particle position.

```python
    def on_draw(self):
        """ Draw everything """
        self.clear()

        # Set the particle size
        self.ctx.point_size = 2 * self.get_pixel_ratio()

        # Loop through each burst
        for burst in self.burst_list:

            # Set the uniform data
            self.program['time'] = time.time() - burst.start_time

            # Render the burst
            burst.vao.render(self.program, mode=self.ctx.POINTS)
```

Update Vertex Shader

Our vertex shader needs to be updated. We now take in a `uniform float` called time. Uniform data is set once, and each vertex in the program can use it. In our case, we don't need a separate copy of the burst's start time for each particle in the burst, therefore it is uniform data.

We also need to add another vector of two floats that will take in our velocity. We set `in_vel` in *Update Burst Creation*.

Then finally we calculate a new position based on the time and our particle's velocity. We use that new position when setting `gl_Position`.

Listing 48: vertex_shader_v2.glsl

```glsl
#version 330

// Time since burst start
uniform float time;

// (x, y) position passed in
in vec2 in_pos;

// Velocity of particle
in vec2 in_vel;

// Output the color to the fragment shader
out vec4 color;

void main() {

    // Set the RGBA color
    color = vec4(1, 1, 1, 1);

    // Calculate a new position
```

(continues on next page)

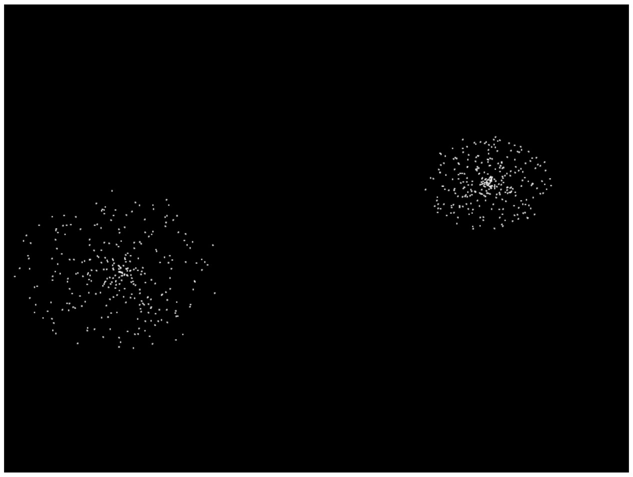
(continued from previous page)

```
    vec2 new_pos = in_pos + (time * in_vel);

    // Set the position. (x, y, z, w)
    gl_Position = vec4(new_pos, 0.0, 1);
}
```

Program Listings

- vertex_shader_v2 ← Where we are right now
- vertex_shader_v2_diff ← What we changed to get here
- gpu_particle_burst_03 ← Where we are right now
- gpu_particle_burst_03_diff ← What we changed to get here

15.6.4 Step 4: Random Angle and Speed



Step 3 didn't do a good job of picking a velocity, as our particles expanded into a rectangle rather than a circle. Rather than just pick a random delta x and y, we need to pick a random direction and speed. Then calculate delta x and y from that.

Update Imports

Import the math library so we can do some trig:

```
import math
```

Update Burst Creation

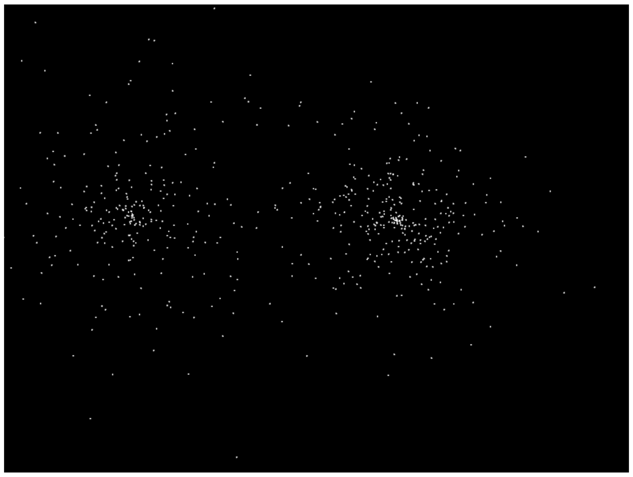
Now, pick a random direction from zero to 2 pi radians. Also, pick a random speed. Then use sine and cosine to calculate the delta x and y.

```
    def on_mouse_press(self, x: float, y: float, button: int, modifiers: int):
        """ User clicks mouse """

        def _gen_initial_data(initial_x, initial_y):
            """ Generate data for each particle """
            for i in range(PARTICLE_COUNT):
                angle = random.uniform(0, 2 * math.pi)
                speed = random.uniform(0.0, 0.3)
                dx = math.sin(angle) * speed
                dy = math.cos(angle) * speed
                yield initial_x
                yield initial_y
                yield dx
                yield dy

```

Program Listings

- gpu_particle_burst_04 ← Where we are right now
- gpu_particle_burst_04_diff ← What we changed to get here

15.6.5 Step 5: Gaussian Distribution



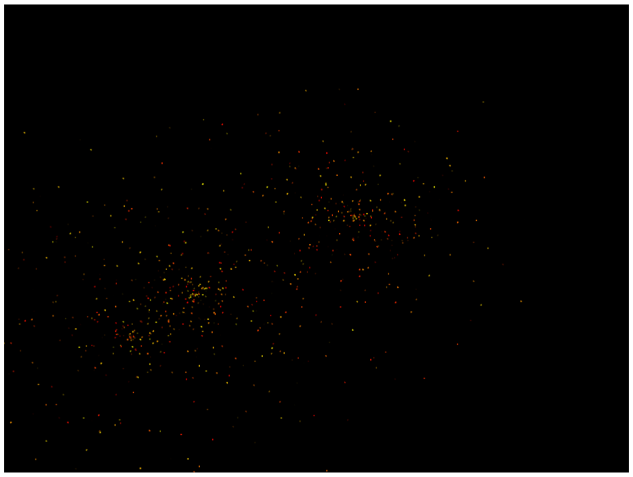
Setting speed to a random amount makes for an expanding circle. Another option is to use a gaussian function to produce more of a ‘splat’ look:

```
                speed = abs(random.gauss(0, 1)) * .5
```

Program Listings

- gpu_particle_burst_05 ← Where we are right now
- gpu_particle_burst_05_diff ← What we changed to get here

15.6.6 Step 6: Add Color



So far our particles have all been white. How do we add in color? We'll need to generate it for each particle. Shaders take colors in the form of RGB floats, so we'll generate a random number for red, and add in some green to get our yellows. Don't add more green than red, or else you get a green tint.

Finally, make sure to update the shader buffer description (VBO) to accept the three color channel floats (`3f`) under the name `in_color`.

```
    def on_mouse_press(self, x: float, y: float, button: int, modifiers: int):
        """ User clicks mouse """

        def _gen_initial_data(initial_x, initial_y):
            """ Generate data for each particle """
            for i in range(PARTICLE_COUNT):
                angle = random.uniform(0, 2 * math.pi)
                speed = abs(random.gauss(0, 1)) * .5
                dx = math.sin(angle) * speed
                dy = math.cos(angle) * speed
                red = random.uniform(0.5, 1.0)
                green = random.uniform(0, red)
                blue = 0
                yield initial_x
                yield initial_y
                yield dx
                yield dy
                yield red
                yield green
                yield blue

        # Recalculate the coordinates from pixels to the OpenGL system with
        # 0, 0 at the center.
```

(continues on next page)

(continued from previous page)

```
        x2 = x / self.width * 2. - 1.
        y2 = y / self.height * 2. - 1.

        # Get initial particle data
        initial_data = _gen_initial_data(x2, y2)

        # Create a buffer with that data
        buffer = self.ctx.buffer(data=array('f', initial_data))

        # Create a buffer description specifying the buffer's data format
        buffer_description = arcade.gl.BufferDescription(
            buffer,
            '2f 2f 3f',
            ['in_pos', 'in_vel', 'in_color'])

        # Create our Vertex Attribute Object
        vao = self.ctx.geometry([buffer_description])

        # Create the Burst object and add it to the list of bursts
        burst = Burst(buffer=buffer, vao=vao, start_time=time.time())
        self.burst_list.append(burst)
```

Then, update the shader to use the color instead of always using white:

Listing 49: vertex_shader_v3.glsl

```
#version 330

// Time since burst start
uniform float time;

// (x, y) position passed in
in vec2 in_pos;

// Velocity of particle
in vec2 in_vel;

// Color of particle
in vec3 in_color;

// Output the color to the fragment shader
out vec4 color;

void main() {

    // Set the RGBA color
    color = vec4(in_color[0], in_color[1], in_color[2], 1);

    // Calculate a new position
    vec2 new_pos = in_pos + (time * in_vel);

    // Set the position. (x, y, z, w)
```

(continues on next page)

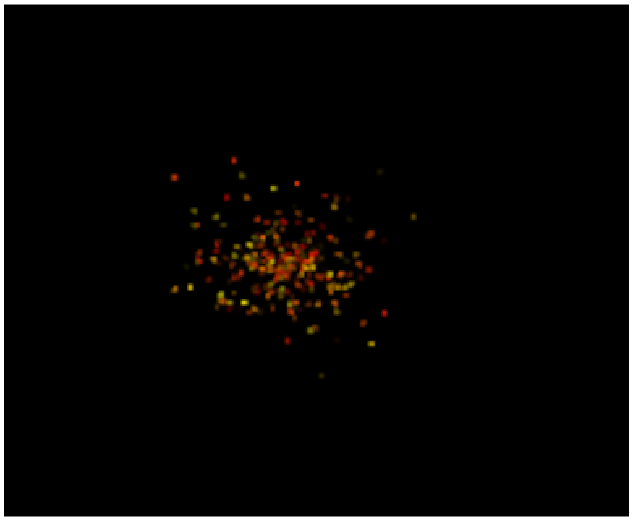
(continued from previous page)

```
    gl_Position = vec4(new_pos, 0.0, 1);
}
```

Program Listings

- vertex_shader_v3 ← Where we are right now
- vertex_shader_v3_diff ← What we changed to get here
- gpu_particle_burst_06 ← Where we are right now
- gpu_particle_burst_06_diff ← What we changed to get here

15.6.7 Step 7: Fade Out



Right now the explosion particles last forever. Let's get them to fade out. Once a burst has faded out, let's remove it from `burst_list`.

Constants

First, let's add a couple constants to control the minimum and maximum times to fade a particle:

```
MIN_FADE_TIME = 0.25
MAX_FADE_TIME = 1.5
```

Update Init

Next, we need to update our OpenGL context to support alpha blending. Go back to the `__init__` method and update the `enable_only` call to:

```
self.ctx.enable_only(self.ctx.BLEND)
```

Add Fade Rate to Buffer

Next, add the fade rate float to the VBO:

```
    def on_mouse_press(self, x: float, y: float, button: int, modifiers: int):
        """ User clicks mouse """

        def _gen_initial_data(initial_x, initial_y):
            """ Generate data for each particle """
            for i in range(PARTICLE_COUNT):
                angle = random.uniform(0, 2 * math.pi)
                speed = abs(random.gauss(0, 1)) * .5
                dx = math.sin(angle) * speed
                dy = math.cos(angle) * speed
                red = random.uniform(0.5, 1.0)
                green = random.uniform(0, red)
                blue = 0
                fade_rate = random.uniform(
                    1 / MAX_FADE_TIME, 1 / MIN_FADE_TIME)

                yield initial_x
                yield initial_y
                yield dx
                yield dy
                yield red
                yield green
                yield blue
                yield fade_rate

        # Recalculate the coordinates from pixels to the OpenGL system with
        # 0, 0 at the center.
        x2 = x / self.width * 2. - 1.
        y2 = y / self.height * 2. - 1.

        # Get initial particle data
        initial_data = _gen_initial_data(x2, y2)

        # Create a buffer with that data
        buffer = self.ctx.buffer(data=array('f', initial_data))

        # Create a buffer description specifying the buffer's data format
        buffer_description = arcade.gl.BufferDescription(
            buffer,
            '2f 2f 3f f',
            ['in_pos', 'in_vel', 'in_color', 'in_fade_rate'])
```

(continues on next page)

(continued from previous page)

```

        # Create our Vertex Attribute Object
        vao = self.ctx.geometry([buffer_description])

        # Create the Burst object and add it to the list of bursts
        burst = Burst(buffer=buffer, vao=vao, start_time=time.time())
        self.burst_list.append(burst)
```

Update Shader

Update the shader. Calculate the alpha. If it is less that 0, just use 0.

Listing 50: vertex_shader_v4.glsl

```
#version 330

// Time since burst start
uniform float time;

// (x, y) position passed in
in vec2 in_pos;

// Velocity of particle
in vec2 in_vel;

// Color of particle
in vec3 in_color;

// Fade rate
in float in_fade_rate;

// Output the color to the fragment shader
out vec4 color;

void main() {

    // Calculate alpha based on time and fade rate
    float alpha = 1.0 - (in_fade_rate * time);
    if(alpha < 0.0) alpha = 0;

    // Set the RGBA color
    color = vec4(in_color[0], in_color[1], in_color[2], alpha);

    // Calculate a new position
    vec2 new_pos = in_pos + (time * in_vel);

    // Set the position. (x, y, z, w)
    gl_Position = vec4(new_pos, 0.0, 1);
}
```

Remove Faded Bursts

Once our burst has completely faded, no need to keep it around. So in our `on_update` remove the burst from the burst_list after it has been faded.

```python
    def on_update(self, dt):
        """ Update game """

        # Create a copy of our list, as we can't modify a list while iterating
        # it. Then see if any of the items have completely faded out and need
        # to be removed.
        temp_list = self.burst_list.copy()
        for burst in temp_list:
            if time.time() - burst.start_time > MAX_FADE_TIME:
                self.burst_list.remove(burst)
```

Program Listings

- vertex_shader_v4 ← Where we are right now
- vertex_shader_v4_diff ← What we changed to get here
- gpu_particle_burst_07 ← Where we are right now
- gpu_particle_burst_07_diff ← What we changed to get here

15.6.8 Step 8: Add Gravity

You could also add come gravity to the particles by adjusting the velocity based on a gravity constant. (In this case, 1.1.)

```glsl
// Adjust velocity based on gravity
vec2 new_vel = in_vel;
new_vel[1] -= time * 1.1;

// Calculate a new position
vec2 new_pos = in_pos + (time * new_vel);
```

Program Listings

- vertex_shader_v5 ← Where we are right now
- vertex_shader_v5_diff ← What we changed to get here

15.7 Working With Shaders

Shaders are graphics programs that run on GPU and can be used for many varied purposes.

Here we look at some very simple shader programs and learn how to pass data to and from shaders

15.7.1 Basic Arcade Program

Listing 51: Starting template

```
import arcade

SCREEN_WIDTH = 800
SCREEN_HEIGHT = 600
SCREEN_TITLE = "Basic Arcade Template"


class MyWindow(arcade.Window):
    def __init__(self):
        super().__init__(SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_TITLE)
        self.center_window()
        self.background_color = arcade.color.ALMOND

    def on_draw(self):
        # Draw a simple circle to the screen
        self.clear()
        arcade.draw_circle_filled(
            SCREEN_WIDTH / 2,
            SCREEN_HEIGHT / 2,
            100,
            arcade.color.AFRICAN_VIOLET
        )


app = MyWindow()
arcade.run()
```

15.7.2 Basic Shader Program

From here we add a very basic shader and draw it to the screen. This shader simply sets color and alpha based on the horizontal coordinate of the pixel.

We have to define vertex shader and fragment shader programs.

- Vertex shaders run on each passed coorninate and can modify it. Here we use it only to pass on the coordinate on to the fragment shader
- Fragment shaders set color for each passed pixel. Here we set a fixed color for every pixel and vary alpha based on horizontal position

We need to pass the shader the pixel coordinates so create an object `quad_fs` to facilitate it.

Listing 52: Simple shader

```python
import arcade

SCREEN_WIDTH = 800
SCREEN_HEIGHT = 600
SCREEN_TITLE = "Basic Vertex and Fragment Shader"


class MyWindow(arcade.Window):
    def __init__(self):
        super().__init__(SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_TITLE)
        self.center_window()
        self.background_color = arcade.color.ALMOND

        # GL geometry that will be used to pass pixel coordinates to the shader
        # It has the same dimensions as the screen
        self.quad_fs = arcade.gl.geometry.quad_2d_fs()

        # Create a simple shader program
        self.prog = self.ctx.program(
            vertex_shader="""
            #version 330
            in vec2 in_vert;
            void main()
            {
                gl_Position = vec4(in_vert, 0., 1.);
            }
            """,
            fragment_shader="""
            #version 330
            out vec4 fragColor;
            void main()
            {
                // Set the pixel colour and alpha based on x position
                fragColor = vec4(0.9, 0.5, 0.5, sin(gl_FragCoord.x / 50));
            }
            """
        )

    def on_draw(self):
        # Draw a simple circle
        self.clear()
        arcade.draw_circle_filled(
            SCREEN_WIDTH / 2,
            SCREEN_HEIGHT / 2,
            100,
            arcade.color.AFRICAN_VIOLET
        )

        # Run the shader and render to screen
        # The shader code is run once for each pixel coordinate in quad_fs
        # and the fragColor output added to the screen
```

(continues on next page)

(continued from previous page)

```
        self.quad_fs.render(self.prog)


app = MyWindow()
arcade.run()
```

15.7.3 Passing Data To The Shader

To pass data to the shader program we can define uniforms. Uniforms are global shader variables that act as parameters passed from outside the shader program.

We have to define uniform within the shader and then register the python variable with the shader program before rendering.

It is important to make sure that the uniform type is appropriate for the data being passed.

Listing 53: Uniforms

```
import arcade

SCREEN_WIDTH = 800
SCREEN_HEIGHT = 600
SCREEN_TITLE = "Shader With Uniform"


class MyWindow(arcade.Window):
    def __init__(self):
        super().__init__(SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_TITLE)
        self.center_window()
        self.background_color = arcade.color.ALMOND

        # GL geometry that will be used to pass pixel coordinates to the shader
        # It has the same dimensions as the screen
        self.quad_fs = arcade.gl.geometry.quad_2d_fs()

        # Create a simple shader program
        self.prog = self.ctx.program(
            vertex_shader="""
            #version 330
            in vec2 in_vert;
            void main()
            {
                gl_Position = vec4(in_vert, 0., 1.);
            }
            """,
            fragment_shader="""
            #version 330
            // Define an input to receive total_time from python
            uniform float time;
            out vec4 fragColor;
            void main()
            {
```

(continues on next page)

(continued from previous page)

```
                // Set the pixel colour and alpha based on x position and time
                fragColor = vec4(0.9, 0.5, 0.5, sin(gl_FragCoord.x / 50 + time));
            }
            """
        )

        # Create a variable to track program run time
        self.total_time = 0

    def on_update(self, delta_time):
        # Keep tract o total time
        self.total_time += delta_time

    def on_draw(self):
        # Draw a simple circle
        self.clear()
        arcade.draw_circle_filled(
            SCREEN_WIDTH / 2,
            SCREEN_HEIGHT / 2,
            100,
            arcade.color.AFRICAN_VIOLET
        )

        # Register the uniform in the shader program
        self.prog['time'] = self.total_time

        # Run the shader and render to screen
        # The shader code is run once for each pixel coordinate in quad_fs
        # and the fragColor output added to the screen
        self.quad_fs.render(self.prog)


app = MyWindow()
arcade.run()
```

15.7.4 Accessing Textures From The Shader

To make the shader more useful we may wish to pass textures to it.

Here we create to textures (and associated framebuffers) and pass them to the shader as uniform sampler objects. Unlike other uniforms we need to assign a reference to an integer texture channel (rather than directly to the python object) and `.use()` the texture to bind it to that channel.

Listing 54: Textures

```
import arcade

SCREEN_WIDTH = 800
SCREEN_HEIGHT = 600
SCREEN_TITLE = "Shader with Textures"

```

(continues on next page)

(continued from previous page)

```python

class MyWindow(arcade.Window):
    def __init__(self):
        super().__init__(SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_TITLE)
        self.center_window()
        self.background_color = arcade.color.ALMOND

        # GL geometry that will be used to pass pixel coordinates to the shader
        # It has the same dimensions as the screen
        self.quad_fs = arcade.gl.geometry.quad_2d_fs()

        # Create textures and FBOs
        self.tex_0 = self.ctx.texture((self.width, self.height))
        self.fbo_0 = self.ctx.framebuffer(color_attachments=[self.tex_0])

        self.tex_1 = self.ctx.texture((self.width, self.height))
        self.fbo_1 = self.ctx.framebuffer(color_attachments=[self.tex_1])

        # Fill the textures with solid colours
        self.fbo_0.clear(color=(0.0, 0.0, 1.0, 1.0), normalized=True)
        self.fbo_1.clear(color=(1.0, 0.0, 0.0, 1.0), normalized=True)

        # Create a simple shader program
        self.prog = self.ctx.program(
            vertex_shader="""
            #version 330
            in vec2 in_vert;
            // Get normalized coordinates
            in vec2 in_uv;
            out vec2 uv;
            void main()
            {
                gl_Position = vec4(in_vert, 0., 1.);
                uv = in_uv;
            }
            """,
            fragment_shader="""
            #version 330
            // Define an input to receive total_time from python
            uniform float time;
            // Define inputs to access textures
            uniform sampler2D t0;
            uniform sampler2D t1;
            in vec2 uv;
            out vec4 fragColor;
            void main()
            {
                // Set pixel color as a combination of the two textures
                fragColor = mix(
                    texture(t0, uv),
                    texture(t1, uv),
                    smoothstep(0.0, 1.0, uv.x));
```

(continues on next page)

(continued from previous page)

```
                // Set the alpha based on time
                fragColor.w = sin(time);
            }
            """
        )

        # Register the texture uniforms in the shader program
        self.prog['t0'] = 0
        self.prog['t1'] = 1

        # Create a variable to track program run time
        self.total_time = 0

    def on_update(self, delta_time):
        # Keep tract o total time
        self.total_time += delta_time

    def on_draw(self):
        # Draw a simple circle
        self.clear()
        arcade.draw_circle_filled(
            SCREEN_WIDTH / 2,
            SCREEN_HEIGHT / 2,
            100,
            arcade.color.AFRICAN_VIOLET
        )

        # Register the uniform in the shader program
        self.prog['time'] = self.total_time

        # Bind our textures to channels
        self.tex_0.use(0)
        self.tex_1.use(1)

        # Run the shader and render to screen
        # The shader code is run once for each pixel coordinate in quad_fs
        # and the fragColor output added to the screen
        self.quad_fs.render(self.prog)


app = MyWindow()
arcade.run()
```

15.7.5 Drawing To Texture From The Shader

Finally we have an example of reading from and writing to the same texture with a shader.

We use the `with fbo:` syntax to tell arcade that we wish to render to the new frambuffer rather than default one.

Once the shader has updated the framebuffer we need to copy its contents to the screen to be displayed.

Listing 55: Textures

```python
import arcade

SCREEN_WIDTH = 800
SCREEN_HEIGHT = 600
SCREEN_TITLE = "An Empty Program"


class MyWindow(arcade.Window):
    def __init__(self):
        super().__init__(SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_TITLE)
        self.center_window()
        self.background_color = arcade.color.ALMOND

        # GL geometry that will be used to pass pixel coordinates to the shader
        # It has the same dimensions as the screen
        self.quad_fs = arcade.gl.geometry.quad_2d_fs()

        # Create texture and FBO
        self.tex = self.ctx.texture((self.width, self.height))
        self.fbo = self.ctx.framebuffer(color_attachments=[self.tex])

        # Put something in the framebuffer to start
        self.fbo.clear(arcade.color.ALMOND)
        with self.fbo:
            arcade.draw_circle_filled(
                SCREEN_WIDTH / 2,
                SCREEN_HEIGHT / 2,
                100,
                arcade.color.AFRICAN_VIOLET
            )

        # Create a simple shader program
        self.prog = self.ctx.program(
            vertex_shader="""
            #version 330
            in vec2 in_vert;
            void main()
            {
                gl_Position = vec4(in_vert, 0., 1.);
            }
            """,
            fragment_shader="""
            #version 330
            // Define input to access texture
```

(continues on next page)

(continued from previous page)

```
            uniform sampler2D t0;
            out vec4 fragColor;
            void main()
            {
                // Overwrite this pixel with the colour from its neighbour
                ivec2 pos = ivec2(gl_FragCoord.xy) + ivec2(-1, -1);
                fragColor = texelFetch(t0, pos, 0);
            }
            """
        )

        # Register the texture uniform in the shader program
        self.prog['t0'] = 0

    def on_draw(self):
        # Activate our new framebuffer to render to
        with self.fbo:
            # Bind our texture to the first channel
            self.tex.use(0)

            # Run the shader and render to the framebuffer
            self.quad_fs.render(self.prog)

        # Copy the framebuffer to the screen to display
        self.ctx.copy_framebuffer(self.fbo, self.ctx.screen)


app = MyWindow()
arcade.run()
```

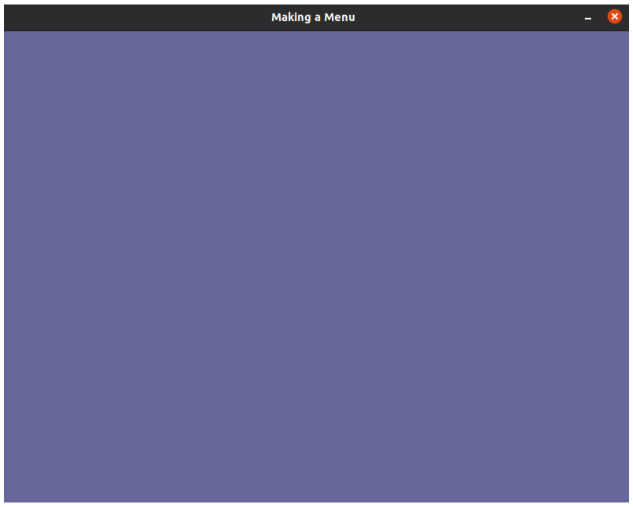
CHAPTER

SIXTEEN

MAKING A MENU WITH ARCADE'S GUI

This tutorial shows how to use most of arcade's gui's widgets.

16.1 Step 1: Open a Window



First, let's start a blank window with a view.

Listing 1: Opening a Window

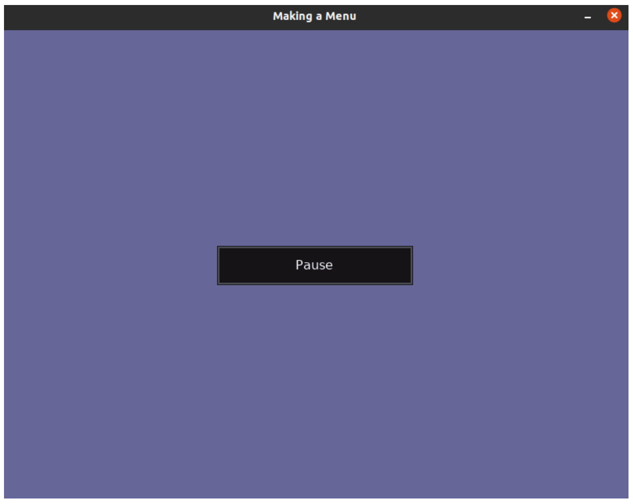
```
"""
Menu.

Shows the usage of almost every gui widget, switching views and making a modal.
"""
import arcade

# Screen title and size
SCREEN_WIDTH = 800
SCREEN_HEIGHT = 600
SCREEN_TITLE = "Making a Menu"


class MainView(arcade.View):
```

(continues on next page)

(continued from previous page)

```python
    """ Main application class."""

    def __init__(self):
        super().__init__()

    def on_show_view(self):
        """ This is run once when we switch to this view """
        arcade.set_background_color(arcade.color.DARK_BLUE_GRAY)

    def on_draw(self):
        """ Render the screen. """
        # Clear the screen
        self.clear()


def main():
    window = arcade.Window(SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_TITLE, resizable=True)
    main_view = MainView()
    window.show_view(main_view)
    arcade.run()


if __name__ == "__main__":
    main()
```

16.2 Step 2: Switching to Menu View



For this section we will switch the current view of the window to the menu view.

16.2.1 Imports

First we will import the arcade gui:

Listing 2: Importing arcade.gui

```
Shows the usage of almost every gui widget, switching views and making a modal.
"""
```

16.2.2 Modify the MainView

We are going to add a button to change the view. For drawing a button we would need a `UIManager`.

Listing 3: Intialising the Manager

```
    """This is the class where your normal game would go."""

    def __init__(self):
        super().__init__()
```

After initialising the manager we need to enable it when the view is shown and disable it when the view is hiddien.

Listing 4: Enabling the Manager

```
    def on_show_view(self):
        """ This is run once when we switch to this view """
        arcade.set_background_color(arcade.color.DARK_BLUE_GRAY)

        # Enable the UIManager when the view is showm.
        self.manager.enable()
```

Listing 5: Disabling the Manager

```
    def on_hide_view(self):
        # Disable the UIManager when the view is hidden.
        self.manager.disable()
```

We also need to draw the childrens of the menu in `on_draw`.

Listing 6: Drawing Children's of the Manager

```
    def on_draw(self):
        """ Render the screen. """
        # Clear the screen
        self.clear()

        # Draw the manager.
        self.manager.draw()
```

Now we have successfully setup the manager, only thing left it to add the button. We are using `UIAnchorLayout` to position the button. We also setup a function which is called when the button is clicked.

Listing 7: Initialising the Button

```
        self.manager = arcade.gui.UIManager()

        switch_menu_button = arcade.gui.UIFlatButton(text="Pause", width=250)

        # Initialise the button with an on_click event.
        @switch_menu_button.event("on_click")
        def on_click_switch_button(event):
            # Passing the main view into menu view as an argument.
            menu_view = MenuView(self)
            self.window.show_view(menu_view)

        # Use the anchor to position the button on the screen.
        self.anchor = self.manager.add(arcade.gui.UIAnchorLayout())

        self.anchor.add(
            anchor_x="center_x",
            anchor_y="center_y",
```

16.2.3 Initialise the Menu View

We make a boiler plate view just like we did in Step-1 for switching the view when the pause button is clicked.

Listing 8: Initialise the Menu View

```
class MenuView(arcade.View):
    """Main menu view class."""

    def __init__(self, main_view):
        super().__init__()

        self.manager = arcade.gui.UIManager()

        self.main_view = main_view

    def on_hide_view(self):
        # Disable the UIManager when the view is hidden.
        self.manager.disable()

    def on_show_view(self):
        """ This is run once when we switch to this view """

        # Makes the background darker
        arcade.set_background_color([rgb - 50 for rgb in arcade.color.DARK_BLUE_GRAY])

        self.manager.enable()

    def on_draw(self):
        """ Render the screen. """

        # Clear the screen
```

(continues on next page)

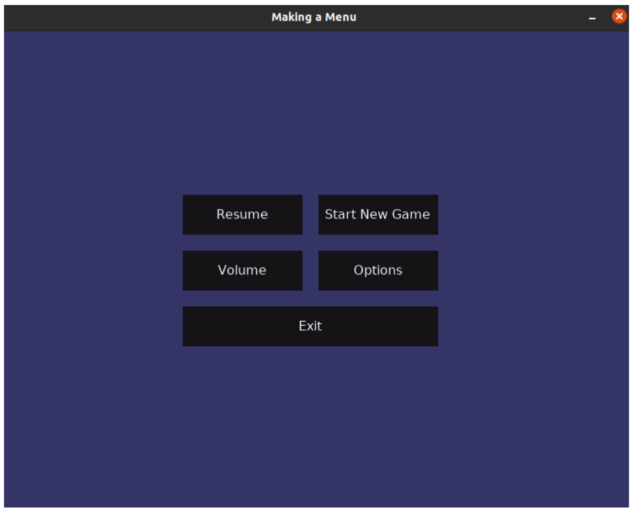
(continued from previous page)

```
        self.clear()
        self.manager.draw()
```

16.2.4 Program Listings

- menu_02 ← Where we are right now
- menu_02_diff ← What we changed to get here

16.3 Step 3: Setting Up the Menu View



In this step we will setup the display buttons of the actual menu. The code written in this section is written for `MenuView`

16.3.1 Initialising the Buttons

First we setup buttons for resume, starting a new game, volume, options and exit.

Listing 9: Initialising the Buttons

```
        self.manager = arcade.gui.UIManager()

        resume = arcade.gui.UIFlatButton(text="Resume", width=150)
        start_new_game = arcade.gui.UIFlatButton(text="Start New Game", width=150)
        volume = arcade.gui.UIFlatButton(text="Volume", width=150)
        options = arcade.gui.UIFlatButton(text="Options", width=150)
```

16.3.2 Displaying the Buttons in a Grid

After setting up the buttons we add them to `UIGridLayout`, so that they can displayed in a grid like manner.

Listing 10: Setting up the Grid

```
        exit = arcade.gui.UIFlatButton(text="Exit", width=320)

        # Initialise a grid in which widgets can be arranged.
        self.grid = arcade.gui.UIGridLayout(column_count=2, row_count=3, horizontal_
↪spacing=20, vertical_spacing=20)

        # Adding the buttons to the layout.
        self.grid.add(resume, col_num=0, row_num=0)
        self.grid.add(start_new_game, col_num=1, row_num=0)
        self.grid.add(volume, col_num=0, row_num=1)
        self.grid.add(options, col_num=1, row_num=1)
        self.grid.add(exit, col_num=0, row_num=2, col_span=2)

        self.anchor = self.manager.add(arcade.gui.UIAnchorLayout())

        self.anchor.add(
            anchor_x="center_x",
            anchor_y="center_y",
```

Final code for the `__init__` method after these.

Listing 11: __init__

```
    def __init__(self, main_view):
        super().__init__()

        self.manager = arcade.gui.UIManager()

        resume = arcade.gui.UIFlatButton(text="Resume", width=150)
        start_new_game = arcade.gui.UIFlatButton(text="Start New Game", width=150)
        volume = arcade.gui.UIFlatButton(text="Volume", width=150)
        options = arcade.gui.UIFlatButton(text="Options", width=150)

        exit = arcade.gui.UIFlatButton(text="Exit", width=320)

        # Initialise a grid in which widgets can be arranged.
        self.grid = arcade.gui.UIGridLayout(column_count=2, row_count=3, horizontal_
```

(continues on next page)

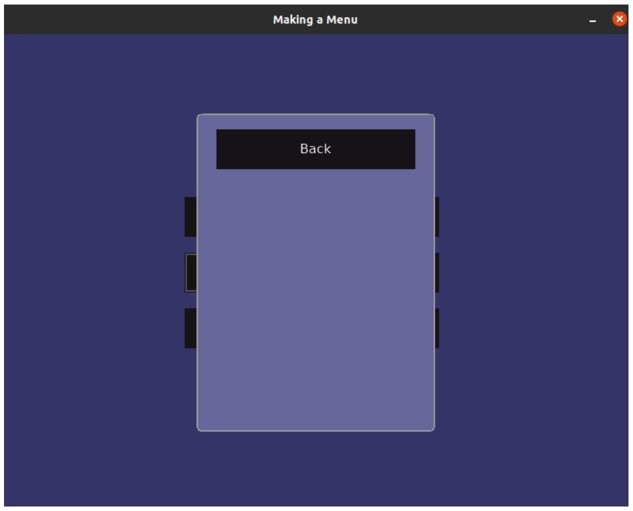
(continued from previous page)

```
↪spacing=20, vertical_spacing=20)

        # Adding the buttons to the layout.
        self.grid.add(resume, col_num=0, row_num=0)
        self.grid.add(start_new_game, col_num=1, row_num=0)
        self.grid.add(volume, col_num=0, row_num=1)
        self.grid.add(options, col_num=1, row_num=1)
        self.grid.add(exit, col_num=0, row_num=2, col_span=2)

        self.anchor = self.manager.add(arcade.gui.UIAnchorLayout())

        self.anchor.add(
            anchor_x="center_x",
            anchor_y="center_y",
            child=self.grid,
        )

        self.main_view = main_view
```

16.3.3 Program Listings

- menu_03 ← Where we are right now
- menu_03_diff ← What we changed to get here

16.4 Step 4: Configuring the Menu Buttons



We basically add event listener for `on_click` for buttons.

16.4.1 Adding `on_click` Callback for Resume, Start New Game and Exit

First we will add the event listener to resume, start_new_game and exit button as they don't have much to explain.

Listing 12: Adding callback for button events 1

```
        self.main_view = main_view

        @resume_button.event("on_click")
        def on_click_resume_button(event):
            # Pass already created view because we are resuming.
            self.window.show_view(self.main_view)

        @start_new_game_button.event("on_click")
        def on_click_start_new_game_button(event):
            # Create a new view because we are starting a new game.
            main_view = MainView()
            self.window.show_view(main_view)

        @exit_button.event("on_click")
```

16.4.2 Adding `on_click` Callback for Volume and Options

Now we need to implement an actual menu for volume and options, for that we have to make a class that acts like a window. Using `UIMouseFilterMixin` we catch all the events happening for the parent and respond nothing to them. Thus making it act like a window/view.

Listing 13: Making a Fake Window.

```
class SubMenu(arcade.gui.UIMouseFilterMixin, arcade.gui.UIAnchorLayout):
    """Acts like a fake view/window."""

    def __init__(self, ):
        super().__init__(size_hint=(1, 1))

        # Setup frame which will act like the window.
        frame = self.add(arcade.gui.UIAnchorLayout(width=300, height=400, size_
↪hint=None))
        frame.with_padding(all=20)

        # Add a background to the window.
        frame.with_background(texture=arcade.gui.NinePatchTexture(
            left=7,
            right=7,
            bottom=7,
            top=7,
            texture=arcade.load_texture(
                ":resources:gui_basic_assets/window/dark_blue_gray_panel.png"
            )
        ))

        back_button = arcade.gui.UIFlatButton(text="Back", width=250)
        # The type of event listener we used earlier for the button will not work here.
```

(continues on next page)

(continued from previous page)

```
        back_button.on_click = self.on_click_back_button

        # Internal widget layout to handle widgets in this class.
        widget_layout = arcade.gui.UIBoxLayout(align="left", space_between=10)

        widget_layout.add(back_button)

        frame.add(child=widget_layout, anchor_x="center_x", anchor_y="top")

    def on_click_back_button(self, event):
        # Removes the widget from the manager.
        # After this the manager will respond to its events like it previously did.
        self.parent.remove(self)
```

We have got ourselves a fake window currently. We now, pair it up with the volume and options button to trigger it when they are clicked.

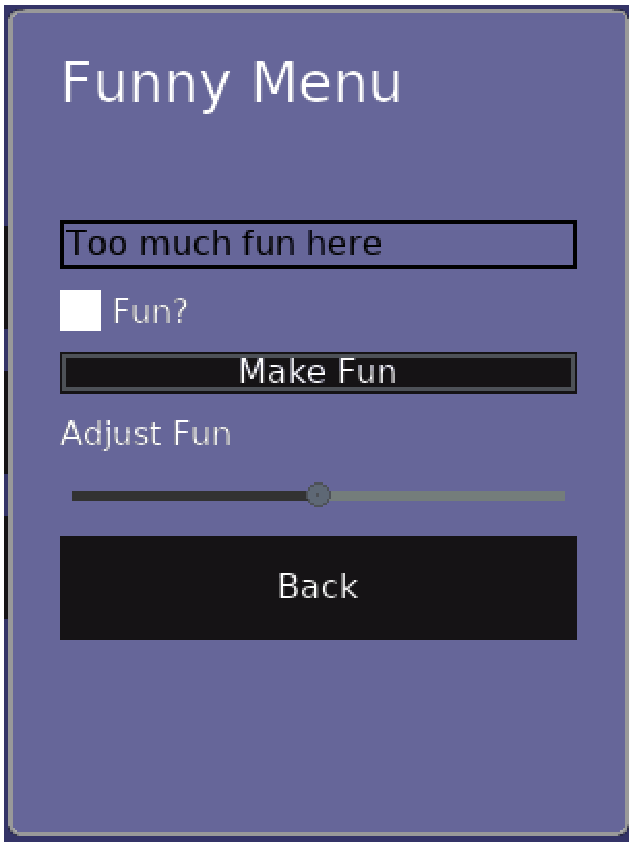
Listing 14: Adding callback for button events 2

```
            arcade.exit()

        @volume_button.event("on_click")
        def on_click_volume_button(event):
            volume_menu = SubMenu()
            self.manager.add(
                volume_menu,
                layer=1
            )

        @options_button.event("on_click")
        def on_click_options_button(event):
            options_menu = SubMenu()
            self.manager.add(
                options_menu,
```

16.4.3 Program Listings

- menu_04 ← Where we are right now
- menu_04_diff ← What we changed to get here

16.5 Step 5: Finalising the Fake Window aka the Sub Menu



We finalise the menu or you can call it the last step!

16.5.1 Editing the Parameters for the Sub Menu

We will edit the parameters for the sub menu to suit our needs. Will explain later why are those parameters needed.

Listing 15: Editing parameters

```
        self.clear()
        self.manager.draw()
```

We also need to change accordingly the places where we have used this class i.e options and volume `on_click` event listener. The layer parameter being set 1, means that this layer is always drawn on top i.e its the first layer.

Listing 16: Editing arguments

```
        @exit_button.event("on_click")
        def on_click_exit_button(event):
            arcade.exit()

        @volume_button.event("on_click")
```

(continues on next page)

(continued from previous page)

```
        def on_click_volume_button(event):
            volume_menu = SubMenu(
                "Volume Menu", "How do you like your volume?", "Enable Sound",
                ["Play: Rock", "Play: Punk", "Play: Pop"],
                "Adjust Volume",
            )
            self.manager.add(
                volume_menu,
                layer=1
            )

        @options_button.event("on_click")
        def on_click_options_button(event):
            options_menu = SubMenu(
                "Funny Menu", "Too much fun here", "Fun?",
                ["Make Fun", "Enjoy Fun", "Like Fun"],
                "Adjust Fun",
            )
```

Now you might be getting a little idea why we have edited the parameters but
follow on to actually know the reason.

16.6 Adding a Title label

We will be adding a `UILabel` that explains the menu. `UISpace` is a widget that can be used to add space around some widget, you can set its color to the background color so it appears invisible.

Listing 17: Adding title label

```
        back_button = arcade.gui.UIFlatButton(text="Back", width=250)
        # The type of event listener we used earlier for the button will not work here.
        back_button.on_click = self.on_click_back_button
```

Adding it to the widget layout.

Listing 18: Adding title label to the layout

```
        style_dict = {"press": pressed_style, "normal": default_style, "hover": default_
↪style, "disabled": default_style}
        # Configuring the styles is optional.
        slider = arcade.gui.UISlider(value=50, width=250, style=style_dict)
```

16.6.1 Adding a Input Field

We will use `UIInputText` to add an input field. The `with_border()` function creates a border around the widget with color(default argument is black) black and thickness(default argument is 2px) 2px. Add this just below the title label.

Listing 19: Adding input field

```
        title_label = arcade.gui.UILabel(text=title, align="center", font_size=20,␣
↪multiline=False)
```

Adding it to the widget layout.

Listing 20: Adding input field to the layout

```
        style_dict = {"press": pressed_style, "normal": default_style, "hover": default_
↪style, "disabled": default_style}
        # Configuring the styles is optional.
        slider = arcade.gui.UISlider(value=50, width=250, style=style_dict)

```

If you paid attention when we defined the `input_text` variable we passed the `text` parameter with our `input_text_default` argument. We basically added those parameters in our sub menu so that it can be used by both volume and options button, with texts respecting their names. We will repeat this again in the last also for those of you who are skipping through this section :P.

16.6.2 Adding a Toggle Button

Don't go on the section title much, in arcade the `UITextureToggle` is not really a button it switches between two textures when clicked. Yes, it functions like a button but by "is not really a button" we meant that it doesn't inherits the button class. We also pair it up horizontally with the toggle label.

Listing 21: Adding toggle button

```
        # Load the on-off textures.
        on_texture = arcade.load_texture(":resources:gui_basic_assets/toggle/circle_
↪switch_on.png")
        off_texture = arcade.load_texture(":resources:gui_basic_assets/toggle/circle_
↪switch_off.png")

        # Create the on-off toggle and a label
        toggle_label = arcade.gui.UILabel(text=toggle_label)
        toggle = arcade.gui.UITextureToggle(
            on_texture=on_texture,
            off_texture=off_texture,
```

(continues on next page)

(continued from previous page)

```
            width=20,
            height=20
        )
```

Adding it to the widget layout. Add this line after you have added the input field.

Listing 22: Adding toggle button to the layout

```
        widget_layout = arcade.gui.UIBoxLayout(align="left", space_between=10)
```

16.6.3 Adding a Dropdown

We add a dropdown by using `UIDropdown`.

Listing 23: Adding dropdown

```
        toggle_group = arcade.gui.UIBoxLayout(vertical=False, space_between=5)
        toggle_group.add(toggle)
```

Adding it to the widget layout.

Listing 24: Adding dropdown to the layout

```
        widget_layout.add(title_label)
```

16.6.4 Adding a Slider

The final widget. In arcade you can use `UISlider` to implement a slider. Theres a functionality to style the slider, this is also present for `UIFlatButton` and `UITextureButton`.

Listing 25: Adding slider

```

        # Create dropdown with a specified default.
```

Adding it to the widget layout.

Listing 26: Adding slider to the layout

```
        widget_layout.add(title_label_space)
        widget_layout.add(input_text_widget)
```

16.6.5 Finishing touches

As we mentioned earlier, to explain the use of those parameters to the class. We basically used them so it can be used by both options and volume as we wanted to have different text for both. For those who have read the full tutorial line-by-line; ‘They will never know’. :D. We also recommend to see the full code for this section.

16.6.6 Program Listings

- menu_05 ← Where we are right now
- menu_05_diff ← What we changed to get here

CHAPTER

SEVENTEEN

WORKING WITH FRAMEBUFFER OBJECTS

Start with a simple window:

Listing 1: Starting template

```python
import arcade

SCREEN_WIDTH = 800
SCREEN_HEIGHT = 600
SCREEN_TITLE = "Frame Buffer Object Demo"


class MyGame(arcade.Window):

    def __init__(self, width, height, title):
        super().__init__(width, height, title)

        self.background_color = arcade.color.ALMOND

    def setup(self):
        pass

    def on_draw(self):
        self.clear()


def main():
    """ Main function """
    window = MyGame(SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_TITLE)
    window.setup()
    arcade.run()


if __name__ == "__main__":
    main()
```

Then create a simple program with a frame buffer:

Listing 2: Pass-through frame buffer

```python
import arcade
from arcade.experimental.texture_render_target import RenderTargetTexture
```

(continues on next page)

(continued from previous page)

```

SCREEN_WIDTH = 800
SCREEN_HEIGHT = 600
SCREEN_TITLE = "Starting Template Simple"


class RandomFilter(RenderTargetTexture):
    def __init__(self, width, height):
        super().__init__(width, height)
        self.program = self.ctx.program(
            vertex_shader="""
            #version 330

            in vec2 in_vert;
            in vec2 in_uv;
            out vec2 uv;

            void main() {
                gl_Position = vec4(in_vert, 0.0, 1.0);
                uv = in_uv;
            }
            """,
            fragment_shader="""
            #version 330

            uniform sampler2D texture0;

            in vec2 uv;
            out vec4 fragColor;

            void main() {
                vec4 color = texture(texture0, uv);
                fragColor = color;
            }
            """,
        )

    def use(self):
        self._fbo.use()

    def draw(self):
        self.texture.use(0)
        self._quad_fs.render(self.program)


class MyGame(arcade.Window):

    def __init__(self, width, height, title):
        super().__init__(width, height, title)
        self.filter = RandomFilter(width, height)

    def on_draw(self):
```

(continues on next page)

(continued from previous page)

```
        self.clear()
        self.filter.clear()
        self.filter.use()
        arcade.draw_circle_filled(self.width / 2, self.height / 2, 100, arcade.color.RED)
        arcade.draw_circle_filled(400, 300, 100, arcade.color.GREEN)

        self.use()
        self.filter.draw()


def main():
    """ Main function """
    MyGame(SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_TITLE)
    arcade.run()


if __name__ == "__main__":
    main()

```

Now, color everything that doesn't have an alpha of zero as green:

Listing 3: Pass-through frame buffer

```
import arcade
from arcade.experimental.texture_render_target import RenderTargetTexture

SCREEN_WIDTH = 800
SCREEN_HEIGHT = 600
SCREEN_TITLE = "Starting Template Simple"


class RandomFilter(RenderTargetTexture):
    def __init__(self, width, height):
        super().__init__(width, height)
        self.program = self.ctx.program(
            vertex_shader="""
            #version 330

            in vec2 in_vert;
            in vec2 in_uv;
            out vec2 uv;

            void main() {
                gl_Position = vec4(in_vert, 0.0, 1.0);
                uv = in_uv;
            }
            """,
            fragment_shader="""
            #version 330

            uniform sampler2D texture0;
```

(continues on next page)

(continued from previous page)

```

            in vec2 uv;
            out vec4 fragColor;

            void main() {
                vec4 color = texture(texture0, uv);

                if (color.a > 0)
                    fragColor = vec4(0, 1, 0, 1.0);
                else
                    fragColor = vec4(0, 0, 0, 0);
            }
            """,
        )

    def use(self):
        self._fbo.use()

    def draw(self):
        self.texture.use(0)
        self._quad_fs.render(self.program)


class MyGame(arcade.Window):

    def __init__(self, width, height, title):
        super().__init__(width, height, title)
        self.filter = RandomFilter(width, height)

    def on_draw(self):
        self.clear()
        self.filter.clear()
        self.filter.use()
        arcade.draw_circle_filled(self.width / 2, self.height / 2, 100, arcade.color.RED)

        self.use()
        self.filter.draw()


def main():
    """ Main function """
    MyGame(SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_TITLE)
    arcade.run()


if __name__ == "__main__":
    main()
```

Something about passing uniform data to the shader:

Listing 4: Pass-through frame buffer

```python
import arcade
from arcade.experimental.texture_render_target import RenderTargetTexture

SCREEN_WIDTH = 800
SCREEN_HEIGHT = 600
SCREEN_TITLE = "Starting Template Simple"


class RandomFilter(RenderTargetTexture):
    def __init__(self, width, height):
        super().__init__(width, height)
        self.program = self.ctx.program(
            vertex_shader="""
            #version 330

            in vec2 in_vert;
            in vec2 in_uv;
            out vec2 uv;

            void main() {
                gl_Position = vec4(in_vert, 0.0, 1.0);
                uv = in_uv;
            }
            """,
            fragment_shader="""
            #version 330

            uniform sampler2D texture0;

            in vec2 uv;
            uniform vec4 my_color;
            out vec4 fragColor;

            void main() {
                vec4 color = texture(texture0, uv);

                if (color.a > 0)
                    fragColor = my_color;
                else
                    fragColor = vec4(0, 0, 0, 0);
            }
            """,
        )
        self.program["my_color"] = 1, 0, 1, 1

    def use(self):
        self._fbo.use()

    def draw(self):
        self.texture.use(0)
        self._quad_fs.render(self.program)
```

(continues on next page)

(continued from previous page)

```


class MyGame(arcade.Window):

    def __init__(self, width, height, title):
        super().__init__(width, height, title)
        self.filter = RandomFilter(width, height)

    def on_draw(self):
        self.clear()
        self.filter.clear()
        self.filter.use()
        arcade.draw_circle_filled(self.width / 2, self.height / 2, 100, arcade.color.RED)

        self.use()
        self.filter.draw()


def main():
    """ Main function """
    MyGame(SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_TITLE)
    arcade.run()


if __name__ == "__main__":
    main()
```

CHAPTER EIGHTEEN

DRAWING & USING SPRITES

Most games built with Arcade will use sprites and sprite lists to draw image data. This section of the programming guide will help you achieve that by covering:

- What sprites & sprite lists are
- The essentials of how to use them
- How to get started with images
- Non-drawing features such as collisions
- Overviews of various advanced techniques

Beginners should start by reading & following *What's a Sprite?* page (~10 minute read). If you get stuck, see *How to Get Help*.

18.1 Contents

18.1.1 What's a Sprite?

Each sprite describes where a game object is & how to draw it. This includes:

- Where it is in the world
- Where to find the image data
- How big the image should be

The rest of this page will explain using the `SpriteList` class to draw sprites to the screen.

18.1.2 Why SpriteLists?

They're How Hardware Works

Graphics hardware is designed to draw groups of objects at the same time. These groups are called **batches**.

Each `SpriteList` automatically translates every `Sprite` in it into an optimized batch. It doesn't matter if a batch has one or hundreds of sprites: it still takes the same amount of time to draw!

This means that using fewer batches helps your game run faster, and that you should avoid trying to draw sprites one at a time.

They Help Develop Games Faster

Sprite lists do more than just draw. They also have built-in features which save you time & effort, including:

- Automatically skipping off-screen sprites
- Collision detection
- Debug drawing for hit boxes

18.1.3 Drawing with Sprites and SpriteLists

Let's get to the example code.

There are 3 steps to drawing sprites with a sprite list:

1. Create a *SpriteList*
2. Create & append your *Sprite* instance(s) to the list
3. Call *draw()* on your SpriteList inside an *on_draw()* method

Here's a minimal example:

Listing 1: sprite_minimal.py

```python
"""
Minimal Sprite Example

Draws a single sprite in the middle screen.

If Python and Arcade are installed, this example can be run from the command line with:
python -m arcade.examples.sprite_minimal
"""
import arcade


class WhiteSpriteCircleExample(arcade.Window):

    def __init__(self):
        super().__init__(800, 600, "White SpriteCircle Example")
        self.sprites = None
        self.setup()

    def setup(self):
        # 1. Create the SpriteList
        self.sprites = arcade.SpriteList()

        # 2. Create & append your Sprite instance to the SpriteList
        self.circle = arcade.SpriteCircle(30, arcade.color.WHITE)  # 30 pixel radius␣
↪circle
        self.circle.position = self.width // 2, self.height // 2  # Put it in the middle
        self.sprites.append(self.circle)  # Append the instance to the SpriteList

    def on_draw(self):
        # 3. Call draw() on the SpriteList inside an on_draw() method
        self.sprites.draw()
```

(continues on next page)

(continued from previous page)

```python


if __name__ == "__main__":
    game = WhiteSpriteCircleExample()
    game.run()
```

Using Images with Sprites

Beginners should see the following to learn more, such as how to load images into sprites:

- *Arcade's Sprite examples*
- *Arcade's Simple Platformer Tutorial*
- The `Sprite` API documentation

Viewports, Cameras, and Screens

Intermediate users can move past the limitations of `arcade.Window` with the following classes:

- `arcade.Camera` (*examples*) to control which part of game space is drawn
- `arcade.View` (*examples*) for start, end, and menu screens

18.1.4 Advanced SpriteList Techniques

This page provides overviews of advanced techniques. Runnable examples are not guaranteed, as the reader is expected to be able to put the work into implementing them.

Beginners should be careful of the following sections. Some of these techniques can slow down or crash your game if misused.

Draw Order & Sorting

In some cases, you can combine two features of SpriteList:

- By default, SpriteLists draw starting from their lowest index.
- `SpriteList` has a `sort()` method nearly identical to `list.sort()`.

First, Consider Alternatives

Sorting in Python is a slow, CPU-bound function. Consider the following techniques to eliminate or minimize this cost:

- Use multiple sprite lists or `arcade.Scene` to achieve layering
- Chunk your game world into smaller regions with sprite lists for each, and only sort when something inside moves or changes
- Use the `Sprite.depth` attribute with *shaders* to sort on the GPU

For a conceptual overview of chunks as used in a commercial 2D game, please see the following:

- Chunks in Factorio

Sorting SpriteLists

Although the alternative listed above are often better, sorting sprite lists to control draw order can still be useful.

Like Python's built-in `list.sort()`, you can pass a callable object via the key argument to specify how to sort, along with an optional `reverse` keyword to reverse the direction of sorting.

Here's an example of how you could use sorting to quickly create an inefficient prototype:

```python
import random
import arcade


# Warning: the bottom property is extra slow compared to other attributes!
def bottom_edge_as_sort_key(sprite):
    return sprite.bottom


class InefficientTopDownGame(arcade.Window):
    """
    Uses sorting to allow the player to move in front of & behind shrubs

    For non-prototyping purposes, other approaches will be better.
    """

    def __init__(self, num_shrubs=50):
        super().__init__(800, 600, "Inefficient Top-Down Game")

        self.background_color = arcade.color.SAND
        self.shrubs = arcade.SpriteList()
        self.drawable = arcade.SpriteList()

        # Randomly place pale green shrubs around the screen
        for i in range(num_shrubs):
            shrub = arcade.SpriteSolidColor(20, 40, color=arcade.color.BUD_GREEN)
            shrub.position = random.randrange(self.width), random.randrange(self.height)
            self.shrubs.append(shrub)
            self.drawable.append(shrub)

        self.player = arcade.SpriteSolidColor(16, 30, color=arcade.color.RED)
        self.drawable.append(self.player)

    def on_mouse_motion(self, x, y, dx, dy):
        # Update the player position
        self.player.position = x, y
        # Sort the sprites so the highest on the screen draw first
        self.drawable.sort(key=bottom_edge_as_sort_key, reverse=True)

    def on_draw(self):
        self.clear()
        self.drawable.draw()


game = InefficientTopDownGame()
```

(continues on next page)

(continued from previous page)

```
game.run()
```

Custom Texture Atlases

A `TextureAtlas` represents *`Texture`* data packed side-by-side in video memory. As textures are added, the atlas grows to fit them all into the same portion of your GPU's memory.

By default, each *`SpriteList`* uses the same default atlas. Use the `atlas` keyword argument to specify a custom atlas for an instance.

This is especially useful to prevent problems when using large or oddly shaped textures.

Please see the following for more information:

- *Custom Atlas*
- The `TextureAtlas` API documentation

Lazy SpriteLists

You can delay creating the OpenGL resources for a *`SpriteList`* by passing `lazy=True` on creation:

```
sprite_list = SpriteList(lazy=True)
```

The SpriteList won't create the OpenGL resources until forced to by one of the following:

1. The first *`SpriteList.draw()`* call on it
2. *`SpriteList.initialize()`*
3. GPU-backed collisions, if enabled

This behavior is most useful in the following cases:

Case	Primary Purpose
Creating SpriteLists before a Window	CPU-only unit tests which never draw
Parallelized SpriteList creation	Faster loading & world generation via `threading` or `subprocess` & `pickle`

Parallelized Loading

To increase loading speed & reduce stutters during gameplay, you can run pre-gameplay tasks in parallel, such as pre-generating maps or pre-loading assets from disk into RAM.

Warning: Only the main thread is allowed to access OpenGL!

Attempting to access OpenGL from non-main threads will raise an OpenGL Error!

To safely implement parallel loading, you will want to use the following general approach before allowing gameplay to begin:

1. Pass `lazy=True` when creating *`SpriteList`* instances in your loading code as described above

2. Sync the SpriteList data back to the main thread or process once loading is finished
3. Inside the main thread, call *`Spritelist.initialize()`* on each sprite list once it's ready to allocate GPU resources

Very advanced users can use `subprocess` to create SpriteLists inside another process and the `pickle` module to help pass data back to the main process.

Please see the following for additional information:

- *Arcade's OpenGL notes* for arcade-specific threading considerations
- Python's `threading` documentation
- Python's `subprocess` and `pickle` documentation

18.2 I'm Impatient!

Beginners should at least skim *What's a Sprite?* (~10 minute read), but you can skip to the tutorials and full example code if you'd like:

- *Drawing with Sprites and SpriteLists*
- *Arcade's Sprite Examples*
- *Arcade's Simple Platformer Tutorial*

CHAPTER

NINETEEN

KEYBOARD

19.1 Events

19.1.1 What is a keyboard event?

Keyboard events are arcade's representation of physical keyboard interactions.

For example, if your keyboard is working correctly and you type the letter A into the window of a running arcade game, it will see two separate events:

1. a key press event with the key code for `A`
2. a key release event with the key code for `A`

19.1.2 How do I handle keyboard events?

You must implement key event handlers. These functions are called whenever a key event is detected:

- `arcade.Window.on_key_press()`
- `arcade.Window.on_key_release()`

You need to implement your own versions of the above methods on your subclass of `arcade.Window`. The *arcade.key* module contains constants for specific keys.

For runnable examples, see the following:

- sprite_move_keyboard
- sprite_move_keyboard_better
- sprite_move_keyboard_accel

Note: If you are using `Views`, you can also implement key event handler methods on them.

19.2 Modifiers

19.2.1 What is a modifier?

Modifiers are keys that modify the behavior of keyboard input. Examples include keys such as shift, control, and command. Lock keys such as capslock are also modifiers.

19.2.2 What does active mean?

Modifiers can be active in two ways:

1. A modifier key is currently held down by the user (example: shift)
2. A lock modifier is currently turned on (example: capslock)

This is important because lock modifiers can be active without their corresponding key held down. Instead, they are switched on and off by pressing their keys.

19.2.3 How do I use modifiers?

As long as you don't need to distinguish between the left and right versions of modifiers keys, you can rely on the `modifiers` argument of *key event handlers*.

For every key event, the current state of all modifiers is passed to the handler method through the `modifiers` argument as a single integer. For each active modifier during an event, a corresponding bit is set to 1.

Constants for each of these bits are defined in *arcade.key*:

```
MOD_SHIFT
MOD_CTRL
MOD_ALT         Not available on Mac OS X
MOD_WINDOWS     Available on Windows only
MOD_COMMAND     Available on Mac OS X only
MOD_OPTION      Available on Mac OS X only
MOD_CAPSLOCK
MOD_NUMLOCK
MOD_SCROLLLOCK
MOD_ACCEL       Equivalent to MOD_CTRL, or MOD_COMMAND on Mac OS X.
```

You can use these constants with bitwise operations to check if a specific modifier is active during a keyboard event:

```python
# this should be implemented on a subclass of Window or View
def on_key_press(self, symbol, modifiers):

    if modifiers & arcade.key.MOD_SHIFT:
        print("The shift key is held down")

    if modifiers & arcade.key.MOD_CAPSLOCK:
        print("Capslock is on")
```

19.2.4 How do I tell left & right modifers apart?

Many keyboards have both left and right versions of modifiers such as shift and control. However, the `modifiers` argument to key handlers does not tell you which specific modifier keys are currently pressed!

Instead, you have to use specific key codes for left and right versions from *arcade.key* to *track press and release events*.

CHAPTER TWENTY

SOUND

This page will help you get started by covering the essentials of sound.

In addition each section's concepts, there may also be links to example code and documentation.

I'm Impatient!

Users who want to skip to example code should consult the following:

1. sound_demo
2. sound_speed_demo
3. music_control_demo
4. *Platformer Tutorial - Step 9 - Adding Sound*

20.1 Why Is Sound Important?

Sound helps players make sense of what they see.

For example, have you ever run into one of these common problems?

- Danger you never knew was there
- A character whose reaction seemed unexpected or out of place
- Items or abilities which appeared similar, but were very different
- An unclear warning or confirmation dialog

How much progress did it cost you? A few minutes? The whole playthrough? More importantly, how did you feel? You probably didn't want to keep playing.

You can use sound to prevent moments like these. In each example above, the right audio can provide the information players need for the game to feel fair.

20.2 Sound Basics

20.2.1 Loading Sounds

Before you can play a sound, you need to load its data into memory.

Arcade provides two ways to do this. Both accept the same arguments and return an *arcade.Sound* instance.

The easiest way is to use *arcade.load_sound()*:

```
import arcade

# You can pass strings containing a built-in resource handle,
hurt_sound = arcade.load_sound(":resources:sounds/hurt1.wav")
# a pathlib.Path,
pathlib_sound = arcade.load_sound(Path("imaginary\\windows\\path\\file.wav"))
# or an ordinary string describing a path.
string_path_sound = arcade.load_sound("imaginary/mac/style/path.wav")
```

If you prefer a more object-oriented style, you can create *Sound* instances directly:

```
from arcade import Sound  # You can also use arcade.Sound directly

# Although Sound accepts the same arguments as load_sound,
# only the built-in resource handle is shown here.
hurt_sound = Sound(":resources:sounds/hurt1.wav")
```

See the following to learn more:

1. *Built-In Resources*
2. `pathlib`
3. *Streaming or Static Loading?*

20.2.2 Playing Sounds

There are two easy ways to play a *Sound* object.

One is to call *Sound.play* directly:

```
self.hurt_player = hurt_sound.play()
```

The other is to pass a *Sound* instance as the first argument of *arcade.play_sound()*:

```
# Important: this *must* be a Sound instance, not a path or string!
self.hurt_player = arcade.play_sound(hurt_sound)
```

Both return a `pyglet.media.player.Player`. You should store it somewhere if you want to be able to stop or alter a specific playback of a *Sound*'s data.

`arcade.Sound` vs pyglet's `Player`

This is a very important distinction:

- An *`arcade.Sound`* is a source of audio data in memory
- Starting a playback of audio data returns a new pyglet `Player` which controls that specific playback

Imagine you have two non-player characters (NPCs) in a game which both play the same selection of *`Sound`* data. Since they are separate characters in the world, their playbacks of the data must be independent. To do this, each NPC will keep the pyglet `Player` returned when they start playing a sound.

For example, an NPC may get close enough to the user's character to talk, attack, or perform some other action which requires playing a different sound. You would handle this as follows:

1. Use the approaching NPC's pyglet `Player` to stop its current playback
2. If the NPC starts playing a different sound, store the returned pyglet `Player`

This is especially important when a dangerous NPC or other hazard can be invisible. Making invisible hazards play sounds is one of the easiest and most popular ways of making their gameplay feel balanced, fair, and fun.

See the following to learn more:

1. *Why Is Sound Important?*
2. sound_demo

20.2.3 Stopping Sounds

Arcade's helper functions are the easiest way to stop playback. To use them:

1. Do one of the following:
   - Pass the stored pyglet `Player` to *`arcade.stop_sound()`*:
     ```
     arcade.stop_sound(self.current_playback)
     ```
   - Pass the stored pyglet `Player` to the sound's *`stop()`* method:
     ```
     self.hurt_sound.stop(self.current_playback)
     ```
2. Clear any references to the player to allow its memory to be freed:
   ```
   # For each object, Python tracks how many other objects use it. If
   # nothing else uses an object, it will be marked as garbage which
   # Python can delete automatically to free memory.
   self.current_playback = None
   ```

See the following to learn more:

- *The Most Reliable Formats & Features*
- *Advanced Playback Control*

20.3 Streaming or Static Loading?

Streaming	Best[1] Format	Decompressed	Best Uses
`False` (Default)	`.wav`	Whole file	2+ overlapping playbacks, short, repeated, unpredictable
`True`	`.mp3`	Predicted data	1 copy & file at a time, long, uninterrupted

By default, arcade decompresses the entirety of each sound into memory.

This is the best option for most game sound effects. It's called "static"[2] audio because the data never changes.

The alternative is streaming. Enable it by passing `True` through the `streaming` keyword argument when you *load a sound*:

```
# Both loading approaches accept the streaming keyword.
classical_music_track = arcade.load_sound(":resources:music/1918.mp3", streaming=True)
funky_music_track = arcade.Sound(":resources:music/funkyrobot.mp3", streaming=True)
```

For an interactive example, see the music_control_demo.

The following subheadings will explain each option in detail.

20.3.1 Static Sounds are for Speed

Static sounds can help your game run smoothly by preloading data before gameplay.

This is because disk access is one of the slowest things a computer can do. Waiting for sounds to load during gameplay can make the your game run slowly or stutter. The best way to prevent this is to load your sound data ahead of time. Popular approaches for this include:

- Loading screens
- Small inter-level "rooms"
- Multi-threading (best used by experienced programmers)

Unless music is a central part of your gameplay, you should avoid storing fully decompressed albums of music in RAM. Each decompressed minute of CD quality audio uses slightly over 10 MB of RAM. This adds up quickly, and can slow down or freeze a computer if it fills RAM completely.

For music and long background audio, you should should strongly consider *streaming* from compressed files instead.

When to Use Static Sounds

If an audio file meets one or more of the following conditions, you may want to load it as static audio:

- You need to start playback quickly in response to gameplay.
- Two or more "copies" of the sound can be playing at the same time.
- You will unpredictably skip to different times in the file.
- You will unpredictably restart playback.
- You need to automatically loop playback.
- The file is a short clip.

[1] See *The Most Reliable Formats & Features* to learn more.
[2] See the `pyglet.media.StaticSource` class used by arcade.

20.3.2 Streaming Saves Memory

Streaming audio from files is very similar to streaming video online.

Both save memory by keeping only part of a file into memory at any given time. Even on the slowest recent hardware, this usually works if:

- You only stream one media source at a time.
- You don't need to synchronize it closely with anything else.

When to Stream

The best way to use streaming is to only use it when you need it.

Advanced users may be able to handle streaming multiple tracks at a time. However, issues with synchronization & interruptions will grow with the quantity and quality of the audio tracks involved.

If you're unsure, avoid streaming unless you can say yes to all of the following:

1. The *`Sound`* will have at most one playback at a time.
2. The file is long enough to make it worth it.
3. Seeking (skipping to different parts) will be infrequent.
    - Ideally, you will never seek or restart playback suddenly.
    - If you do seek, the jumps will ideally be close enough to land in the same or next chunk.

See the following to learn more:

- *Change Ongoing Playbacks via Player Objects*
- The `pyglet.media.StreamingSource` class used to implement streaming

Streaming Can Cause Freezes

Failing to meet the requirements above can cause buffering issues.

Good compression on files can help, but it can't fully overcome it. Each skip outside the currently loaded data requires reading and decompressing a replacement.

In the worst-case scenario, frequent skipping will mean constantly buffering instead of playing. Although video streaming sites can downgrade quality, your game will be at risk of stuttering or freezing.

The best way to handle this is to only use streaming when necessary.

20.4 Advanced Playback Control

Arcade's functions for *Stopping Sounds* are convenience wrappers around the passed pyglet `Player`.

You can alter a playback of *`Sound`* data with more precision by:

- Using the properties and methods of its `Player` any time before playback has finished
- Passing keyword arguments with the same (or similar) names as the Player's properties when *playing the sound*.

20.4.1 Stopping via the Player Object

The simplest form of advanced control is pausing and resuming playback.

Pausing

There is no stop method. Instead, call the `Player.pause()` method:

```
# Assume this is inside an Enemy class subclassing arcade.Sprite
self.current_player.pause()
```

Stopping Permanently

After you've paused a player, you can stop playback permanently:

1. Call the player's `delete()` method:

   ```
   # Permanently deletes the operating system half of this playback.
   self.current_player.delete()
   ```

   This specific playback is now permanently over, but you can start new ones.

2. Make sure all references to the player are replaced with `None`:

   ```
   # Python will delete the pyglet Player once there are 0 references to it
   self.current_player = None
   ```

For a more in-depth explanation of references and auto-deletion, skim the start of Python's page on garbage collection. Reading the Abstract section of this page should be enough to get started.

20.4.2 Changing Aspects of Playback

There are more ways to alter playback than stopping. Some are more qualitative. Many of them can be applied to both new and ongoing sound data playbacks, but in different ways.

Change Ongoing Playbacks via Player Objects

`Player.pause()` is one of many method and property members which change aspects of an ongoing playback. It's impossible to cover them all here, especially given the complexity of *positional audio*.

Instead, the table below summarizes a few of the most useful members in the context of arcade. Superscripts link info about potential issues, such as name differences between properties and equivalent keyword arguments to arcade functions.

Player Member	Type	Default	Purpose
pause()	method	N/A	Pause playback resumably.
play()	method	N/A	Resume paused playback.
seek()	method	N/A **Warning:** *Using this option with streaming can cause freezes!*	Skip to the passed float timestamp measured as seconds from the audio's start.
volume	float property	1.0	The scaling factor to apply to the original audio's volume. Must be between 0.0 (silent) and 1.0 (full volume).
loop[3]	bool property	False	Whether to restart playback automatically after finishing.[4]
pitch[5]	float property	1.0	How fast to play the sound data; also affects pitch.

Configure New Playbacks via Keyword Arguments

Arcade's helper functions for playing sound also accept keyword arguments for configuring playback. As mentioned above, the names of these keywords are similar or identical to those of properties on Player. See the following to learn more:

- *arcade.play_sound()*
- *Sound.play()*
- sound_speed_demo

[3] *arcade.play_sound()* uses looping instead. See:

- *Configure New Playbacks via Keyword Arguments*
- The related GitHub issue.

[4] Looping is unavailable when streaming=True; see pyglet's guide to controlling playback.

[5] Arcade's equivalent keyword for *Playing Sounds* is speed

20.5 Cross-Platform Compatibility

The sections below cover the easiest approach to compatibility.

You can try other options if you need to. Be aware that doing so requires grappling with the many factors affecting audio compatibility:

1. The formats which can be loaded
2. The features supported by playback
3. The hardware, software, and settings limitations on the first two
4. The interactions of project requirements with all of the above

20.5.1 The Most Reliable Formats & Features

For most users, the best approach to formats is:

- Use 16-bit PCM Wave (`.wav`) files for *sound effects*
- Use MP3 files for *long background audio like music*

As long as a user has working audio hardware and drivers, the following basic features should work:

1. *Loading Sounds* sound effects from Wave files
2. *Playing Sounds* and *Stopping Sounds*
3. *Adjusting playback volume and speed of playback*

Advanced functionality or subsets of it may not, especially *positional audio*. To learn more, see the rest of this page and pyglet's guide to supported media types.

Why 16-bit PCM Wave for Effects?

Storing sound effects as 16-bit PCM `.wav` ensures all users can load them:

1. pyglet *has built-in in support for this format*
2. *Some platforms can only play 16-bit audio*

The files must also be mono rather than stereo if you want to use *positional audio*.

Accepting these limitations is usually worth the compatibility benefits, especially as a beginner.

Why MP3 For Music and Ambiance?

1. Nearly every system which can run arcade has a supported MP3 decoder.
2. MP3 files are much smaller than Wave equivalents per minute of audio, which has multiple benefits.

See the following to learn more:

- *Loading In-Depth*
- Pyglet's Supported Media Types

Converting Audio Formats

Don't worry if you have a great sound in a different format.

There are multiple free, reliable, open-source tools you can use to convert existing audio. Two of the most famous are summarized below.

Name & Link for Tool	Difficulty	Summary
Audacity	Beginner[6]	A free GUI application for editing sound
FFmpeg's command line tool	Advanced	Powerful media conversion tool included with the library

Most versions of these tools should handle the following common tasks:

- Converting audio files from one encoding format to another
- Converting from stereo to mono for use with *positional audio*.

To integrate FFmpeg with Arcade as a decoder, you must use FFmpeg version 4.X, 5.X, or 6.X. See *Loading In-Depth* to learn more.

20.5.2 Loading In-Depth

There are 3 ways arcade can read audio data through pyglet:

1. The built-in pyglet `.wav` loading features
2. Platform-specific components or nearly-universal libraries
3. Supported cross-platform media libraries, such as PyOgg or FFmpeg

To load through FFmpeg, you must install FFmpeg 4.X, 5.X, or 6.X. This is a requirement imposed by pyglet. See pyglet's notes on installing FFmpeg to learn more.

Everyday Usage

In practice, Wave is universally supported and MP3 nearly so.[7]

Limiting yourself to these formats is usually worth the increased compatibility doing so provides. Benefits include:

1. Smaller download & install sizes due to having fewer dependencies
2. Avoiding binary dependency issues common with PyInstaller and Nuitka
3. Faster install and loading, especially when using MP3s on slow drives

These benefits become even more important during game jams.

[6] Linux users may need to install the LAME MP3 encoder separately to export MP3 files.
[7] The only time MP3 will be absent is on unusual Linux configurations. See pyglet's guide to supported media types to learn more.

20.5.3 Backends Determine Playback Features

As with formats, you can maximize compatibility by only using the lowest common denominators among features. The most restrictive backends are:

- Mac's only backend, an OpenAL version limited to 16-bit audio
- PulseAudio on Linux, which has multiple limitations:
  - It lacks support for *positional audio*
  - It can crash under certain circumstances when other backends will not:
    - Pausing / resuming in debuggers
    - Rarely and unpredictably when multiple sounds are playing

On Linux, the best way to deal with the PulseAudio bug is to install OpenAL. It will often already be installed as a dependency of other packages.

Other differences between backends are less drastic. Usually, they will be things like the specific positional features supported and the maximum number of simultaneous sounds.

See the following to learn more:

- Pyglet's Audio Backends
- *Other Sound Libraries*

20.5.4 Choosing the Audio Backend

By default, arcade will try pyglet audio back-ends in the following order until it finds one which loads:

1. `"openal"`
2. `"xaudio2"`
3. `"directsound"`
4. `"pulse"`
5. `"silent"`

You can override through the `ARCADE_SOUND_BACKENDS` environment variable. The following rules apply to its value:

1. It must be a comma-separated string
2. Each name must be an audio back-ends supported by pyglet
3. Spaces do not matter and will be ignored

For example, you could need to test OpenAL on a specific system. This example first tries OpenAL, then gives up instead using fallbacks.

```
ARCADE_SOUND_BACKENDS="openal,silent" python mygame.py
```

Please see the following to learn more:

- pyglet's audio driver documentation
- Working with Environment Variables in Python

20.6 Other Sound Libraries

Advanced users may have reasons to use other libraries to handle sound.

20.6.1 Using Pyglet

The most obvious external library for audio handling is pyglet:

- It's guaranteed to work wherever arcade's sound support does.
- It offers far better control over media than arcade
- You may have already used parts of it directly for *Advanced Playback Control*

Note that `arcade.Sound`'s `source` attribute holds a `pyglet.media.Source`. This means you can start off by cleanly using arcade's resource and sound loading with pyglet features as needed.

Notes on Positional Audio

Positional audio is a set of features which automatically adjust sound volumes across the channels for physical speakers based on in-game distances.

Although pyglet exposes its support for this through its `Player`, arcade does not currently offer integrations. You will have to do the setup work yourself.

If you already have some experience with Python, the following sequence of links should serve as a primer for trying positional audio:

1. *Why 16-bit PCM Wave for Effects?*
2. *Backends Determine Playback Features*
3. The following sections of pyglet's media guide:
    1. Controlling playback
    2. Positional audio
4. `pyglet.media.player.Player`'s full documentation

20.6.2 External Libraries

Some users have reported success with using PyGame CE or SDL2 to handle sound. Both these and other libraries may work for you as well. You will need to experiment since this isn't officially supported.

CHAPTER TWENTYONE

TEXTURES

21.1 Introduction

The *arcade.Texture* type is how arcade normally interacts with images either loaded from disk or created manually. This is basically a wrapper for PIL/Pillow images including detection for hit box data using pymunk depending on the selected hit box algorithm. These texture objects are in other words responsible to provide raw RGBA pixel data to OpenGL and hit box geometry to the sprite engine.

There is another texture type in Arcade in the lower level OpenGL API: `arcade.gl.Texture`. This represents an actual OpenGL texture and should only be used when dealing with the low level rendering API `arcade.gl`.

Textures can be created/loaded before or after the window is created because they don't interact with OpenGL directly.

21.2 Texture Uniqueness

When a texture is created a `name` is required. This should be a unique string. If two more more textures have the same name we will run into trouble. When loading textures the absolute path to the file is used as part of the name including vertical/horizontal/diagonal, size and other parameter for a truly unique name.

When loading texture through arcade the name of the texture will be the absolute path to the image and various parameters such as size, flipping, xy position etc.

Also remember that the texture class do hit box detection with pymunk by looking at the raw pixel data. This means for example a texture with different flipping will be loaded multiple times (or fetched from cache) because we rely in the transformed pixel data to get the hit box.

21.3 Texture Cache

Arcade is caching texture instances based on the `name` attribute to significantly speed up loading times.

```python
# The texture will only be loaded during the first sprite creation
tex_name = "path/to/sprite.png"
sprite_1 = arcade.Sprite(tex_name)
sprite_2 = arcade.Sprite(tex_name)
sprite_3 = arcade.Sprite(tex_name)
# Will be loaded and cached because we need fresh pixel data for hit box detection
sprite_4 = arcade.Sprite(tex_name, flipped_vertically=True)
# Fetched from cache
sprite_5 = arcade.Sprite(tex_name, flipped_vertically=True)
```

The above also applies when using `arcade.load_texture()` or other texture loading functions.

Arcade's texture cache can be cleared using `arcade.cleanup_texture_cache()`.

21.4 Custom Textures

We can manually create textures by creating PIL/Pillow images. How this is done is entirely up to you. Using the drawing functionality of Pillow or simply providing raw pixel data from another library/source into a Pillow image. A random example is getting raw pixel data from matplotlib.

```python
# Create a image from raw pixel data from some source
image = PIL.Image.frombuffer(raw_data)

# NOTE: Also make sure you use a sane hit_box_algorithm
texture = arcade.Texture("unique_name", image, hit_box_algorithm=...)
```

Again, how you create the image is up to you. There are many possibilities with Pillow.

CHAPTER

TWENTYTWO

SECTIONS

In a simple game, the whole viewport is used to display the game "map". In more advanced games it's fairly normal to have this viewport divided into different "sections" with different usages. Areas where different information is displayed and processed. For example you can have a menu at the top, some info panel at the right and the game main "screen" (the "map") covering the rest of the viewport.

To achieve this separation of game logic you have Sections. A `Section` is a way to divide a `View` space into smaller parts, each one will then receive events redirected depending on configuration and the space of the view occupied. Sections can isolate code that otherwise goes packed together in a `View` . This way the code remains exactly where it belongs and not mixed together with code from other parts of the program.

By configuring a `Section` you can capture some events or for example only capture certain keys from keyboard events. Also you can configure which events are propagated to other underlying sections or even to the view itself.

Sections can also be "modal" meaning that they will capture all the events first but draw last and also will prevent other views from receiving the `on_update` event.

Also note that if you don't use sections in your code, nothing changes. Even the `SectionManager` is not created if you don't add sections.

Key features of Sections:

- Divide the screen into logical components (Sections).
- Event dispatching: a `Section` will capture mouse events based on the space occupied from the view. Also keyboard events will be captured based on configuration.
- Prevent dispatching: a `Section` can be configured to prevent dispatching events captured or let events flow to other sections underneath.
- Event capturing order: based on a `Section` insertion order you can configure the order in which sections will capture events.
- Draw order: you can configure the order in which sections are drawn (sections can overlap!).
- `Section` "enable" property to show or hide sections. You can toogle that.
- Modal Sections: sections that draw last but capture all events and also stop other sections from updating.
- Automated camera swich: Sections will try to activate and deactivate cameras when changing between sections.

Important: You don't need to cover 100% of the View with sections. Sections can work with the View as well. Also, Sections can overlap.

22.1 A simple example

A small program without the use of sections needs to perform some checks inside a `on_mouse_release` event to know what to do depending on the mouse position.

For example maybe if the mouse is on top of the map you want to do something, but if the mouse is somewhere else you may need to do other things.

This is what this somehow looks without sections:

```python
class MyView(arcade.View):
    # ...

    def on_mouse_release(x: int, y: int, *args, **kwargs):
        if x > 700:
            # click in the side
            do_some_logic_when_side_clicking()
        else:
            # click on the game map
            do_something_in_the_game_map()
```

This code can and often become long and with a lot of checks to know what to do.

By using Sections, you can improve this code and automate this cimple checks.

This is what looks like using Sections:

```python
class Map(arcade.Section):

    # ...

    def on_mouse_release(x: int, y: int, *args, **kwargs):
        # clicks on the map are handled here
        pass


class Side(arcade.Section):

    # ...

    def on_mouse_release(x: int, y: int, *args, **kwargs):
        # clicks on the side of the screen are handled here
        pass


class MyView(arcade.View):

    def __init__(self, *args, **kwargs):
        self.map_section = Map(0, 0, 700, self.window.height)
        self.side_section = SideSpace(700, 0, 100, self.window.height)

        self.add_section(self.map_section)
        self.add_section(self.side_section)

    # ...
```

22.2 How to work with Sections

To work with sections you first need to have a `View`. Sections depend on Views and are handled by a special `SectionManager` inside the `View`. Don't worry, 99% of the time you won't need to interact with the `SectionManager`.

To create a `Section` start by inheriting from `arcade.Section`.

Based on the `Section` configuration your section will start receiving events from the View `SectionManager`. A `Section` has all the events a `View` has like `on_draw`, `on_update`, `on_mouse_press`, etc.

On instantiation define the positional arguments (left, bottom, width, height) of the section. These are very important properties of a `Section`: as they define the event capture rectangular area.

Properties of a `Section`:

position: (left, bottom, width, height):
This are mandatory arguments that you need to provide when instantiating a `Section`. This is very important as this rectangular positioning will determine the event capture space for mouse related events. This also will help you determine inside a class the space that is holding for example when you want to draw something or calculate coordinates.

name:
A `Section` can optionally get a name so it will be easier to debug and indetify what Section is doing what. When logging for example is very nice to log the `Section` name at the beginnig so you have a reference from where the log was generated.

accept_keyboard_keys:
This allows to tell if a `Section` can receive keyboard events (accept_keyboard_keys=False) or to tell which keyboard keys are captured in this `Section` (accept_keyboard_keys={arade.key.UP, arcade.key.DOWN})

accept_mouse_events:
This allows to tell if a `Section` can receive mouse events or which mouse events are accepted. For example: accept_mouse_events={'on_mouse_move'} means only mouse move events will be captured.

prevent_dispatch:
This tells a `Section` if it should prevent the dispatching of certain events to other sections down event capture stream. By default a `Section` will prevent dispatching all handled events. By passing `prevent_dispatch={'on_mouse_press'}` all events will propagate down the event capture stream except the `on_mouse_press` event. Note that passing `prevent_dispatch=None` (the default) is the same as passing `prevent_dispatch={True}` which means "prevent all events" from dispatching to other sections. You can also set `prevent_dispatch={False}` to dispatch all events to other sections.

prevent_dispatch_view:
This allows to tell a `Section` if events (and what events) should not be dispatched to the underlying `View`. This is handy if you want to do some action in the `View` code whether or not the event was handled by another `Section`. By default a `Section` will prevent dispatching all handled events to the `View`. Note that passing `prevent_dispatch=None` (the default) is the same as passing `prevent_dispatch={True}` which means "prevent all events" from dispatching to the view. You can also set `prevent_dispatch={False}` to dispatch all events to other sections. **Also note that in order for the view to receive any event, ALL the sections need to allow the dispatch of that particular event. If at least one section prevents it, the event will not be delivered to the view.**

local_mouse_coordinates:
If True the section mouse events will receive x, y coordinates section related to the section dimensions and position (not related to the screen). **Note that although this seems very usefull, section local coordinates doesn't work with arcade collision methods. You can use Section ``get_xy_screen_relative`` to transform local mouse coordinates to screen coordinates that work with arcade collision methods**

enabled:
By default all sections are enabled. This allows to tell if this particuar `Section` should be enabled or not. If a `Section` is not enabled, it will not capture any event, draw, update, etc. It will be as it didn't exist. You can enable and disable sections at any time allowing some cool efects. Nota that setting this property will trigger the section `on_show_section` or `on_hide_section` events.

modal:
This tells the `SectionManager` that this `Section` is modal. This means that the `Section` will capture all events first and not deliver any events to the underlying sections or view. Also, It will draw last (on top of other `on_draw` calls). When enabled a modal `Section` will prevent all other sections from receive `on_update` events.

draw_order:
This allows to define the draw order this `Section` will have. The lower the number the earlier this section will get draw. This is handy when you have overlaping sections and you want some `Section` to be drawn ontop of another. By default sections will be draw in the order they are added (except modal sections which no matter what will be drawn last). Note that this can be different from the event capture order or the on_update order which is defined by the insertion order in the `SectionManager`.

Other handy `Section` properties:

- block_updates: if True this section will not have the `on_update` method called.
- camera: this is meant to hold a `arcade.Camera` but it is None by default. The SectionManager will trigger the use of the camera when is needed automatically.

Handy `Section`: methods:

- overlaps_with: this will tell if another `Section` overlaps with this one.
- mouse_is_on_top: this will tell if given a x, y coodinate, the mouse is on top of the section.
- get_xy_screen_relative: get screen x, y coordinates from x, y section coordinates.
- get_xy_section_relative: get section x, y coordinates from x, y screen coordinates.

22.3 Sections configuration and logic with an example

Imagine a game where you have this basic components:

- A 800x600 screen viewport
- A game map
- A menu bar at the top of the screen
- A side right panel with data from the game
- Popup messages (dialogs)

With this configuration you can divide this logic into sections with a some configuration.

Lets look what this configuration may look:

```python
import arcade


class Map(arcade.Section):
    #... define all the section logic


```

(continues on next page)

(continued from previous page)

```python
class Menu(arcade.Section):
    #... define all the section logic


class Panel(arcade.Section):
    #... define all the section logic


class PopUp(arcade.Section):
    def __init__(message, *args, **kwargs):
        super().__init(*args, **kwargs)
        self.message = message

    # define draw logic, etc...


class MyView(arcade.View):

    def __init__(self, *args, **kwargs):
        self.map = Map(left=0, bottom=0, width=600, height=550,
                       name='Map', draw_order=2)
        self.menu = Menu(left=0, bottom=550, width=800, height=50,
                         name='Menu', accept_keyboard_keys=False,
                         accept_mouse_events={'on_mouse_press'})
        self.panel = Panel(left=600, bottom=0, width=200, height=550,
                           name='Panel', accept_keyboard_keys=False,
                           accept_mouse_events=False)

        popup_left = (self.view.window.width // 2) - 200
        popup_bottom = (self.view.window.height // 2) - 100
        popup_width = 400
        popup_height = 200
        self.popup = PopUp(message='', popup_left, popup_bottom, popup_width,
                           popup_height, enabled=False, modal=True)

        self.add_section(self.map)
        self.add_section(self.menu)
        self.add_section(self.panel)
        self.add_section(self.popup)

    def close():
        self.popup.message = 'Are you sure you want to close the view?'
        self.popup.enabled = True
```

Lets go step by step. First we configure a Map section that will hold the map. This Section will start at left, bottom = 0,0 and will not occupy the whole screen. Mouse events that occur outside of this coordinates will not be handled by the Map event handlers. So Map will only need to take care of what happens inside the map.

Second we configure a Menu section that will hold some buttons. This menu takes the top space of the screen that the Map has left. The Map + the Menu will occupy 100% of the height of the screen. The menu section is configured to not receive any keyboard events and to only receive on_mouse_press events, ignoring all other type of mouse events.

Third, the Panel also doesn't receive keyboard events. So the Map is the only handling keyboard events at the moment. Also no mouse events are allowed in the panel. This panel is just to show data.

For the last part notice that we define a section that it will be disabled at first and that is modal. This section will render something with a message. The section is used when the close method of the view is called. Because PopUp is a modal section, when enabled it's rendered on top of everything. Also, all other section stoped updating and all events are captured by the modal section. So in brief we are "stopping" the world outside the popup section.

22.4 Section Unique Events

There a few unique events that belong to sections and are somehow special in the way they are triggered:

- **`on_mouse_enter` and `on_mouse_leave`:**
  These events are triggered on two ocasions: when the mouse enters/leaves the view and when the `SectionManager` detects by mouse motion (or dragging) that the mouse has enter / leaved the section dimensions.
- **`on_show_section` and `on_hide_section`:**
  There events are triggered only when the section **is enabled** and under certain circumstances that must be known:
  - When the section is added or removed from the `SectionManager` and the `View` is currently being shown
  - When the section is enabled or disabled
  - When Window calls `on_show_view` or `on_hide_view`

22.5 The Section Manager

Behind the scenes, when sections are added to the `View` the `SectionManager` is what will handle all events instead of the `View` itself.

You can access the `SectionManager` by accessing the `View.section_manager`. Note that if you don't use Sections, the section manager inside the View will not be used nor created.

Usually you won't need to work with the `SectionManager`, but there are some cases where you will need to work with it.

You add sections usually with `View.add_section` but the same method exists on the `SectionManager`. Also you have a `remove_section` and a `clear_sections` method.

You can `enable` or `disable` the `SectionManager` to completely enable or disable all sections at once.

There are some other functionality exposed from the `SectionManager` like `get_section_by_name` that can also be useful. Check the api to know about those.

Also there are three attributes that can be configured in the `SectionManager` that are useful and important sometimes.

By default, `on_draw`, `on_update` and `on_resize` are events that will always be triggered in the `View` before any section has triggered them. This is the default but you can configure this with the following attributes:

- `view_draw_first`
- `view_update_first`
- `view_resize_first`

Both three work the same way:

- True (default) to trigger that event in the `View` before the sections.
- False so it's triggered in the `View` after sections corresponding methods.

- None to not trigger that event in the `View` at all.

CHAPTER

TWENTYTHREE

GUI

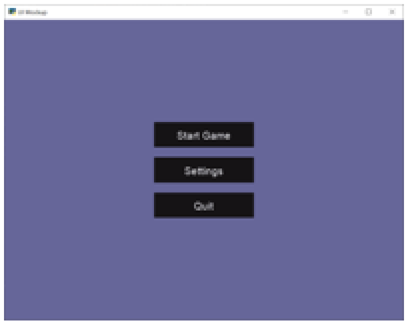


Fig. 1: gui_flat_button

Arcade's GUI module provides you classes to interact with the user using buttons, labels and much more.

Using those classes is way easier if the general concepts are known. It is recommended to read through them.

23.1 GUI Concepts

GUI elements are represented as instances of *`UIWidget`*. The GUI is structured like a tree; every widget can have other widgets as children.

The root of the tree is the *`UIManager`*. The `UIManager` connects the user interactions with the GUI. Read more about *User-interface events*.

Classes of arcade's GUI code are prefixed with `UI-` to make them easy to identify and search for in autocompletion.

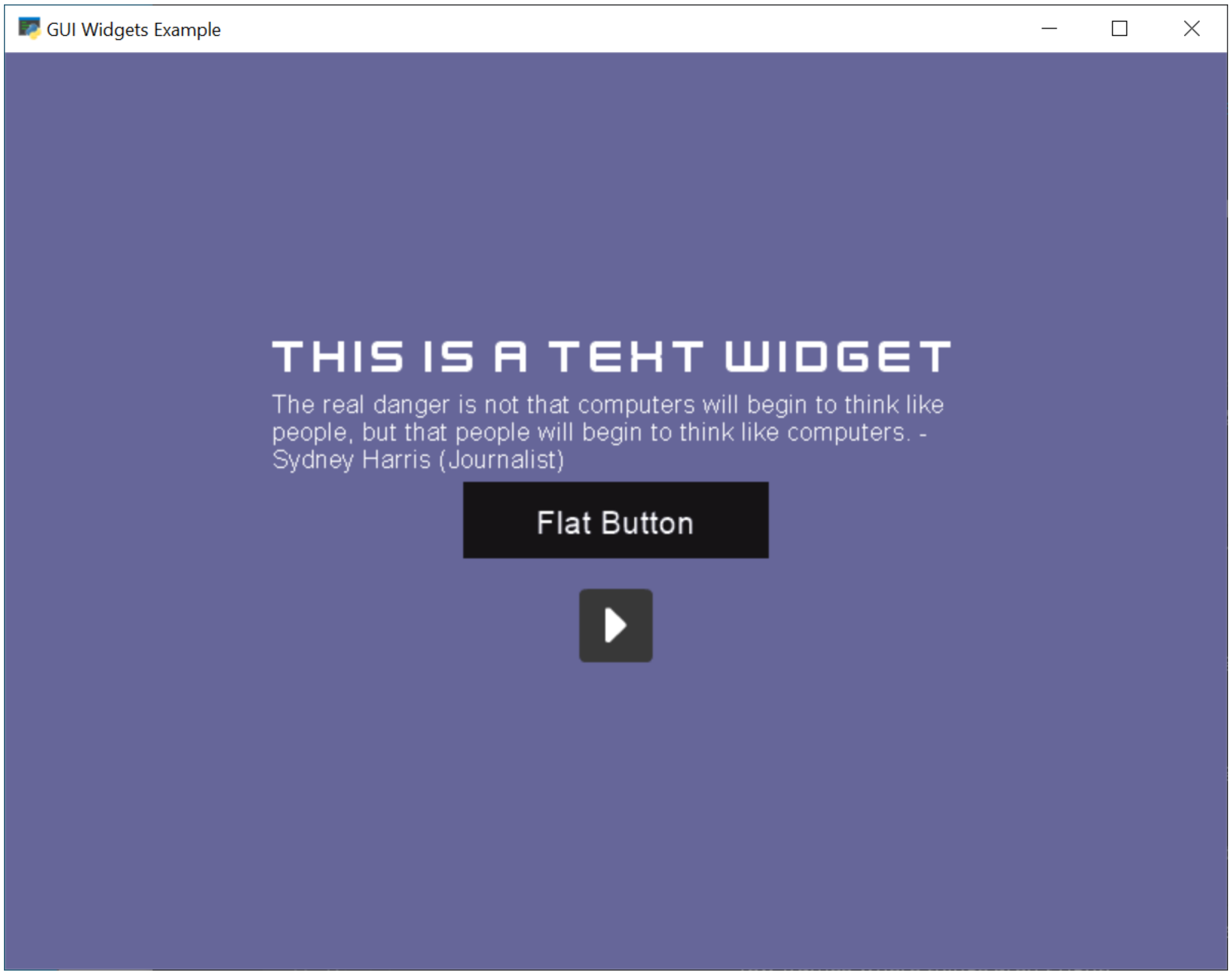
23.1.1 UIWidget

The *`UIWidget`* class is the core of arcade's GUI system. Widgets specify the behavior and graphical representation of any UI element, such as buttons or labels.

A `UIWidget` has following properties.

`rect`
: A tuple with four slots. The first two are x and y coordinates (bottom left of the widget), and the last two are width and height.

`children`
: Child widgets rendered within this widget. A `UIWidget` will not move or resize its children; use a *`UILayout`* instead.



THIS IS A TEXT WIDGET

The real danger is not that computers will begin to think like people, but that people will begin to think like computers. - Sydney Harris (Journalist)

Fig. 2: gui_widgets

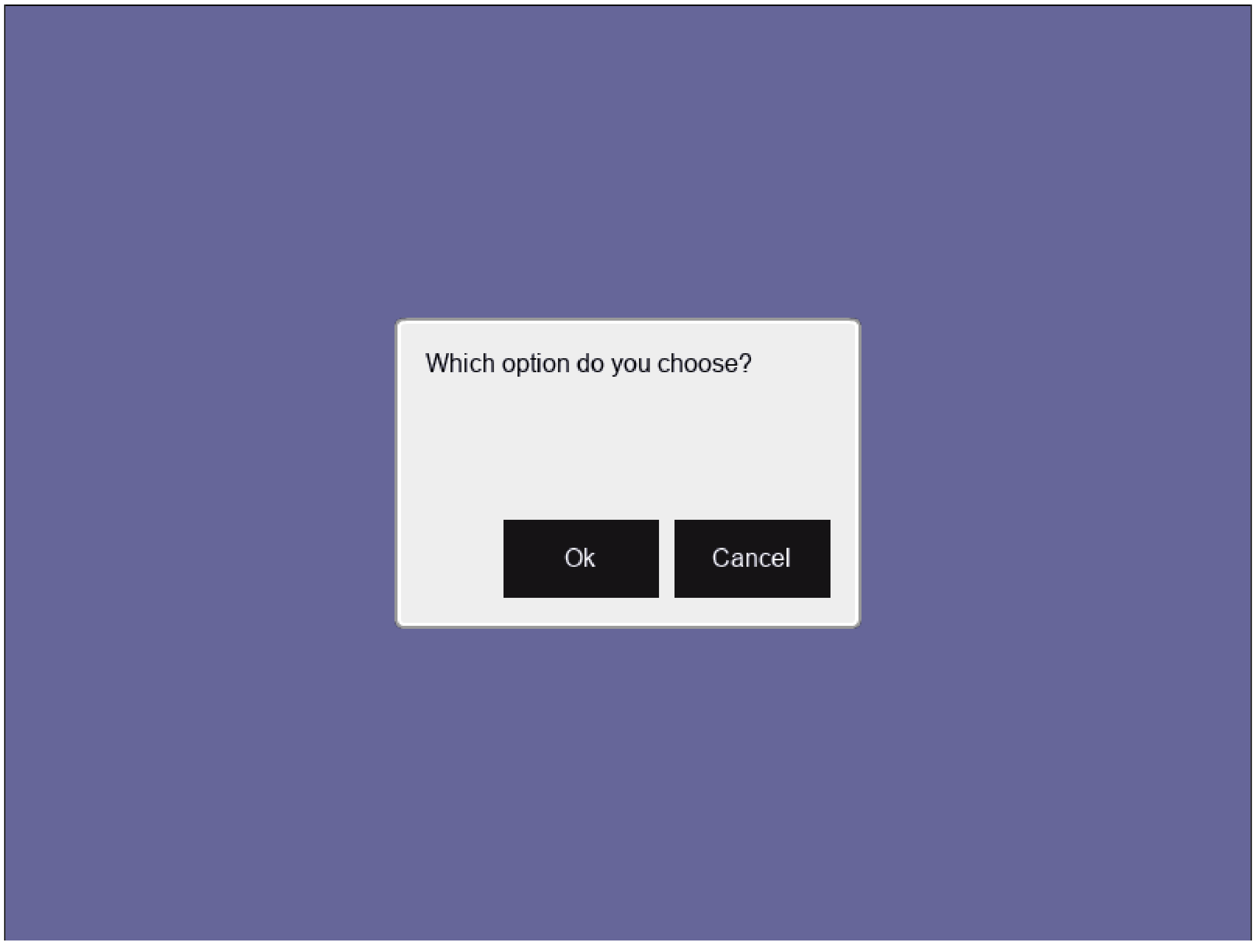
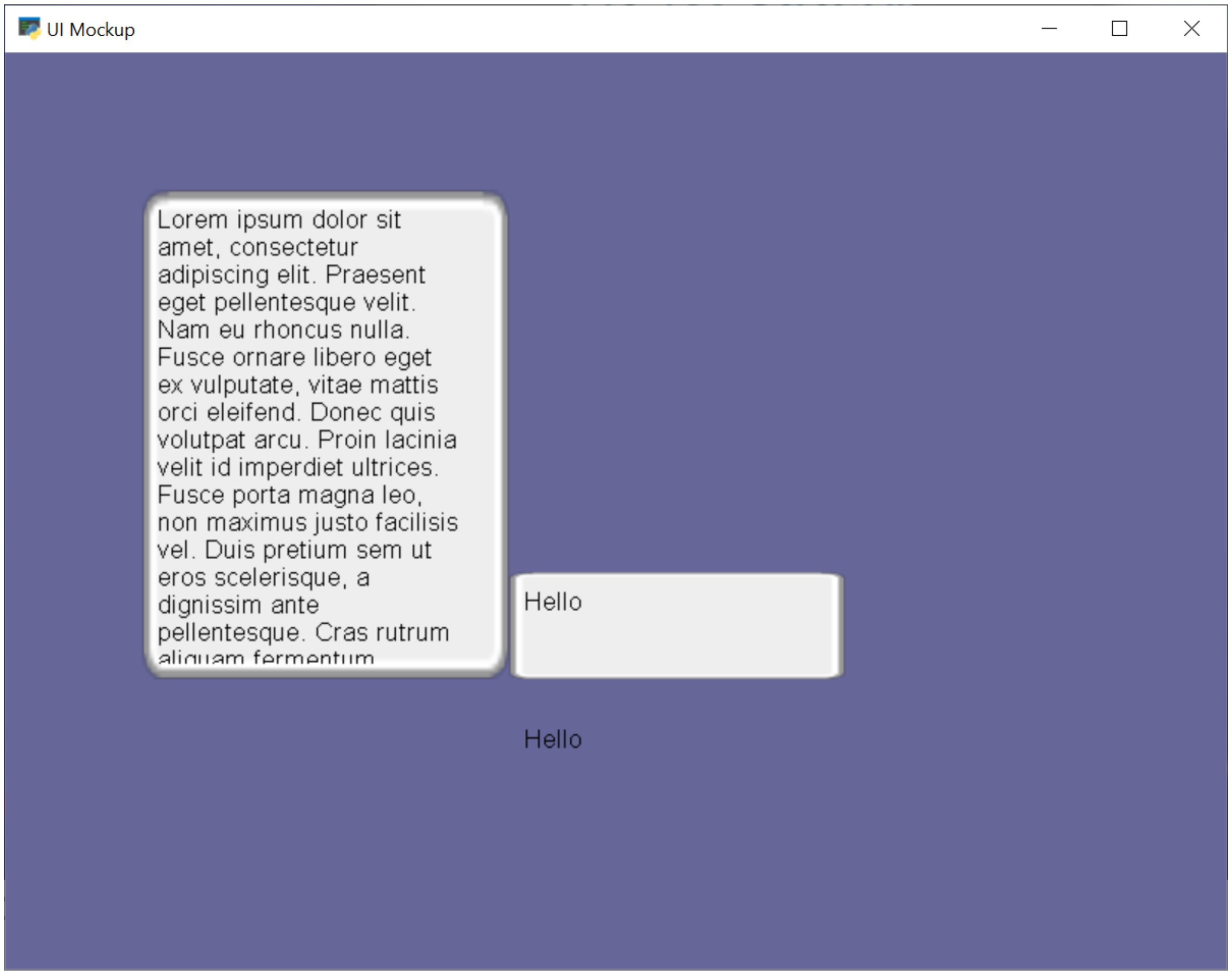


Fig. 3: gui_ok_messagebox



Lorem ipsum dolor sit amet, consectetur adipiscing elit. Praesent eget pellentesque velit. Nam eu rhoncus nulla. Fusce ornare libero eget ex vulputate, vitae mattis orci eleifend. Donec quis volutpat arcu. Proin lacinia velit id imperdiet ultrices. Fusce porta magna leo, non maximus justo facilisis vel. Duis pretium sem ut eros scelerisque, a dignissim ante pellentesque. Cras rutrum aliquam fermentum

Hello

Hello

Fig. 4: gui_scrollable_text

size_hint

A tuple of two normalized floats (0.0-1.0) describing the portion of the parent's width and height this widget prefers to occupy.

Examples:

```python
# Prefer to take up all space within the parent
widget.size_hint = (1.0, 1.0)

# Prefer to take up the full width & half the height of the parent
widget.size_hint = (1.0, 0.5)
# Prefer using 1/10th of the available width & height
widget.size_hint = (0.1, 0.1)
```

size_hint_min

A tuple of two integers defining the minimum width and height of the widget. Attempting to set a smaller width or height on the widget will fail by defaulting to the minimum values specified here.

size_hint_max

A tuple of two integers defining the maximum width and height of the widget. Attempting to set a larger width or height greater will fail by defaulting to the to the maximum values specified here.

Warning: Size hints do nothing on their own!

They are hints to `UILayout` instances, which may choose to use or ignore them.

Rendering

do_render() is called recursively if rendering was requested via *trigger_render()*. In case widgets have to request their parents to render, use *arcade.gui.UIWidget.trigger_full_render()*.

The widget has to draw itself and child widgets within *do_render()*. Due to the deferred functionality render does not have to check any dirty variables, as long as state changes use the *trigger_full_render()* method.

For widgets, that might have transparent areas, they have to request a full rendering.

Warning: Enforced rendering of the whole GUI might be very expensive!

23.1.2 UILayout

UILayout are widgets, which reserve the option to move or resize children. They might respect special properties of a widget like `size_hint`, `size_hint_min`, or `size_hint_max`.

The *arcade.gui.UILayout* only resizes a child's dimension (x or y axis) if `size_hint` provides a value for the axis, which is not `None` for the dimension.

Algorithm

arcade.gui.UIManager triggers the layout and render process right before the actual frame draw. This opens the possibility to adjust to multiple changes only once.

Example: Executed steps within *UIBoxLayout*:

1. **do_layout()**
    1. Collect current `size`, `size_hint`, `size_hint_min` of children
    2. Calculate the new position and sizes
    3. Set position and size of children
2. Recursively call `do_layout` on child layouts (last step in *do_layout()*)

```
─────────          ────────                      ────────
│UIManager│          │UILayout│                      │children│
└─────────          └───────                      └───────
     │  do_layout()                                 │
     │─────────────────>│ │                            │
     │                  │ │                            │
     │                  │ │                            │
     │
     │    place children    │                         │
     │    ────────────────use size, size_hint, ...     │
     │                 │ │<──────────────────────────────
     │                 │ │                           │
     │                 │ │      set size and pos      │
     │                 │ │──────────────────────────────>
     │
     │                  │ │                            │
     │                  │ │                            │
     │
     │    LOOP  │  sub layouts                        │
     │    ───────   │ │                              │
     │               │ │         do_layout()          │
     │               │ │──────────────────────────────>
     │
     │                  └                            │
     │                    │                          │
     │<─ ─ ─ ─ ─ ─ ─ ─ ─ ─│                          │
────────         ───────                      ───────
│UIManager│          │UILayout│                      │children│
└─────────          └────────                      └────────
```

Size hint support

	size_hint	size_hint_min	size_hint_max
`UIAnchorLayout`	X	X	X
`UIBoxLayout`	X	X	X
`UIGridLayout`	X	X	X
`UIManager`	X	X	

23.1.3 UIMixin

Mixin classes are a base class which can be used to apply some specific behaviour. Currently the available Mixins are still under heavy development.

23.1.4 Constructs

Constructs are predefined structures of widgets and layouts like a message box or (not yet available) file dialogues.

23.1.5 Available Elements

Buttons

As with most widgets, buttons take `x`, `y`, `width`, and `height` parameters for their sizing. Buttons specifically have two more parameters - `text` and `multiline`.

All button types support styling. And they are text widgets, which means you can use the `_label` attribute to get the label component of the button.

Flat button

Name: `FlatButton`

A flat button for simple interactions (hover, press, release, click). This button is created with a simple rectangle. Flat buttons can quickly create a nice-looking button. However, depending on your use case, you may want to use a texture button to further customize your look and feel.

Styling options are shown in the table below.

Name	Description
`font_s`	Font size for the button text. Defaults to 12.
`font_n`	Font name or family for the button text. If a tuple is supplied then arcade will attempt to load all of the fonts, prioritizing the first one. Defaults to `("calibri", "arial")`.
`font_c`	Font color for the button text (foreground). Defaults to white for normal, hover, and disabled states. Defaults to black for pressed state.
`bg`	Background color of the button. This modifies the color of the rectangle within the button and not the border. Instead of making each of these different colors for each of your buttons, set these towards a common color theme. Defaults to gray for hover and disabled states. Otherwise it is white.
`borde`	Border color. It is common to only modify this in a focus or hover state. Defaults to white or turquoise for hover.
`borde`	Width of the border/outline of the button. It is common to make this thicker on a hover or focus state, however an overly thick border will result in your GUI looking old or low-quality. Defaults to 2.

Image/texture button

Name: *`UITextureButton`*

An image button. Textures are supplied from *`arcade.load_texture()`* for simple interactions (hover, press, release, click). A texture lets you further customize the look of the widget better than styling.

A texture button a few more arguments than a flat button. `texture`, `texture_hovered`, and `texture_pressed` will change the texture displayed on the button respectively. `scale` will change the scaling or size of the button - it's similar to the sprite `scale`.

Hint: This widget *does* have `width` and `height` parameters, but they only stretch the texture instead of resizing it with keeping the borders. This feature is currently in-progress.

Texture buttons have fewer styling options when they have a texture compared to flat buttons.

Name	Description
`font_si`	Font size for the button text. Defaults to 12.
`font_na`	Font name or family for the button text. If a tuple is supplied then arcade will attempt to load all of the fonts, prioritizing the first one. Defaults to `("calibri", "arial")`.
`font_co`	Font color for the button text (foreground). Defaults to white for normal, hover, and disabled states. Defaults to black for pressed state.
`border_`	Width of the border/outline of the button. It is common to make this thicker on a hover or focus state, however an overly thick border will result in your GUI looking old or low-quality. Defaults to 2.

Text widgets

All text widgets take `x` and `y` positioning parameters. They also accept `text` and `multiline` options.

Label

Name: *UILabel*

A label is used to display text as instruction for the user. Multiline text is supported, and what would have been its style options were moved into the parameters.

This widget has no style options whatsoever, and they have been moved into the parameters. bold and italic will set the text to bold or italic. align specifies the justification of the text. Additionally it takes font_name, font_size, and text_color options.

Using the label property accesses the internal *Text* class.

Hint: A *text* attribute can modify the displayed text. Beware-calling this again and again will give a lot of lag. Use begin_update() and py:meth:~*arcade.Text.end_update* to speed things up.

Text input field

Name: *UIInputText*

A text field allows a user to input a basic string. It uses pyglet's IncrementalTextLayout and its Caret. These are stored in layout and caret properties.

This widget takes width and height properties and uses a rectangle to display a background behind the layout.

A text input field allows the user to move a caret around text to modify it, as well as selecting parts of text to replace or delete it. Motion symbols for a text field are listed in pyglet.window.key module.

Text area

Name: *UITextArea*

A text area is a scrollable text widget. A user can scroll the mouse to view a rendered text document. **This does not support editing text**. Think of it as a scrollable label instead of a text field.

width and height allocate a size for the text area. If text does not fit within these dimensions then only part of it will be displayed. Scrolling the mouse will display other sections of the text incrementally. Other parameters include multiline and scroll_speed. See view_y on scroll speed.

Use layout and doc to get the pyglet layout and document for the text area, respectively.

23.1.6 User-interface events

Arcade's GUI events are fully typed dataclasses, which provide information about an event affecting the UI.

All pyglet window events are converted by the *UIManager* into UIEvents and passed via dispatch_event() to the *on_event()* callbacks.

Widget-specific events (such as *UIOnClickEvent* are dispatched via on_event and are then dispatched as specific event types (like on_click).

A full list of event attributes is shown below.

Event	Attributes
UIEvent	None
UIMouseEvent	x, y
UIMouseMovementEvent	dx, dy
UIMousePressEvent	dx, dy, button, modifiers
UIMouseDragEvent	dx, dy
UIMouseScrollEvent	scroll_x, scroll_y
UIKeyEvent	symbol, modifiers
UIKeyReleaseEvent	None
UITextEvent	text
UITextMotionEvent	motion
UITextMotionSelectEvent	selection
UIOnClickEvent	None
UIOnUpdateEvent	dt
UIOnChangeEvent	old_value, new_value
UIOnActionEvent	action

- *arcade.gui.UIEvent*. Base class for all events.
- ***arcade.gui.UIMouseEvent*. Base class for mouse-related events.**
  - *arcade.gui.UIMouseMovementEvent*. Mouse motion. This event has an additional `pos` property that returns a tuple of the x and y coordinates.
  - *UIMousePressEvent*. Mouse button pressed.
  - *UIMouseDragEvent*. Mouse pressed and moved (drag).
  - *UIMouseReleaseEvent*. Mouse button release.
  - *UIMouseScrollEvent*. Mouse scroll.
- *UITextEvent*. Text input from user. This is only used for text fields and is the text as a string that was inputed.
- *UITextMotionEvent*. Text motion events. This includes moving the text around with the caret. Examples include using the arrow keys, backspace, delete, or any of the home/end and PgUp/PgDn keys. Holding `Control` with an arrow key shifts the caret by a entire word or paragraph. Moving the caret via the mouse does not trigger this event.
- *UITextMotionSelectEvent*. Text motion events for selection. Holding down the `Shift` key and pressing arrow keys (`Control` optional) will select character(s). Additionally, using a `Control-A` keyboard combination will select all text. Selecting text via the mouse does not trigger this event.
- *UIOnUpdateEvent*. This is a callback to the arcade *on_update* method.

Widget-specific events

Widget events are only dispatched as a pyglet event on a widget itself and are not passed through the widget tree.

- *UIOnClickEvent*. Click event of *UIInteractiveWidget* class. This is triggered on widget press.
- *UIOnChangeEvent*. A value of a *UIWidget* has changed.
- *UIOnActionEvent*. An action results from interaction with the *UIWidget* (mostly used in constructs)

23.1.7 Different event systems

Arcade's GUI uses different event systems, dependent on the required flow. A game developer should mostly interact with user-interface events, which are dispatched from specific `UIWidget`s like an ``on_click`` of a button.

In rare cases a developer might implement some widgets themselves or want to modify the existing GUI behavior. In those cases a developer might register own pyglet event types on widgets or overwrite the *on_event* method. In that case, refer to existing widgets as an example.

Pyglet window events

Pyglet window events are received by *UIManager*.

You can dispatch them via:

```
UIWidget.dispatch_event("on_event", UIEvent(...))
```

Window events are wrapped into subclasses of *UIEvent*.

Pyglet event dispatcher - UIWidget

Widgets implement pyglet's `EventDispatcher` and register an `on_event` event type.

on_event() contains specific event handling and should not be overwritten without deeper understanding of the consequences.

To add custom event handling, use the decorator syntax to add another listener:

```
@UIWidget.event("on_event")
```

User-interface events

User-interface events are typed representations of events that are passed within the GUI. Widgets might define and dispatch their own subclasses of these events.

Property

Property is an pure-Python implementation of Kivy Properties. They are used to detect attribute changes of widgets and trigger rendering. They should only be used in arcade internal code.

23.2 GUI Style

With arcade 3.0 a whole new styling mechanism for GUI widgets was introduced. The new styling allows more type safe and clear styling while staying flexible.

Following widgets support styling:

- *UITextureButton*
- *UIFlatButton*
- *UISlider*

For an advanced description about the style system read the 'Advanced' section.

23.2.1 Basic Usage

This section covers how to use the existing stylable widgets.

In the following examples we will use the *UIFlatButton* as the stylable widget, you can do the same with any stylable widget listed above.

Quickstart

The following example shows how to adjust the style.

```
# create an own style
new_style = {
    # provide a style for each widget state
    "normal": UIFlatButton.UIStyle(), # use default values for `normal` state
    "hover": UIFlatButton.UIStyle(
        font_color=arcade.color.BLACK,
        bg=arcade.color.WHITE,
    ),
    "press": UIFlatButton.UIStyle(
        font_color=arcade.color.BLACK,
        bg=arcade.color.WHITE,
        border=arcade.color.WHITE,
    ),
    "disabled": UIFlatButton.UIStyle(
        bg=arcade.color.GRAY,
    )
}

UIFlatButton(style=new_style)
```

Default style

Stylable widgets have a property which holds the default style for the type of widget. For the *UIFlatButton* this is *UIFlatButton.DEFAULT_STYLE*.

This default style will be used if no other style is provided within the constructor. The default style looks like this:

```
class UIFlatButton(UIInteractiveWidget, UIStyledWidget, UITextWidget):

    DEFAULT_STYLE = {
        "normal": UIStyle(),
        "hover": UIStyle(
            font_size=12,
            font_name=("calibri", "arial"),
            font_color=arcade.color.WHITE,
            bg=(21, 19, 21, 255),
            border=(77, 81, 87, 255),
            border_width=2,
        ),
        "press": UIStyle(
            font_size=12,
            font_name=("calibri", "arial"),
```

(continues on next page)

(continued from previous page)

```
            font_color=arcade.color.BLACK,
            bg=arcade.color.WHITE,
            border=arcade.color.WHITE,
            border_width=2,
        ),
        "disabled": UIStyle(
            font_size=12,
            font_name=("calibri", "arial"),
            font_color=arcade.color.WHITE,
            bg=arcade.color.GRAY,
            border=None,
            border_width=2,
        )
    }
```

Style attributes

A UIStyle is a typed description of available style options. For the UIFlatButton the supported attributes are:

Name	Type	Default value	Description
font_size	int	12	Size of the text on the button
font_name	FontNameOrNames	("calibri", "arial")	Font of the text
font_color	RGBA255	arcade.color.WHITE	Color of text
bg	RGBA255	(21, 19, 21, 255)	Background color
border	Optional	None	Border color
border_width	int	0	Border width

The style attribute is a dictionary, which maps a state like 'normal, 'hover' etc. to an instance of UIFlatButton.UIStyle.

Wellknown states

Name	Description
normal	The default state of a widget.
hover	Mouse hovered over an interactive widget.
press	Mouse is pressed while hovering over the widget.
disabled	The widget is disabled.

23.2.2 Advanced

This section describes the styling system itself, and how it can be used to create own stylable widgets or extend existing ones.

Stylable widgets inherit from *UIStyledWidget*, which provides two basic features:

1. owns a style property, which provides a mapping between a widgets state and style to be applied
2. provides an abstractmethod which have to provide a state (which is a simple string)

Tha basic idea:

- a stylable widget has a state (e.g. ‘normal’, ‘hover’, ‘press’, or ‘disabled’)
- the state is used to define, which style will be applied

Your own stylable widget

```
class MyColorBox(UIStyledWidget, UIInteractiveWidget, UIWidget):
    """
    A colored box, which changes on mouse interaction
    """

    # create the style class, which will be used to define style for any widget state
    @dataclass
    class UIStyle(UIStyleBase):
        color: RGBA255 = arcade.color.GREEN


    DEFAULT_STYLE = {
        "normal": UIStyle(),
        "hover": UIStyle(color=arcade.color.YELLOW),
        "press": UIStyle(color=arcade.color.RED),
        "disabled": UIStyle(color=arcade.color.GRAY)
    }

    def get_current_state(self) -> str:
        """Returns the current state of the widget i.e disabled, press, hover or normal."
↪""
        if self.disabled:
            return "disabled"
        elif self.pressed:
            return "press"
        elif self.hovered:
            return "hover"
        else:
            return "normal"

    def do_render(self, surface: Surface):
        self.prepare_render(surface)

        # get current style
        style: MyColorBox.UIStyle = self.get_current_style()

        # Get color from current style, it is a good habit to be
        # bullet proven for missing values in case a dict is provided instead of a␣
↪UIStyle
        color = style.get("color", MyColorBox.UIStyle.bg)

        # render
        if color: # support for not setting a color at all
            surface.clear(bg_color)
```

23.3 Troubleshooting & Hints

23.3.1 `UILabel` does not show the text after it was updated

Currently the size of `UILabel` is not updated after modifying the text. Due to the missing information, if the size was set by the user before, this behaviour is intended for now. To adjust the size to fit the text you can use `UILabel.fit_content()`.

In the future this might be fixed.

CHAPTER

TWENTYFOUR

TEXTURE ATLAS

24.1 Introduction

`arcade.TextureAtlas` is where your textures eventually end up when they are used in a sprite. This is where the image data is moved to graphics memory (OpenGL) and is one of the reasons we can batch draw hundreds of thousands of sprites extremely fast.

A texture atlas is basically a large texture containing multiple textures and we keep track of where these textures are located. Arcade's texture atlas reside in graphics memory and is dynamic meaning textures can be added and removed on the fly.

Arcade's texture atlas also automatically resizes when needed all the way up to the maximum texture size your hardware supports. This requires a complete rebuild of the atlas, something we do on the gpu itself to minimize the impact of this operations. For average hardware it's something you won't notice runtime.

It's also important to note that texture atlases can only be created after the window has been created. Textures and sprites can be created before the window because they don't interact with OpenGL directly. This part is usually the most time consuming while atlases are very fast to create and build.

24.2 Size Restriction

Currently we use a very simple row based allocation algorithm to make room for new textures over time. This means that very tall textures can end up taking a lot of vertical space.

The maximum size of the atlas is usually 16384 x 16384 if we are targeting average hardware.

24.3 Resize

Atlases will resize automatically when full. It will also try to pack the textures better by sorting them by their height.

24.4 Default Texture Atlas

Most users will not be aware that arcade is using a texture atlas under the hood. More advanced users can take advantage of these if they run into limitations.

Arcade has a global default texture atlas stored in `window.ctx.default_atlas`. This is an instance of *arcade.ArcadeContext* where the low level rendering API is accessed (OpenGL).

24.5 Custom Atlas

Instead of relying on the global texture atlas we can also create our own. Sprite lists take an `atlas` argument for supplying your own texture atlas instance. This atlas can also be shared between several sprite lists if needed.

```
# Create an empty 256 x 256 texture atlas
my_atlas = TextureAtlas((256, 256))
spritelist = SpriteList(atlas=my_atlas)
```

When new textures are detected (sprite is added to list) the texture is added to the atlas.

We can also pre-add textures into an atlas before the game starts to avoid potential minor stalls. This is usually not a problem, but when adding a large amount of them it can be noticeable.

```
# List of arcade.Texture instances
list_of_textures = ...

# Create an atlas with a reasonable size for a list of textures
atlas = TextureAtlas.create_from_texture_sequence(list_of_textures)

# Create an atlas with a specific size and initial textures
atlas = TextureAtlas((256, 256), textures=list_of_textures)

# We can also pre-add textures at any time using:
# (can also be done with the default texture atlas)
atlas.add(texture)
```

24.6 Border

Atlases has a `border` property that is 1 by default. This is important to avoid "texture bleeding" between borders of the textures in the atlas. This is a very common issues in games using the gpu based graphics and is even a problem with using `NEAREST` interpolation when sprites are rotating.

Keep the default value of this property unless you know exactly what you are doing.

24.7 Updating Texture

In some instances it can be useful to update a texture. We would normally do this by modifying the Pillow texture in the *arcade.Texture* instance. However, this doesn't update the texture in the atlas itself. We can manually update it:

```
# Change the internal image in a texture
texture.image  # <- Modify or crate a new image with the same size

# Write the new image data to the atlas
atlas.update_texture_image(texture)
```

This updates the already allocated region and the image needs to be exactly the same size. This should be used sparingly or at least not a per frame operation. If can be fast as a per-frame operation, but you'll need to profile that. Animated sprites are much better option, but of course requires pre-determined texture frames.

24.8 Removing Texture

If you have stale textures they can be removed from the atlas using:

```
atlas.remove(texture)
```

This will make the region free for new textures the next time the atlas rebuilds. You can also call `arcade.TextureAtlas.rebuild()` directly if you are removing a large quantity of textures, but generally it's enough to let this happen automatically when needed.

24.9 Rendering Into Atlas

A much faster way to update a texture in the atlas is rendering directly into it. This can for example be used to make a minimap for your game or in any case you need the sprite texture to be really dynamic (not decided by pre-made texture frames). It can be used in many creative ways.

```
# --- Initialization ---
# Create an empty texture so we can allocate some space in the atlas
texture = arcade.Texture.create_empty("render_area_1", size=(256, 256))

# Assign the texture to a sprite
sprite = arcade.Sprite(center_x=200, center_y=300, texture=texture)

# Create the spritelist and add the sprite
spritelist = arcade.SpriteList()
# Adding the sprite will also add the texture to the atlas
spritelist.append(sprite)

# -- Rendering ---
# Let's render something into our texture directly.
# All operations will only affect the allocated portion of the atlas for texture.
# We are given a framebuffer instance representing this area
with spritelist.atlas.render_into(texture) as framebuffer:
    # Clear the allocated region in the atlas (if you need it)
    framebuffer.clear()
```

(continues on next page)

(continued from previous page)

```
    # From here on we can draw using any arcade draw functionality
    arcade.draw_rectangle_filled(128, 128, 160, 160, arcade.color.WHITE, rotation)

# Draw the spritelist and see your animating sprite texture
spritelist.draw()
```

Doing the rendering part above every frame (and incrementing `rotation` by delta time) will give you a sprite with a rotating rectangle a a texture. Again, you can draw anything into this texture area. Spritelists, shapes and whatnot.

We can also specify what should be projected into this texture area in the atlas. By default the projection will be `(0, width, 0, height)`, but this is not always what you want (were `width` and `height` are the region/texture size)

```
# Assuming your window is 800 x 600 we could draw the entire game into this atlas region
projection = 0, 800, 0, 600
with spritelist.atlas.render_into(texture, projection=projection) as framebuffer:
    framebuffer.clear()
    # Draw your game here

# Draw sprite with a texture containing your entire game here
```

Scrolling can also be applied to projection just like cameras.

```
# Scroll projection (or even zoom)
projection = 0 + scroll_x, 800 + scroll_x, 0 + scroll_y, 600 + scroll_y
```

Rendering into an atlas is superior (at least 100 times faster) to updating texture data using Pillow, but that doesn't mean it's free. We can possibly get away with 50-100 of these per frame, but this is something you will have to profile.

24.10 Debugging

When working with atlases it can be useful to see the contents. We provide two methods for this.

`arcade.TextureAtlas.show()` will display the atlas using Pillow:

```
atlas.show()
```

`arcade.TextureAtlas.save()` will save the atlas contents to a png file:

```
atlas.write("path/to/atlas.png")
```

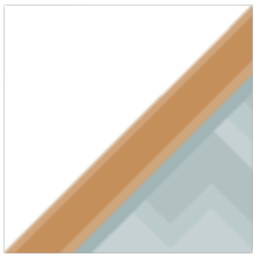
Both of these methods will “download” the atlas texture from graphics memory for you to inspect the raw data.

CHAPTER

TWENTYFIVE

EDGE ARTIFACTS

When working with images, particularly ones with transparency, graphics cards can create graphic artifacts on their edges. Images can have 'borders' where they aren't wanted. For example, here there's a line on the top and left:



Why does this happen? How do we fix it?

25.1 Why Edge Artifacts Appear

This happens when the edge of an image does not fall cleanly onto an image.

25.1.1 Edge Mis-Alignment

Typically edge artifacts happen when the edge of an image doesn't land on an exact pixel boundary. Below in Figure 1, the left image is 128 pixels square and drawn at (100, 100), and looks fine. The image on the right is drawn with a center of (100, 300.5) and has an artifact that shows up as a line on the left edge. That artifact will not appear if the sprite is drawn at (100, 300) instead of (100, 300.5)

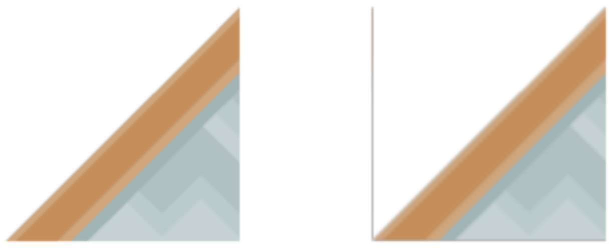


Fig. 1: Figure 1: Edge artifacts caused by images that aren't on integer pixel boundaries.

The left edge falls on a coordinate of 300.5 - (128/2) = 236.5. The computer tries to select a color that's an average between 236 and 237, but since there is no 237 we get a dark color. Typically this only happens if the edge is transparent.

A shape that has a height or width that is not evenly divisible by two can also cause artifacts. If the shape is 15 pixels wide, then the center will fall between the 7th and 8th pixel making it harder to line up the pixels to the screen.

25.1.2 Scaling

Scaling an image can also cause artifacts. In Figure 2, the second sprite is scaled down by two-thirds. Since 128 pixels doesn't evenly scale down by two-thirds, we end up with edge artifacts. If we had scaled down by one-half, that is possible to do with 128 pixels (to 64), so there would be no artifacts.

The third image in Figure 2 is scaled up by a factor of two. The edge spans two pixels and we end up with a line artifact as well. (Scaling down by two usually works if the image is divisible by four. Scaling up typically doesn't.)

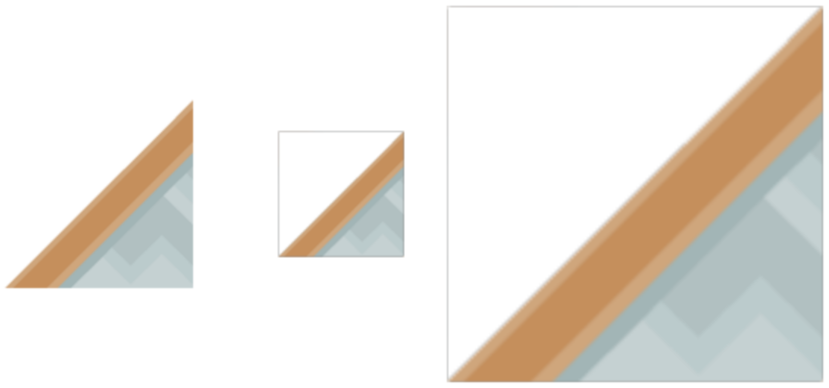


Fig. 2: Figure 2: Edge artifacts caused by scaling.

25.1.3 Rotating

With rotation, it can be very difficult to get pixels lined up, and edge artifacts are common.

25.1.4 Improper Viewport

If a window is 800 wide, and the viewport is set to 799 or 801, then lines can also appear. Alternatively, if a viewport left or right edge is set to a non-integer number such as 23.5, this can cause the artifacts to appear.

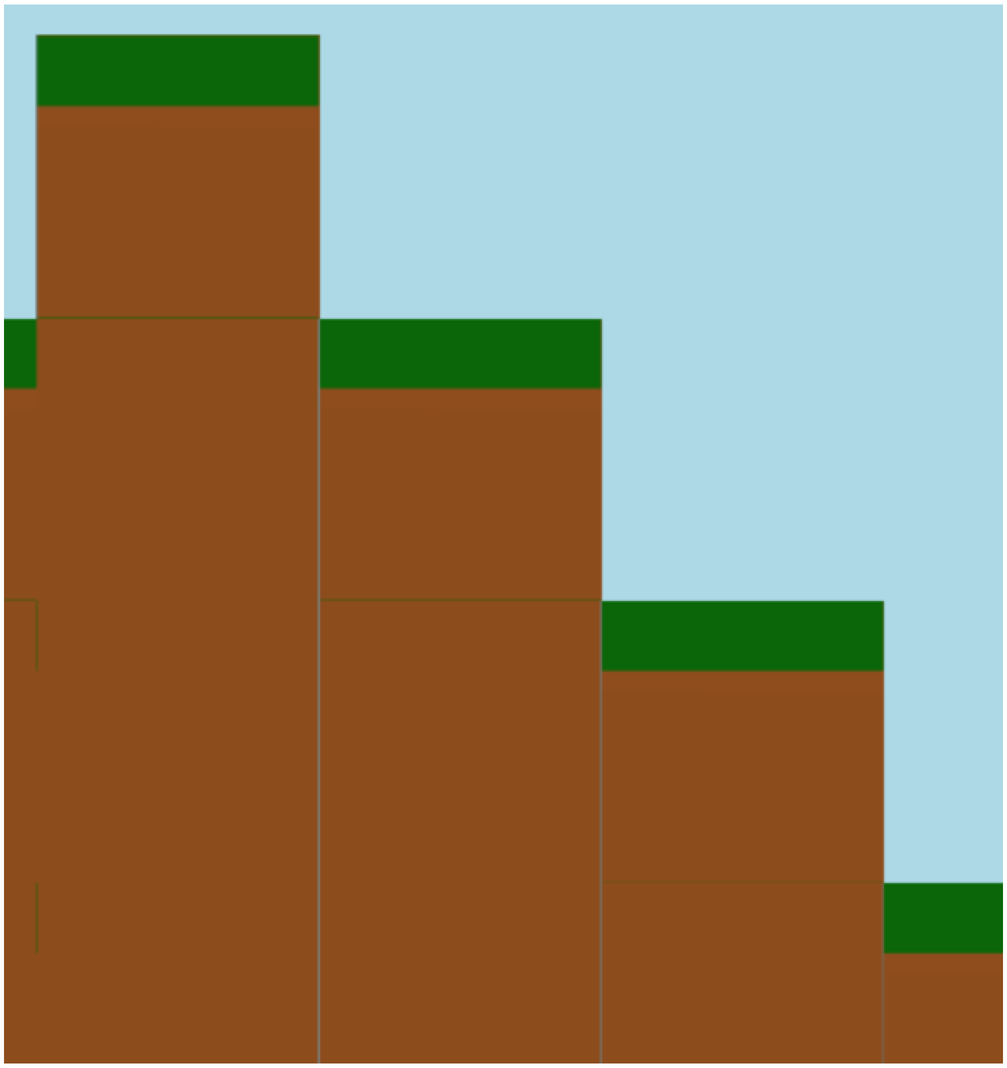


Fig. 3: Figure 3: Incorrect viewport

25.2 Solutions

Keeping sprite sizes to a power of two or at least have a width and heights divisible by 2. For pixel-art types of games, using the `pixelated` drawing mode will greatly reduce the problem.

25.2.1 Aligning to the Nearest Pixel

By default, Arcade draws sprites with a filter called "linear" which makes for smoother scaling and lines. If instead you want a pixel-look, you can use a different filter called "nearest." This filter also reduces issues with edge artifacts.

You enable the nearest filter using the `pixelated` argument when drawing

```python
def on_draw(self):
    self.my_sprite_list.draw(pixelated=True)
```

25.2.2 Double-Check Viewport Code

Double-check your viewport code to make sure the edges are only set to integers and the size of the window matches up exactly, without any off-by-one errors.

CHAPTER

TWENTYSIX

LOGGING

Arcade has a few options to log additional information around timings and how things are working internally. The two major ways to do this by turning on logging, and by querying the OpenGL context.

26.1 Turn on logging

The quickest way to turn on logging is to add this to the start of your main program file:

```
arcade.configure_logging()
```

This will cause the Arcade library to output some basic debugging information:

```
2409.0003967285156 arcade.sprite_list DEBUG - [386411600] Creating SpriteList use_
↪spatial_hash=True capacity=100
2413.9978885650635 arcade.gl.context INFO - Arcade version : 2.4a5
2413.9978885650635 arcade.gl.context INFO - OpenGL version : 3.3
2413.9978885650635 arcade.gl.context INFO - Vendor         : NVIDIA Corporation
2413.9978885650635 arcade.gl.context INFO - Renderer       : GeForce GTX 980 Ti/PCIe/SSE2
2413.9978885650635 arcade.gl.context INFO - Python         : 3.7.4 (tags/v3.7.
↪4:e09359112e, Jul  8 2019, 19:29:22) [MSC v.1916 32 bit (Intel)]
2413.9978885650635 arcade.gl.context INFO - Platform       : win32
3193.9964294433594 arcade.sprite_list DEBUG - [386411600] _calculate_sprite_buffer: 0.
↪013532099999999936 sec
```

26.1.1 Custom Log Configurations

If you want to add your own logging, or change the information printed in the log, you can do it with just a bit more code.

First, in your program import the logging library:

```
import logging
```

The code to turn on logging looks like this:

```
logging.basicConfig(level=logging.DEBUG)
```

You can get even more information by using a formatter to add time, file name, and even line number information to your output:

```
format = '%(asctime)s,%(msecs)03d %(levelname)-8s [%(filename)s:%(lineno)d
↪%(funcName)s()] %(message)s'
logging.basicConfig(format=format,
                    datefmt='%H:%M:%S',
                    level=logging.DEBUG)
```

…which changes the output to look like:

```
13:40:50,226 DEBUG    [sprite_list.py:720 _calculate_sprite_buffer()] [365177904] _
↪calculate_sprite_buffer: 0.008496600000000041 sec
13:40:50,398 DEBUG    [ui_element.py:58 on_mouse_over()] UIElement mouse over
```

You can add logging to your own programs by putting one of these lines at the top of your program:

```
# Get your own logger
LOG = logging.getLogger(__name__)
# or get Arcade's logger
LOG = logging.getLogger('arcade')
```

Then, any time you want to print, just use:

```
LOG.debug("This is my debug statement.")
```

26.2 Getting OpenGL Stats Using Query Objects

If you'd like more information on the time it takes to draw, you can query the OpenGL context `arcade.Window.ctx` as this example shows:

```
def on_draw(self):
    """ Render the screen. """
    self.clear()

    query = self.ctx.query()
    with query:
        # Put the drawing commands you want to get info on here:
        self.my_sprite_list.draw()

    print()
    print(f"Time elapsed       : {query.time_elapsed:,} ns")
    print(f"Samples passed     : {query.samples_passed:,}")
    print(f"Primitives created : {query.primitives_generated:,}")
```

The output from this looks like the following:

```
Time elapsed       : 7,136 ns
Samples passed     : 390,142
Primitives created : 232
```

CHAPTER

TWENTYSEVEN

OPENGL

Arcade is using OpenGL for the underlying rendering. OpenGL functionality is given to use through pyglet when a window is crated. The underlying representation of this is an OpenGL context. Arcade's representation of this context is the *arcade.Window.ctx*. This is an *ArcadeContext*.

Working with OpenGL adds some challenges we need to be aware of.

27.1 Initialization

Certain operations can't be done before a window is created. In Arcade we do deferred initialization in many of our types to make this as painless as possible for the user. *SpriteList* can for example be built before window creation and will be initialized internally in the first draw call.

`TextureAtlas` on the other hand cannot be crated before the window is created, but *Texture* can freely be loaded at any time since these only manage pixel data with Pillow and calculate hit box data on the cpu.

27.2 Garbage Collection & Threads

OpenGL is not thread safe meaning doing actions from anything but the main thread is not possible. You can still use threads with arcade, but they cannot interact with anything that affects OpenGL objects. This will throw an error immediately.

When threads are used in a project or underlying libraries there is always the risk that Python's garbage collector will run outside the main thread. This is just how Python's garbage collector works.

For this reason, Arcade's default garbage collection mode requires actively releasing OpenGL objects. We are doing this for you in the *arcade.Window.flip()* method that is automatically called every frame.

This garbage collection mode is called `context_gc` since dead OpenGL objects are collected in the context and only released when `ctx.gc()` is called.

Garbage collection modes can be configured during window creation or changed runtime in the context.

```
# auto mode works like python's garbage collection (but more risky)
window = Window(gc_mode="auto")

# This context mode is implied by default
window = Window(gc_mode="context_gc")
# From now on you need to manually call window.ctx.gc()
# for OpenGL resources to be deleted. This can be
# done very frame if needed or in shorter intervals
```

(continues on next page)

(continued from previous page)

```
num_released = window.ctx.gc()
print("Resources released:", num_released)

# Change gc mode runtime
window.gc_mode = "auto"
window.gc_mode = "context_gc"
```

If you for some reason need garbage collection to run more often than once per frame it can safely be called as many times as you want from the main thread.

In the vast majority of cases this is nothing you need to be worried about. The current default exists to make your life as easy as possible.

27.3 Threads & vsync

Note that if vsync is enabled all threads will stall when all rendering is done and OpenGL is waiting for the next vertical blank. The only way to combat this is to disable vsync or use sub-processes.

27.4 SpriteList & Threads

SpriteLists can be created in threads if they are created with the `lazy=True` parameters. This ensures OpenGL resources are not created until the first `draw()` call or `initialize()` is called.

27.5 Writing Raw Bytes to GL Buffers & Textures

Many of arcade's OpenGL classes support creation from or writing to any object that supports the buffer protocol. The classes most useful to end users are:

- *`arcade.gl.Buffer`*
- `arcade.gl.Texture`

This functionality can be used for displaying the results of calculations such as:

- Scientific visualizations displaying data from numpy arrays
- Simple console emulators drawing their internal screen buffer

There should be no typing issues when using Python's built-in buffer protocol objects as arguments to the `write` method of arcade's GL objects. We list these built-in types in the `arcade.arcade_types.BufferProtocol` Union type.

For objects from third-party libraries, your type checker may warn you about type mismatches. This is because Python will not support general annotations for buffer protocol objects until version 3.12 at the earliest.

In the meantime, there are workarounds for users who want to write to arcade's GL objects from third-party buffer protocol objects:

- use the typing.cast method to convert the object's type for the linter
- use `# type: ignore` to silence the warnings

CHAPTER

TWENTYEIGHT

PERFORMANCE

The three areas where a game might experience the greatest slowdowns are collision detection, drawing primitive performance, and sprite drawing performance.

28.1 Collision detection performance

Detecting collisions between sprites can take a while. If you have a map with 50,000 sprites making up walls, then every frame you have to make 50,000 checks. (An O(N) operation, if you are familiar with **Big O** notation.) If your game includes multiple things that need to check for collisions (enemies, bullets, etc.) then each of those need to do checks. That can take long enough a game can start slowing below 60 FPS.

How can we speed things up? Arcade can use a technique called **spatial hashing**.

28.1.1 Spatial Hashing

Arcade divides the screen up into a grid. We track which grid location(s) each sprite overlaps, and put them in a **hash map**. For each grid location, we can quickly pull the sprites in that grid in a fast O(1) operation. When looking for sprites that collide with our target sprite, we only look at sprites in sharing its grid location. This can reduce checks from 50,000 to just 3 or 4.

There is a drawback. If the sprite moves, we have to recalculate and re-hash its location. This takes time. This doesn't mean we can't *ever* move the sprite! But it does mean we have to make a choice around using spatial hashing or not:

- Only have a few sprites? Less than 100? Then it is too small to matter what you pick.
- Do we not need to check for collisions with a sprite list? Spatial hashing off.
- Do all the sprites in our sprite list move every frame? Spatial hashing off.
- Are the sprites platforms? Most of them not moving? Spatial hashing on.

Arcade defaults to no spatial hashing. Spatial hashing can be turned on by:

```python
self.my_sprite_list = arcade.SpriteList(use_spatial_hashing=True)
```

28.1.2 Compute Shader

Currently on the drawing board, is the use of a **compute shader** on your graphics card to detect collisions. This has the speed advantages of spatial hashing, without the speed penalty.

28.2 Drawing primitive performance

Drawing lines, rectangles, and circles can be slow. Every drawing command is sent individually to the graphics card 60 times per second. If you are drawing hundreds or thousands of lines/boxes then performance will be terrible.

If you are encountering this, you can speed things up by using `arcade.ShapeElement` lists where you batch together the drawing commands. If you can group items together, than drawing a complex tree can be done with just one command.

For more information see: shape_list_demo.

28.3 Sprite drawing performance

Sprite drawing is done in batches via the `arcade.SpriteList` class. Sprites are loaded to the graphics card and drawn in a batch. Sprites that don't move can be re-drawn incredibly fast. Sprites that do move only need their position updated. Sprite drawing with Arcade is incredibly fast, and requires rarely needs any extra effort from the programmer.

28.4 Text drawing performance

Arcade's `arcade.draw_text()` can be quite slow. To speed things up, use text objects. See drawing_text_objects.

CHAPTER

TWENTYNINE

HEADLESS ARCADE

For some applications, it may be that we want to run Arcade, but not open up a window. We might want to draw to a buffer and save an image to be used in a server or data science visualization. In remote cloud operations, we might not even have a monitor for the computer. Running Arcade this way is called headless mode.

Arcade can render in headless mode on Linux servers with EGL installed. This should work both in a desktop environment and on servers and even in virtual machines. Both software and hardware rendering should be acceptable depending on your use case.

We are leveraging the headless mode in pyglet. If you are seeking knowledege about the inner workings of headless, that's the right place to look.

29.1 Enabling headless mode

Headless mode needs to be configured **before** arcade is imported. This can be done in the following ways:

```
# Before arcade is imported
import os
os.environ["ARCADE_HEADLESS"] = "True"

# The above is a shortcut for
import pyglet
pyglet.options["headless"] = True
```

This of course also means you can configure headless externally.

```
$ export ARCADE_HEADLESS=True
```

To quickly check the enviroment such as renderer and versions:

```
$ python -m arcade

Arcade 2.6.12
-------------
vendor: AMD
renderer: AMD Radeon(TM) Vega 11 Graphics (RAVEN, DRM 3.41.0, 5.13.0-37-generic, LLVM 12.
↪0.0)
version: (4, 6)
python: 3.9.9 (main, Dec 20 2021, 08:19:16)
[GCC 9.3.0]
platform: linux
```

29.2 How is this affecting my code?

In headless mode we don't have any window events or inputs events. This means events like `on_key_press` and `on_mouse_motion` will never be called. A project not created for a headless setting will need some tweaking.

In headless mode the arcade `Window` will extend pyglet's headless window instead. We've added a property `arcade.Window.headless` (bool) that can be used to separate headless logic.

Note that the window itself still has a framebuffer you can render to and read pixels from. The size of this framebuffer is the size you specify when creating the window. More framebuffers can be created through the `ArcadeContext` if needed.

Warning: If you are creating and destroying a lot of arcade objects you might want to look into `arcade.ArcadeContext.gc_mode`. In Arcade we normally do garbage collection of OpenGL objects once per frame by calling `gc()`.

Warning: If you are loading an increasing amount of textures you might need to clean up the texture cache. This only caches `arcade.Texture` objects. See `cleanup_texture_cache()`. This might also involve removing them from the global texture atlas if you are using these textures on sprites.

29.3 Examples

There are two recommended approaches: *Simple headless mode* and *Headless mode while extending the Arcade Window*.

29.3.1 Simple headless mode

For simpler applications we don't need to subclass the window.

```python
# Configure headless before importing arcade
import os
os.environ["ARCADE_HEADLESS"] = "true"
import arcade

# Create a 100 x 100 headless window
window = arcade.open_window(100, 100)

# Draw a quick rectangle
arcade.draw_rectangle_filled(50, 50, 50, 50, color=arcade.color.AMAZON)

# Dump the framebuffer to a png
image = arcade.get_image(0, 0, *window.get_size())
image.save(f"framebuffer.png")
```

You are free to `clear()` the window and render new contents at any time.

29.3.2 Headless mode while extending the Arcade Window

For Arcade users extending the window, this method makes more sense. The *run()* method supports headless mode and will emulate Pyglet's event loop by calling `on_update`, `on_draw` and `flip()` (swap buffers) in a loop until you close the window.

```python
import os
os.environ["ARCADE_HEADLESS"] = "true"
import arcade

class App(arcade.Window):

    def __init__(self):
        super().__init__(200, 200)
        self.frame = 0
        self.sprite = arcade.Sprite(
            ":resources:images/animated_characters/female_adventurer/femaleAdventurer_
↪idle.png",
            center_x=self.width / 2,
            center_y=self.height / 2,
        )

    def on_draw(self):
        self.clear()
        self.sprite.draw()

        # Dump the window framebuffer to disk
        image = arcade.get_image(0, 0, *self.get_size())
        image.save("framebuffer.png")

    def on_update(self, delta_time: float):
        # Close the window on the second frame
        if self.frame == 2:
            self.close()

        self.frame += 1

App().run()
```

You can also split your code into *arcade.View* classes if needed. Doing it this way might make it simpler to work with headless and non-headless mode during development. You just need to programmatically close the window and switch views. We can easily separate logic with the *arcade.Window.headless* flag. When calling `run()` we also garbage collect OpenGL resources every frame.

29.4 Advanced

The lower level rendering API is of course still available through `arcade.Window.ctx`. It exposes methods to create framebuffers, textures, shaders (including compute shaders) and other higher level wrappers over OpenGL types.

When working in a multi-gpu environment you can also select a specific device id. This is 0 by default and must be set before the window is created. These device ids usually refers to a physical device (graphics card) or a virtual card/device.

```
# Default setting
pyglet.options['headless_device'] = 0

# Use the second gpu/device
pyglet.options['headless_device'] = 1
```

29.5 Issues?

If you run into issues or have questions please create an issue on github or join our discord server.

CHAPTER

THIRTY

VERTICAL SYNCHRONIZATION

30.1 What is vertical sync?

Vertical synchronization (vsync) is a window option in which the video card is prevented from doing anything visible to the display memory until after the monitor finishes its current refresh cycle.

To enable vsync in arcade:

```python
# On window creation
arcade.Window(800, 600, "Window Title", vsync=True)

# While the application is running
window.set_vsync(True)
```

This have advantages and disadvantages depending on the situation.

Most windows are what we call “double buffered”. This means the window actually has two surfaces. A visible surface and a hidden surface. All drawing commands will end up in the hidden surface. When we’re done drawing our frame the hidden and visible surfaces swap places and the new frame is revealed to the user.

If this “dance” of swapping surfaces is not timed correctly with your monitor you might experience small hiccups in movement.

30.2 Vertical sync disabled as a default

The arcade window is by default created with vertical sync disabled. This is a much safer default for a number of reasons.

- In some environments vertical sync is capped to 30 fps. This can make the game run at half the speed if `delta_time` is not accounted for. We don’t expect beginners take `delta_time` into consideration in their projects.
- If threads are used all threads will stall while the application is waiting for vertical sync

We cannot guarantee that vertical sync is disabled if this is enforced on driver level. The vast amount of driver defaults lets the application control this.

30.3 Advantages of vertical sync

If you have any kind of movement, scrolling or animation in your application you might have noticed a very subtle hiccup periodically or randomly. This can be reduced or entirely removed by enabling vertical sync. In some environments/platforms you can even experience screen tearing.

When vsync is enabled we have to make sure all movement is takes `delta_time` into consideration. **This can also improve smoothness when vsync is not enabled**:

```
# Move 100 units in one second
MOVEMENT_SPEED = 100

def on_update(self, delta_time):
    # Move your sprite based on the time since the last frame.
    # This will make the sprite move along the x axis by
    # 100 units in one second
    self.sprite.center_x += MOVEMENT_SPEED * delta_time
```

CHAPTER

THIRTYONE

PYGAME COMPARISON

Both Pygame and Arcade are Python libraries for making it easy to create 2D games. Pygame is raster-graphics based. It is very fast at manipulating individual pixels and can run on almost anything. Arcade uses OpenGL. It is very fast at drawing sprites and off-loads functions such as rotation and transparency to the graphics card.

In 2023 Pygame split between the Original Pygame and the Pygame Community Edition (Pygame-ce). At this point, the code bases are still pretty similar.

31.1 Library Information

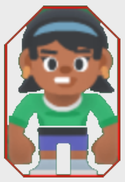
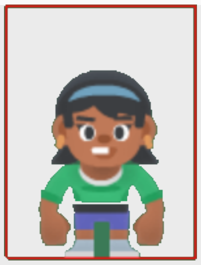
Table 1: Library Information

Feature	Arcade	Pygame Original	Pygame CE
Website	https://arcade.academy	https://www.pygame.org	https://pyga.me/
API Docs	*API Docs*	API Docs	API Docs
Example code	*Arcade Examples*	Pygame Examples	
License	MIT License	LGPL	LGPL
Back-end graphics engine	OpenGL 3.3+ and Pyglet	SDL 2	SDL 2
Back-end audio engine	ffmpeg via Pyglet	SDL 2	SDL 2
Example Projects	*Games Made With Arcade*	Games Made With Pygame	
First Started	2016	Before 2000	Branched 2023

31.2 Feature Comparison

Here are some comparisons between Arcade 3.0 and Pygame 2.2.0 ce:

Table 2: Feature Comparison

Feature	Arcade	Pygame
Drawing primitives support rotation	Yes	No[1]
Sprites support rotation	Yes	No[Page 384, 1]
Sprites support scaling	Yes	No[1]
Texture atlas[2]	Yes	No
Type Hints	Yes	No
Transparency support	Yes	Must specify transparent colorkey
Camera support	Yes	No
Android support	No	Yes
Raspberry Pi support	No	Yes
Batch drawing	Via GPU	Via Surface[3]
Default Hitbox		
Tiled Map Support	Yes	No
Physics engines	Simple, platformer, and PyMunk	None
Event Management	Pyglet-based, write functions to handle events	Write your own event loop. Can get around this by add-ons like Pygame Zero)
View Support	Yes	No
Light Support	Yes	No
GUI Support	Yes	No (or add pygame-gui)
GPU Shader Support	Yes	No
Built-in Resources	Yes	No

31.3 Performance Comparison

These performance tests were done on an Intel Core i7-9700F with GeForce GTX 980 Ti. Source code for tests available at:

- https://craven-performance-testing.s3-us-west-2.amazonaws.com/index.html
- https://github.com/pythonarcade/performance_tests

[1] To support rotation and/or scaling, PyGame programs must write the image to a surface, transform the surface, then create a sprite out of the surface. This takes a lot of CPU. Arcade off-loads all these operations to the graphics card. See for more information.

[2] When creating a sprite from an image, Pygame will load the image from the disk every time unless the user caches the image with their own code for better performance. Arcade will create an atlas of textures, so that multiple sprites with the same image will just reference the same atlas location.

[3] A programmer can achieve a similar result by drawing to a surface, then draw the surface to the screen.

31.3.1 Sprite Drawing

How fast can the graphics libraries draw sprites that don't move? This graph shows the Frames Per Second (FPS) the computer can maintain vs. the number of sprites being drawn each frame:

Why is Arcade so fast? Arcade loads the sprites to the GPU and can redraw stationary sprites with almost no CPU effort. This allows it to scale drawing of stationary sprites to even 1 million plus, and still keep 60 FPS.

While Pygame's speed may drop off fast, there's still a few thousand sprites that can be drawn on the screen before FPS drops off. For many games that's plenty. Also, for sprites that don't move, Pygame programs can draw the sprites to a 'surface' at the start of a game. A program can then use that surface to the screen in one operation.

How fast can we draw moving sprites? Moving sprites are more challenging to draw, as we can't simply use what we did in the prior frame.

Arcade only updates the changed location of the sprite, keeping the dimensions and image on the GPU allowing it to still have fast updates.

Arcade also has two sprite classes available. The full-featured *arcade.Sprite* class and the smaller and faster *arcade.BasicSprite* class. If you don't need collision detection or physics support, the `BasicSprite` class works great.

31.3.2 Collision Processing

Another time-critical component in games is the time it takes to figure out if sprites collide:

Normally collision detection is an O(N) operation. That is, if are checking to see if a sprite collides with any of 1,000 other sprites, we have 1,000 checks to do. If there are a lot of sprites, this takes time.

Arcade has two ways to speed this up.

1. Spatial Hashing. If we know those 1,000 sprites aren't going to move at all (or very much) we can set up a grid. We figure out what grid location the player is in. Then we only check the player against whichever of the 1,000 sprites are in the same grid location. This works great for tiled maps where the platforms, ramps, etc. don't move. It gets us closer to O(1) time.
2. Off-load to the GPU. As there are 1,000s of processors on your graphics card, we can calculate collisions there. However it takes time to set up the GPU. This is only faster if we have more than 1500 or so sprites to check.
3. "Simple" checks everything. There are still a lot of tricks used to make this faster, and particularly with Python 3.11 code, it runs fine for most cases.

Arcade has multiple modes that allow you to select these collision options.

31.3.3 Shapes

Aside from sprites, how fast can a library draw various graphical shapes? Rectangle, circles, arcs, and more?

This next benchmark looks at drawing rectangles. Important things to keep in mind:

- Pygame uses memory bliting which is crazy fast and why it comes out in first-place. This doesn't work as well if you are drawing anything but unrotated rectangles.
- Arcade's shapes are easy, but crazy-slow. Thankfully you can use Pyglet shapes in the same program as Arcade. For anything more than a dozen or so shapes, a program should do that.
- Arcade has a Sprite class for solid-color rectangles. If you needed rectangles the *SpriteSolidColor* would be a high performance option not shown here.

What if a shape needs to be rotated? Pyglet can offload this to the GPU and this allows it to perform faster than Pygame that relies on the CPU.

CHAPTER

THIRTYTWO

API INDEX

- *arcade.color package*
- *arcade.csscolor package*
- *arcade.key package*
- *Built-In Resources*

32.1 The arcade module

Name	Group
arcade.shape_list.Shape	Shape Lists
arcade.shape_list.ShapeElementList	Shape Lists
arcade.shape_list.create_ellipse()	Shape Lists
arcade.shape_list.create_ellipse_filled()	Shape Lists
arcade.shape_list.create_ellipse_filled_with_colors()	Shape Lists
arcade.shape_list.create_ellipse_outline()	Shape Lists
arcade.shape_list.create_line()	Shape Lists
arcade.shape_list.create_line_generic()	Shape Lists
arcade.shape_list.create_line_generic_with_colors()	Shape Lists
arcade.shape_list.create_line_loop()	Shape Lists
arcade.shape_list.create_line_strip()	Shape Lists
arcade.shape_list.create_lines()	Shape Lists
arcade.shape_list.create_lines_with_colors()	Shape Lists
arcade.shape_list.create_polygon()	Shape Lists
arcade.shape_list.create_rectangle()	Shape Lists
arcade.shape_list.create_rectangle_filled()	Shape Lists
arcade.shape_list.create_rectangle_filled_with_colors()	Shape Lists
arcade.shape_list.create_rectangle_outline()	Shape Lists
arcade.shape_list.create_rectangles_filled_with_colors()	Shape Lists
arcade.shape_list.create_triangles_filled_with_colors()	Shape Lists

continues on next page

Table 1 – continued from previous page

Name	Group
arcade.shape_list.create_triangles_strip_filled_with_colors()	Shape Lists
arcade.shape_list.get_rectangle_points()	Shape Lists
arcade.NoOpenGLException	Window and View
arcade.View	Window and View
arcade.Window	Window and View
arcade.get_screens()	Window and View
arcade.open_window()	Window and View
arcade.close_window()	Window and View
arcade.exit()	Window and View
arcade.finish_render()	Window and View
arcade.get_display_size()	Window and View
arcade.get_window()	Window and View
arcade.pause()	Window and View
arcade.run()	Window and View
arcade.schedule()	Window and View
arcade.schedule_once()	Window and View
arcade.set_background_color()	Window and View
arcade.set_viewport()	Window and View
arcade.set_window()	Window and View
arcade.start_render()	Window and View
arcade.unschedule()	Window and View
arcade.draw_arc_filled()	Drawing - Primitives
arcade.draw_arc_outline()	Drawing - Primitives
arcade.draw_circle_filled()	Drawing - Primitives
arcade.draw_circle_outline()	Drawing - Primitives
arcade.draw_ellipse_filled()	Drawing - Primitives
arcade.draw_ellipse_outline()	Drawing - Primitives
arcade.draw_line()	Drawing - Primitives
arcade.draw_line_strip()	Drawing - Primitives
arcade.draw_lines()	Drawing - Primitives
arcade.draw_lrbt_rectangle_filled()	Drawing - Primitives
arcade.draw_lrbt_rectangle_outline()	Drawing - Primitives
arcade.draw_lrtb_rectangle_filled()	Drawing - Primitives
arcade.draw_lrtb_rectangle_outline()	Drawing - Primitives
arcade.draw_lrwh_rectangle_textured()	Drawing - Primitives
arcade.draw_parabola_filled()	Drawing - Primitives
arcade.draw_parabola_outline()	Drawing - Primitives
arcade.draw_point()	Drawing - Primitives
arcade.draw_points()	Drawing - Primitives
arcade.draw_polygon_filled()	Drawing - Primitives
arcade.draw_polygon_outline()	Drawing - Primitives
arcade.draw_rectangle_filled()	Drawing - Primitives
arcade.draw_rectangle_outline()	Drawing - Primitives
arcade.draw_scaled_texture_rectangle()	Drawing - Primitives
arcade.draw_texture_rectangle()	Drawing - Primitives
arcade.draw_triangle_filled()	Drawing - Primitives
arcade.draw_triangle_outline()	Drawing - Primitives
arcade.draw_xywh_rectangle_filled()	Drawing - Primitives
arcade.draw_xywh_rectangle_outline()	Drawing - Primitives

continues on next page

Table 1 – continued from previous page

Name	Group
arcade.get_image()	Drawing - Primitives
arcade.get_pixel()	Drawing - Primitives
arcade.ArcadeContext	OpenGL Context
arcade.types.Color	Types
arcade.types.TiledObject	Types
arcade.Camera	Camera
arcade.SimpleCamera	Camera
arcade.AStarBarrierList	Pathfinding
arcade.astar_calculate_path()	Pathfinding
arcade.has_line_of_sight()	Pathfinding
arcade.easing.EasingData	Easing
arcade.easing.ease_angle()	Easing
arcade.easing.ease_angle_update()	Easing
arcade.easing.ease_in()	Easing
arcade.easing.ease_in_back()	Easing
arcade.easing.ease_in_out()	Easing
arcade.easing.ease_in_out_sin()	Easing
arcade.easing.ease_in_sin()	Easing
arcade.easing.ease_out()	Easing
arcade.easing.ease_out_back()	Easing
arcade.easing.ease_out_bounce()	Easing
arcade.easing.ease_out_elastic()	Easing
arcade.easing.ease_out_sin()	Easing
arcade.easing.ease_position()	Easing
arcade.easing.ease_update()	Easing
arcade.easing.ease_value()	Easing
arcade.easing.easing()	Easing
arcade.easing.linear()	Easing
arcade.easing.smoothstep()	Easing
arcade.Sound	Sound
arcade.load_sound()	Sound
arcade.play_sound()	Sound
arcade.stop_sound()	Sound
arcade.utils.ByteRangeError	Misc Utility Functions
arcade.utils.FloatOutsideRangeError	Misc Utility Functions
arcade.utils.IntOutsideRangeError	Misc Utility Functions
arcade.utils.NormalizedRangeError	Misc Utility Functions
arcade.utils.OutsideRangeError	Misc Utility Functions
arcade.utils.PerformanceWarning	Misc Utility Functions
arcade.utils.ReplacementWarning	Misc Utility Functions
arcade.utils.generate_uuid_from_kwargs()	Misc Utility Functions
arcade.utils.get_raspberry_pi_info()	Misc Utility Functions
arcade.utils.is_raspberry_pi()	Misc Utility Functions
arcade.utils.warning()	Misc Utility Functions
arcade.Scene	Sprite Scenes
arcade.SceneKeyError	Sprite Scenes
arcade.PerfGraph	Performance Information
arcade.get_points_for_thick_line()	Drawing - Utility
arcade.math.clamp()	Math
arcade.math.get_angle_degrees()	Math

continues on next page

Table 1 – continued from previous page

Name	Group
arcade.math.get_angle_radians()	Math
arcade.math.get_distance()	Math
arcade.math.lerp()	Math
arcade.math.lerp_angle()	Math
arcade.math.lerp_vec()	Math
arcade.math.rand_angle_360_deg()	Math
arcade.math.rand_angle_spread_deg()	Math
arcade.math.rand_in_circle()	Math
arcade.math.rand_in_rect()	Math
arcade.math.rand_on_circle()	Math
arcade.math.rand_on_line()	Math
arcade.math.rand_vec_magnitude()	Math
arcade.math.rand_vec_spread_deg()	Math
arcade.math.rotate_point()	Math
arcade.math.round_fast()	Math
arcade.Text	Text
arcade.create_text_sprite()	Text
arcade.draw_text()	Text
arcade.load_font()	Text
arcade.clear_timings()	Performance Information
arcade.disable_timings()	Performance Information
arcade.enable_timings()	Performance Information
arcade.get_fps()	Performance Information
arcade.get_timings()	Performance Information
arcade.print_timings()	Performance Information
arcade.timings_enabled()	Performance Information
arcade.configure_logging()	Misc Utility Functions
arcade.geometry.are_lines_intersecting()	Geometry Support
arcade.geometry.are_polygons_intersecting()	Geometry Support
arcade.geometry.get_triangle_orientation()	Geometry Support
arcade.geometry.is_point_in_box()	Geometry Support
arcade.geometry.is_point_in_polygon()	Geometry Support
arcade.get_game_controllers()	Joystick Support
arcade.get_joysticks()	Joystick Support
arcade.isometric.create_isometric_grid_lines	Isometric Map Support (incomplete)
arcade.isometric.isometric_grid_to_screen()	Isometric Map Support (incomplete)
arcade.isometric.screen_to_isometric_grid()	Isometric Map Support (incomplete)
arcade.PymunkException	Physics Engines
arcade.PymunkPhysicsEngine	Physics Engines
arcade.PymunkPhysicsObject	Physics Engines
arcade.Section	Window and View
arcade.SectionManager	Window and View
arcade.earclip.earclip()	Earclip
arcade.PhysicsEnginePlatformer	Physics Engines
arcade.PhysicsEngineSimple	Physics Engines
arcade.ControllerManager	Game Controller Support
arcade.get_controllers()	Game Controller Support
arcade.check_for_collision()	Sprite Lists
arcade.check_for_collision_with_list()	Sprite Lists
arcade.check_for_collision_with_lists()	Sprite Lists

continues on next page

Table 1 – continued from previous page

Name	Group
arcade.get_closest_sprite()	Sprite Lists
arcade.get_distance_between_sprites()	Sprite Lists
arcade.get_sprites_at_exact_point()	Sprite Lists
arcade.get_sprites_at_point()	Sprite Lists
arcade.get_sprites_in_rect()	Sprite Lists
arcade.SpatialHash	Sprite Lists
arcade.SpriteList	Sprite Lists
arcade.PyMunk	Sprites
arcade.PymunkMixin	Sprites
arcade.SpriteCircle	Sprites
arcade.SpriteSolidColor	Sprites
arcade.AnimatedWalkingSprite	Sprites
arcade.TextureAnimation	Sprites
arcade.TextureAnimationSprite	Sprites
arcade.TextureKeyframe	Sprites
arcade.Sprite	Sprites
arcade.BasicSprite	Sprites
arcade.load_animated_gif()	Sprites
arcade.make_circle_texture()	Texture Management
arcade.make_soft_circle_texture()	Texture Management
arcade.make_soft_square_texture()	Texture Management
arcade.texture.transforms.FlipLeftRightTransform	Texture Transforms
arcade.texture.transforms.FlipTopBottomTransform	Texture Transforms
arcade.texture.transforms.Rotate180Transform	Texture Transforms
arcade.texture.transforms.Rotate270Transform	Texture Transforms
arcade.texture.transforms.Rotate90Transform	Texture Transforms
arcade.texture.transforms.Transform	Texture Transforms
arcade.texture.transforms.TransposeTransform	Texture Transforms
arcade.texture.transforms.TransverseTransform	Texture Transforms
arcade.texture.transforms.VertexOrder	Texture Transforms
arcade.texture.transforms.get_orientation()	Texture Transforms
arcade.cleanup_texture_cache()	Texture Management
arcade.get_default_image()	Texture Management
arcade.get_default_texture()	Texture Management
arcade.TextureManager	Texture Management
arcade.load_spritesheet()	Texture Management
arcade.load_texture()	Texture Management
arcade.load_texture_pair()	Texture Management
arcade.load_textures()	Texture Management
arcade.Texture	Texture Management
arcade.SpriteSheet	Texture Management
arcade.texture_atlas.AtlasRegion	Texture Atlas

continues on next page

Table 1 – continued from previous page

Name	Group
arcade.texture_atlas.TextureAtlas	Texture Atlas
arcade.texture_atlas.TextureAtlasBase	Texture Atlas
arcade.gui.UIDraggableMixin	GUI
arcade.gui.UIMouseFilterMixin	GUI
arcade.gui.UIWindowLikeMixin	GUI
arcade.gui.UIStyleBase	GUI Style
arcade.gui.UIStyledWidget	GUI Style
arcade.gui.Surface	GUI
arcade.gui.UIButtonRow	GUI
arcade.gui.UIMessageBox	GUI
arcade.gui.UIManager	GUI
arcade.gui.NinePatchTexture	GUI
arcade.gui.UIEvent	GUI Events
arcade.gui.UIKeyEvent	GUI Events
arcade.gui.UIKeyPressEvent	GUI Events
arcade.gui.UIKeyReleaseEvent	GUI Events
arcade.gui.UIMouseDragEvent	GUI Events
arcade.gui.UIMouseEvent	GUI Events
arcade.gui.UIMouseMovementEvent	GUI Events
arcade.gui.UIMousePressEvent	GUI Events
arcade.gui.UIMouseReleaseEvent	GUI Events
arcade.gui.UIMouseScrollEvent	GUI Events
arcade.gui.UIOnActionEvent	GUI Events
arcade.gui.UIOnChangeEvent	GUI Events
arcade.gui.UIOnClickEvent	GUI Events
arcade.gui.UIOnUpdateEvent	GUI Events
arcade.gui.UITextEvent	GUI Events
arcade.gui.UITextMotionEvent	GUI Events
arcade.gui.UITextMotionSelectEvent	GUI Events
arcade.gui.DictProperty	GUI Properties
arcade.gui.ListProperty	GUI Properties
arcade.gui.Property	GUI Properties
arcade.gui.bind()	GUI Properties
arcade.gui.UIImage	GUI Widgets
arcade.gui.UISlider	GUI Widgets
arcade.gui.UISliderStyle	GUI Widgets
arcade.gui.UIAnchorLayout	GUI Widgets
arcade.gui.UIBoxLayout	GUI Widgets
arcade.gui.UIGridLayout	GUI Widgets
arcade.gui.UIDropdown	GUI Widgets
arcade.gui.UIInputText	GUI Widgets
arcade.gui.UILabel	GUI Widgets
arcade.gui.UITextArea	GUI Widgets
arcade.gui.UITextWidget	GUI Widgets
arcade.gui.Rect	GUI Widgets
arcade.gui.UIDummy	GUI Widgets
arcade.gui.UIInteractiveWidget	GUI Widgets
arcade.gui.UILayout	GUI Widgets
arcade.gui.UISpace	GUI Widgets
arcade.gui.UISpriteWidget	GUI Widgets

continues on next page

Table 1 – continued from previous page

Name	Group
arcade.gui.UIWidget	GUI Widgets
arcade.gui.UITextureToggle	GUI Widgets
arcade.gui.UIFlatButton	GUI Widgets
arcade.gui.UITextureButton	GUI Widgets
arcade.gui.UITextureButtonStyle	GUI Widgets
arcade.tilemap.TileMap	Tiled Map Reader
arcade.tilemap.load_tilemap()	Tiled Map Reader
arcade.tilemap.read_tmx()	Tiled Map Reader

CHAPTER

THIRTYTHREE

API REFERENCE

This page documents the Application Programming Interface (API) for the Python Arcade library. See also:

- *API Index*
- *How-To Example Code*

33.1 Types

class arcade.types.**Color**(*r: int*, *g: int*, *b: int*, *a: int = 255*)

Bases: `Tuple[int, int, int, int]`

A `tuple` subclass representing an RGBA Color.

This class provides helpful utility methods and properties. When performance or brevity matters, arcade will usually allow you to use an ordinary `tuple` of RGBA values instead.

All channels are byte values from 0 to 255, inclusive. If any are outside this range, a *`ByteRangeError`* will be raised, which can be handled as a `ValueError`.

Examples:

```
>>> from arcade.types import Color
>>> Color(255, 0, 0)
Color(r=255, g=0, b=0, a=0)

>>> Color(*rgb_green_tuple, 127)
Color(r=0, g=255, b=0, a=127)
```

Parameters

- **r** – the red channel of the color, between 0 and 255
- **g** – the green channel of the color, between 0 and 255
- **b** – the blue channel of the color, between 0 and 255
- **a** – the alpha or transparency channel of the color, between 0 and 255

classmethod from_gray(*brightness: int*, *a: int = 255*) → Self

Return a shade of gray of the given brightness.

Example:

```
>>> custom_white = Color.from_gray(255)
>>> print(custom_white)
Color(r=255, g=255, b=255, a=255)

>>> half_opacity_gray = Color.from_gray(128, 128)
>>> print(half_opacity_gray)
Color(r=128, g=128, b=128, a=128)
```

Parameters

- **brightness** – How bright the shade should be
- **a** – a transparency value, fully opaque by default

Returns

classmethod from_hex_string(*code: str*) → Self

Make a color from a hex code that is 3, 4, 6, or 8 hex digits long

Prefixing it with a pound sign (`#` / hash symbol) is optional. It will be ignored if present.

The capitalization of the hex digits (`'f'` vs `'F'`) does not matter.

3 and 6 digit hex codes will be treated as if they have an opacity of 255.

3 and 4 digit hex codes will be expanded.

Examples:

```
>>> Color.from_hex_string("#ff00ff")
Color(r=255, g=0, b=255, a=255)

>>> Color.from_hex_string("#ff00ff00")
Color(r=255, g=0, b=255, a=0)

>>> Color.from_hex_string("#FFF")
Color(r=255, g=255, b=255, a=255)

>>> Color.from_hex_string("FF0A")
Color(r=255, g=255, b=0, a=170)
```

classmethod from_iterable(*iterable: Iterable[int]*) → Self

Create a color from an :py:class`Iterable` with 3-4 elements

If the passed iterable is already a Color instance, it will be returned unchanged. If the iterable has less than 3 or more than 4 elements, a ValueError will be raised.

Otherwise, the function will attempt to create a new Color instance. The usual rules apply, ie all values must be between 0 and 255, inclusive.

Parameters

iterable – An iterable which unpacks to 3 or 4 elements, each between 0 and 255, inclusive.

classmethod from_normalized(*color_normalized: Tuple[float, float, float, float]*) → Self

Convert normalized (0.0 to 1.0) channels into an RGBA Color

If the input channels aren't normalized, a `arcade.utils.NormalizedRangeError` will be raised. This is a subclass of :py:class`ValueError` and can be handled as such.

Examples:

```
>>> Color.from_normalized((1.0, 0.0, 0.0, 1.0))
Color(r=255, g=0, b=0, a=255)

>>> normalized_half_opacity_green = (0.0, 1.0, 0.0, 0.5)
>>> Color.from_normalized(normalized_half_opacity_green)
Color(r=0, g=255, b=0, a=127)
```

Parameters
`color_normalized` – The color as normalized (0.0 to 1.0) RGBA values.

Returns

classmethod from_uint24(*color: int*, *a: int = 255*) → Self

Return a Color from an unsigned 3-byte (24 bit) integer.

These ints may be between 0 and 16777215 (`0xFFFFFF`), inclusive.

Example:

```
>>> Color.from_uint24(16777215)
Color(r=255, g=255, b=255, a=255)

>>> Color.from_uint24(0xFF0000)
Color(r=255, g=0, b=0, a=255)
```

Parameters

- **color** – a 3-byte int between 0 and 16777215 (`0xFFFFFF`)
- **a** – an alpha value to use between 0 and 255, inclusive.

classmethod from_uint32(*color: int*) → Self

Return a Color tuple for a given unsigned 4-byte (32-bit) integer

The bytes are interpreted as R, G, B, A.

Examples:

```
>>> Color.from_uint32(4294967295)
Color(r=255, g=255, b=255, a=255)

>>> Color.from_uint32(0xFF0000FF)
Color(r=255, g=0, b=0, a=255)
```

Parameters
`color` – An int between 0 and 4294967295 (`0xFFFFFFFF`)

classmethod random(*r: int | None = None*, *g: int | None = None*, *b: int | None = None*, *a: int | None = None*) → Self

Return a random color.

The parameters are optional and can be used to fix the value of a particular channel. If a channel is not fixed, it will be randomly generated.

Examples:

```
# Randomize all channels
>>> Color.random()
Color(r=35, g=145, b=4, a=200)

# Random color with fixed alpha
>>> Color.random(a=255)
Color(r=25, g=99, b=234, a=255)
```

Parameters

- **r** – Fixed value for red channel
- **g** – Fixed value for green channel
- **b** – Fixed value for blue channel
- **a** – Fixed value for alpha channel

a

b

g

normalized

Return this color as a tuple of 4 normalized floats.

Examples:

```
>>> arcade.color.WHITE.normalized
(1.0, 1.0, 1.0, 1.0)

>>> arcade.color.BLACK.normalized
(0.0, 0.0, 0.0, 1.0)

>>> arcade.color.TRANSPARENT_BLACK.normalized
(0.0, 0.0, 0.0, 0.0)
```

r

class arcade.types.**TiledObject**(*shape*, *properties*, *name*, *type*)

Bases: `NamedTuple`

repr(self)

Return a nicely formatted representation string

name: `str | None`

Alias for field number 2

properties: `Dict[str, float | Path | str | bool | Color] | None`

Alias for field number 1

shape: `Tuple[float, float] | Sequence[Tuple[float, float]] | Tuple[int, int, int, int] | List[int]`

Alias for field number 0

type: `str | None`

Alias for field number 3

33.2 Drawing - Primitives

`arcade.draw_arc_filled`(*center_x: float*, *center_y: float*, *width: float*, *height: float*, *color: Tuple[int, int, int, int]*, *start_angle: float*, *end_angle: float*, *tilt_angle: float = 0*, *num_segments: int = 128*)

Draw a filled in arc. Useful for drawing pie-wedges, or Pac-Man.

Parameters

- **center_x** – x position that is the center of the arc.
- **center_y** – y position that is the center of the arc.
- **width** – width of the arc.
- **height** – height of the arc.
- **color** – A 3 or 4 length tuple of 0-255 channel values or a `Color` instance.
- **start_angle** – start angle of the arc in degrees.
- **end_angle** – end angle of the arc in degrees.
- **tilt_angle** – angle the arc is tilted (clockwise).
- **num_segments** – Number of line segments used to draw arc.

`arcade.draw_arc_outline`(*center_x: float*, *center_y: float*, *width: float*, *height: float*, *color: Tuple[int, int, int, int]*, *start_angle: float*, *end_angle: float*, *border_width: float = 1*, *tilt_angle: float = 0*, *num_segments: int = 128*)

Draw the outside edge of an arc. Useful for drawing curved lines.

Parameters

- **center_x** – x position that is the center of the arc.
- **center_y** – y position that is the center of the arc.
- **width** – width of the arc.
- **height** – height of the arc.
- **color** – A 3 or 4 length tuple of 0-255 channel values or a `Color` instance.
- **start_angle** – start angle of the arc in degrees.
- **end_angle** – end angle of the arc in degrees.
- **border_width** – width of line in pixels.
- **tilt_angle** – angle the arc is tilted (clockwise).
- **num_segments** – float of triangle segments that make up this circle. Higher is better quality, but slower render time.

`arcade.draw_circle_filled`(*center_x: float*, *center_y: float*, *radius: float*, *color: Tuple[int, int, int, int]*, *tilt_angle: float = 0*, *num_segments: int = -1*)

Draw a filled-in circle.

Parameters

- **center_x** – x position that is the center of the circle.
- **center_y** – y position that is the center of the circle.
- **radius** – width of the circle.

- **color** – A 3 or 4 length tuple of 0-255 channel values or a *Color* instance.
- **tilt_angle** – Angle in degrees to tilt the circle. Useful for low segment count circles
- **num_segments** – Number of triangle segments that make up this circle. Higher is better quality, but slower render time. The default value of -1 means arcade will try to calculate a reasonable amount of segments based on the size of the circle.

`arcade.draw_circle_outline`(*center_x: float*, *center_y: float*, *radius: float*, *color: Tuple[int, int, int, int]*, *border_width: float = 1*, *tilt_angle: float = 0*, *num_segments: int = -1*)

Draw the outline of a circle.

Parameters

- **center_x** – x position that is the center of the circle.
- **center_y** – y position that is the center of the circle.
- **radius** – width of the circle.
- **color** – A 3 or 4 length tuple of 0-255 channel values or a *Color* instance.
- **border_width** – Width of the circle outline in pixels.
- **tilt_angle** – Angle in degrees to tilt the circle (clockwise). Useful for low segment count circles
- **num_segments** – Number of triangle segments that make up this circle. Higher is better quality, but slower render time. The default value of -1 means arcade will try to calculate a reasonable amount of segments based on the size of the circle.

`arcade.draw_ellipse_filled`(*center_x: float*, *center_y: float*, *width: float*, *height: float*, *color: Tuple[int, int, int, int]*, *tilt_angle: float = 0*, *num_segments: int = -1*)

Draw a filled in ellipse.

Parameters

- **center_x** – x position that is the center of the circle.
- **center_y** – y position that is the center of the circle.
- **width** – width of the ellipse.
- **height** – height of the ellipse.
- **color** – A 3 or 4 length tuple of 0-255 channel values or a *Color* instance.
- **color** – Either a *Color* instance or an RGBA `tuple` of 4 byte values (0 to 255).
- **tilt_angle** – Angle in degrees to tilt the ellipse (clockwise). Useful when drawing a circle with a low segment count, to make an octagon for example.
- **num_segments** – Number of triangle segments that make up this circle. Higher is better quality, but slower render time. The default value of -1 means arcade will try to calculate a reasonable amount of segments based on the size of the circle.

`arcade.draw_ellipse_outline`(*center_x: float*, *center_y: float*, *width: float*, *height: float*, *color: Tuple[int, int, int, int]*, *border_width: float = 1*, *tilt_angle: float = 0*, *num_segments: int = -1*)

Draw the outline of an ellipse.

Parameters

- **center_x** – x position that is the center of the circle.
- **center_y** – y position that is the center of the circle.

- **width** – width of the ellipse.
- **height** – height of the ellipse.
- **color** – A 3 or 4 length tuple of 0-255 channel values or a `Color` instance.
- **border_width** – Width of the circle outline in pixels.
- **tilt_angle** – Angle in degrees to tilt the ellipse (clockwise). Useful when drawing a circle with a low segment count, to make an octagon for example.
- **num_segments** – Number of triangle segments that make up this circle. Higher is better quality, but slower render time. The default value of -1 means arcade will try to calculate a reasonable amount of segments based on the size of the circle.

arcade.**draw_line**(*start_x: float*, *start_y: float*, *end_x: float*, *end_y: float*, *color: Tuple[int, int, int, int]*, *line_width: float = 1*)

Draw a line.

Parameters

- **start_x** – x position of line starting point.
- **start_y** – y position of line starting point.
- **end_x** – x position of line ending point.
- **end_y** – y position of line ending point.
- **color** – A color, specified as an RGBA tuple or a `Color` instance.
- **line_width** – Width of the line in pixels.

arcade.**draw_line_strip**(*point_list: Sequence[Tuple[float, float]]*, *color: Tuple[int, int, int, int]*, *line_width: float = 1*)

Draw a multi-point line.

Parameters

- **point_list** – List of x, y points that make up this strip
- **color** – A color, specified as an RGBA tuple or a `Color` instance.
- **line_width** – Width of the line

arcade.**draw_lines**(*point_list: Sequence[Tuple[float, float]]*, *color: Tuple[int, int, int, int]*, *line_width: float = 1*)

Draw a set of lines.

Draw a line between each pair of points specified.

Parameters

- **point_list** – List of points making up the lines. Each point is in a list. So it is a list of lists.
- **color** – A color, specified as an RGBA tuple or a `Color` instance.
- **line_width** – Width of the line in pixels.

arcade.**draw_lrbt_rectangle_filled**(*left: float*, *right: float*, *bottom: float*, *top: float*, *color: Tuple[int, int, int, int]*)

Draw a rectangle by specifying left, right, bottom and top edges.

Parameters

- **left** – The x coordinate of the left edge of the rectangle.

- **right** – The x coordinate of the right edge of the rectangle.
- **bottom** – The y coordinate of the rectangle bottom.
- **top** – The y coordinate of the top of the rectangle.
- **color** – The color of the rectangle.

Raises ValueError
Raised if left > right or top < bottom.

arcade.**draw_lrbt_rectangle_outline**(*left: float*, *right: float*, *bottom: float*, *top: float*, *color: Tuple[int, int, int, int]*, *border_width: float = 1*)

Draw a rectangle by specifying left, right, bottom and top edges.

Parameters

- **left** – The x coordinate of the left edge of the rectangle.
- **right** – The x coordinate of the right edge of the rectangle.
- **bottom** – The y coordinate of the rectangle bottom.
- **top** – The y coordinate of the top of the rectangle.
- **color** – The color of the rectangle.
- **border_width** – The width of the border in pixels. Defaults to one.

Raises ValueError
Raised if left > right or top < bottom.

arcade.**draw_lrtb_rectangle_filled**(*left: float*, *right: float*, *top: float*, *bottom: float*, *color: Tuple[int, int, int, int]*)

Draw a rectangle by specifying left, right, top and bottom edges.

Deprecated since version 3.0: Use `draw_lrbt_rectangle_filled()` instead!

Parameters

- **left** – The x coordinate of the left edge of the rectangle.
- **right** – The x coordinate of the right edge of the rectangle.
- **top** – The y coordinate of the top of the rectangle.
- **bottom** – The y coordinate of the rectangle bottom.
- **color** – The color of the rectangle as an RGBA `tuple` or :py:class`~arcade.types.Color` instance.

Raises AttributeError
Raised if left > right or top < bottom.

arcade.**draw_lrtb_rectangle_outline**(*left: float*, *right: float*, *top: float*, *bottom: float*, *color: Tuple[int, int, int, int]*, *border_width: float = 1*)

Draw a rectangle by specifying left, right, top and bottom edges.

Deprecated since version 3.0: Use `draw_lrbt_rectangle_outline()` instead!

Parameters

- **left** – The x coordinate of the left edge of the rectangle.
- **right** – The x coordinate of the right edge of the rectangle.
- **top** – The y coordinate of the top of the rectangle.

- **bottom** – The y coordinate of the rectangle bottom.
- **color** – The color of the rectangle as an RGBA `tuple` or :py:class`~arcade.types.Color` instance.
- **border_width** – The width of the border in pixels. Defaults to one.

Raises AttributeError
Raised if left > right or top < bottom.

`arcade.`**`draw_lrwh_rectangle_textured`**(*bottom_left_x: float*, *bottom_left_y: float*, *width: float*, *height: float*, *texture:* Texture, *angle: float = 0*, *alpha: int = 255*)

Draw a texture extending from bottom left to top right.

Parameters

- **bottom_left_x** – The x coordinate of the left edge of the rectangle.
- **bottom_left_y** – The y coordinate of the bottom of the rectangle.
- **width** – The width of the rectangle.
- **height** – The height of the rectangle.
- **texture** – identifier of texture returned from load_texture() call
- **angle** – rotation of the rectangle. Defaults to zero (clockwise).
- **alpha** – Transparency of image. 0 is fully transparent, 255 (default) is visible

`arcade.`**`draw_parabola_filled`**(*start_x: float*, *start_y: float*, *end_x: float*, *height: float*, *color: Tuple[int, int, int, int]*, *tilt_angle: float = 0*)

Draws a filled in parabola.

Parameters

- **start_x** – The starting x position of the parabola
- **start_y** – The starting y position of the parabola
- **end_x** – The ending x position of the parabola
- **height** – The height of the parabola
- **color** – A 3 or 4 length tuple of 0-255 channel values or a `Color` instance.
- **tilt_angle** – The angle of the tilt of the parabola (clockwise)

`arcade.`**`draw_parabola_outline`**(*start_x: float*, *start_y: float*, *end_x: float*, *height: float*, *color: Tuple[int, int, int, int]*, *border_width: float = 1*, *tilt_angle: float = 0*)

Draws the outline of a parabola.

Parameters

- **start_x** – The starting x position of the parabola
- **start_y** – The starting y position of the parabola
- **end_x** – The ending x position of the parabola
- **height** – The height of the parabola
- **color** – A 3 or 4 length tuple of 0-255 channel values or a `Color` instance.
- **border_width** – The width of the parabola
- **tilt_angle** – The angle of the tilt of the parabola (clockwise)

`arcade.draw_point`(*x: float*, *y: float*, *color: Tuple[int, int, int, int]*, *size: float*)

Draw a point.

Parameters

- **x** – x position of point.
- **y** – y position of point.
- **color** – A color, specified as an RGBA tuple or a `Color` instance.
- **size** – Size of the point in pixels.

`arcade.draw_points`(*point_list: Sequence[Tuple[float, float]]*, *color: Tuple[int, int, int, int]*, *size: float = 1*)

Draw a set of points.

Parameters

- **point_list** – List of points Each point is in a list. So it is a list of lists.
- **color** – A color, specified as an RGBA tuple or a `Color` instance.
- **size** – Size of the point in pixels.

`arcade.draw_polygon_filled`(*point_list: Sequence[Tuple[float, float]]*, *color: Tuple[int, int, int, int]*)

Draw a polygon that is filled in.

Parameters

- **point_list** – List of points making up the lines. Each point is in a list. So it is a list of lists.
- **color** – The color, specified in RGB or RGBA format.

`arcade.draw_polygon_outline`(*point_list: Sequence[Tuple[float, float]]*, *color: Tuple[int, int, int, int]*, *line_width: float = 1*)

Draw a polygon outline. Also known as a "line loop."

Parameters

- **point_list** – List of points making up the lines. Each point is in a list. So it is a list of lists.
- **color** – The color of the outline as an RGBA `tuple` or `Color` instance.
- **line_width** – Width of the line in pixels.

`arcade.draw_rectangle_filled`(*center_x: float*, *center_y: float*, *width: float*, *height: float*, *color: Tuple[int, int, int, int]*, *tilt_angle: float = 0*)

Draw a filled-in rectangle.

Parameters

- **center_x** – x coordinate of rectangle center.
- **center_y** – y coordinate of rectangle center.
- **width** – width of the rectangle.
- **height** – height of the rectangle.
- **color** – The color of the rectangle as an RGBA `tuple` or :py:class`~arcade.types.Color` instance.
- **tilt_angle** – rotation of the rectangle (clockwise). Defaults to zero.

`arcade.`**`draw_rectangle_outline`**(*center_x: float*, *center_y: float*, *width: float*, *height: float*, *color: Tuple[int, int, int, int]*, *border_width: float = 1*, *tilt_angle: float = 0*)

Draw a rectangle outline.

Parameters

- **`center_x`** – x coordinate of top left rectangle point.
- **`center_y`** – y coordinate of top left rectangle point.
- **`width`** – width of the rectangle.
- **`height`** – height of the rectangle.
- **`color`** – The color of the rectangle as an RGBA `tuple` or :py:class`~arcade.types.Color` instance.
- **`border_width`** – width of the lines, in pixels.
- **`tilt_angle`** – rotation of the rectangle. Defaults to zero (clockwise).

`arcade.`**`draw_scaled_texture_rectangle`**(*center_x: float*, *center_y: float*, *texture: Texture*, *scale: float = 1.0*, *angle: float = 0*, *alpha: int = 255*)

Draw a textured rectangle on-screen.

Warning: This method can be slow!

Most users should consider using *`arcade.Sprite`* with *`arcade.SpriteList`* instead of this function.

OpenGL accelerates drawing by using batches to draw multiple things at once. This method doesn't do that.

If you need finer control or less overhead than arcade allows, consider pyglet's batching features.

Parameters

- **`center_x`** – x coordinate of rectangle center.
- **`center_y`** – y coordinate of rectangle center.
- **`texture`** – identifier of texture returned from load_texture() call
- **`scale`** – scale of texture
- **`angle`** – rotation of the rectangle (clockwise). Defaults to zero.
- **`alpha`** – Transparency of image. 0 is fully transparent, 255 (default) is fully visible

`arcade.`**`draw_texture_rectangle`**(*center_x: float*, *center_y: float*, *width: float*, *height: float*, *texture: Texture*, *angle: float = 0*, *alpha: int = 255*)

Draw a textured rectangle on-screen.

Parameters

- **`center_x`** – x coordinate of rectangle center.
- **`center_y`** – y coordinate of rectangle center.
- **`width`** – width of texture
- **`height`** – height of texture
- **`texture`** – identifier of texture returned from load_texture() call
- **`angle`** – rotation of the rectangle. Defaults to zero (clockwise).

- **alpha** – Transparency of image. 0 is fully transparent, 255 (default) is visible

arcade.**draw_triangle_filled**(*x1: float*, *y1: float*, *x2: float*, *y2: float*, *x3: float*, *y3: float*, *color: Tuple[int, int, int, int]*)

Draw a filled in triangle.

Parameters

- **x1** – x value of first coordinate.
- **y1** – y value of first coordinate.
- **x2** – x value of second coordinate.
- **y2** – y value of second coordinate.
- **x3** – x value of third coordinate.
- **y3** – y value of third coordinate.
- **color** – Color of the triangle as an RGBA `tuple` or `Color` instance.

arcade.**draw_triangle_outline**(*x1: float*, *y1: float*, *x2: float*, *y2: float*, *x3: float*, *y3: float*, *color: Tuple[int, int, int, int]*, *border_width: float = 1*)

Draw a the outline of a triangle.

Parameters

- **x1** – x value of first coordinate.
- **y1** – y value of first coordinate.
- **x2** – x value of second coordinate.
- **y2** – y value of second coordinate.
- **x3** – x value of third coordinate.
- **y3** – y value of third coordinate.
- **color** – RGBA255 of triangle as an RGBA `tuple` or :py:class`~arcade.types.Color` instance.
- **border_width** – Width of the border in pixels. Defaults to 1.

arcade.**draw_xywh_rectangle_filled**(*bottom_left_x: float*, *bottom_left_y: float*, *width: float*, *height: float*, *color: Tuple[int, int, int, int]*)

Draw a filled rectangle extending from bottom left to top right

Parameters

- **bottom_left_x** – The x coordinate of the left edge of the rectangle.
- **bottom_left_y** – The y coordinate of the bottom of the rectangle.
- **width** – The width of the rectangle.
- **height** – The height of the rectangle.
- **color** – The color of the rectangles an RGBA `tuple` or :py:class`~arcade.types.Color` instance.

arcade.**draw_xywh_rectangle_outline**(*bottom_left_x: float*, *bottom_left_y: float*, *width: float*, *height: float*, *color: Tuple[int, int, int, int]*, *border_width: float = 1*)

Draw a rectangle extending from bottom left to top right

Parameters

- **bottom_left_x** – The x coordinate of the left edge of the rectangle.
- **bottom_left_y** – The y coordinate of the bottom of the rectangle.
- **width** – The width of the rectangle.
- **height** – The height of the rectangle.
- **color** – The color of the rectangle as an RGBA `tuple` or :py:class`~arcade.types.Color` instance.
- **border_width** – The width of the border in pixels. Defaults to one.

`arcade.`**`get_image`**(*x: int = 0*, *y: int = 0*, *width: int | None = None*, *height: int | None = None*) → Image

Get an image from the screen.

Example:

```
image = get_image()
image.save('screenshot.png', 'PNG')
```

Parameters

- **x** – Start (left) x location
- **y** – Start (top) y location
- **width** – Width of image. Leave blank for grabbing the 'rest' of the image
- **height** – Height of image. Leave blank for grabbing the 'rest' of the image

Returns
A Pillow Image

`arcade.`**`get_pixel`**(*x: int*, *y: int*, *components: int = 3*) → Tuple[int, ...]

Given an x, y, will return a color value of that point.

Parameters

- **x** – x location
- **y** – y location
- **components** – Number of components to fetch. By default we fetch 3 3 components (RGB). 4 components would be RGBA.

33.3 Shape Lists

class `arcade.shape_list.`**`Shape`**(*points: Sequence[Tuple[float, float]]*, *colors: Sequence[Tuple[int, int, int, int]]*, *mode: int = 4*, *program: Program | None = None*)

Bases:

A container for arbitrary geometry representing a shape.

This shape can be drawn using the draw() method, or added to a ShapeElementList for drawing in batch.

Parameters

- **points** – A list of points that make up the shape.
- **colors** – A list of colors that correspond to the points.

- **mode** – The OpenGL drawing mode. Defaults to GL_TRIANGLES.
- **program** – The program to use when drawing this shape (Shape.draw() only)

`draw()`

Draw this shape. Drawing this way isn't as fast as drawing multiple shapes batched together in a ShapeElementList.

`class arcade.shape_list.ShapeElementList`

Bases: `Generic`[TShape]

A ShapeElementList is a list of shapes that can be drawn together in a back for better performance. ShapeElementLists are suited for drawing a large number of shapes that are static. If you need to move a lot of shapes it's better to use pyglet's shape system.

Adding new shapes is fast, but removing them is slow.

`iter`(self) → Iterable[TShape]

Return an iterable object of sprites.

`len`(self) → int

Return the length of the sprite list.

`append`(*item: TShape*)

Add a new shape to the list.

`clear`(*position: bool = True*, *angle: bool = True*) → None

Clear all the contents from the shape list.

Parameters

- **position** – Reset the position to 0,0
- **angle** – Reset the angle to 0

`draw()` → None

Draw all the shapes.

`move`(*change_x: float*, *change_y: float*)

Change the center_x/y of the shape list relative to the current position.

Parameters

- **change_x** – Amount to move on the x axis
- **change_y** – Amount to move on the y axis

`remove`(*item: TShape*)

Remove a specific shape from the list.

`update()` → None

Update the internals of the shape list. This is automatically called when you call draw().

In some instances you may need to call this manually to update the shape list before drawing.

`angle`

Get or set the rotation in degrees (clockwise)

`center_x`

Get or set the center x coordinate of the ShapeElementList.

`center_y`

Get or set the center y coordinate of the ShapeElementList.

`position`

Get or set the position of the ShapeElementList.

This is the equivalent of setting center_x and center_y

`arcade.shape_list.`**`create_ellipse`**(*center_x: float, center_y: float, width: float, height: float, color: Tuple[int, int, int, int], border_width: float = 1, tilt_angle: float = 0, num_segments: int = 32, filled: bool = True*) → *Shape*

This creates an ellipse vertex buffer object (VBO).

The function returns a Shape object that can be drawn with `my_shape.draw()`. Don't create the shape in the draw method, create it in the setup method and then draw it in `on_draw`.

For even faster performance, add multiple shapes into a ShapeElementList and draw that list. This allows nearly unlimited shapes to be drawn just as fast as one.

Parameters

- **`center_x`** – X position of the center of the ellipse.
- **`center_y`** – Y position of the center of the ellipse.
- **`width`** – Width of the ellipse.
- **`height`** – Height of the ellipse.
- **`color`** – Color of the ellipse.
- **`border_width`** – Width of the border.
- **`tilt_angle`** – Angle to tilt the ellipse.
- **`num_segments`** – Number of segments to use to draw the ellipse.
- **`filled`** – If True, create a filled ellipse. If False, create an outline.

`arcade.shape_list.`**`create_ellipse_filled`**(*center_x: float, center_y: float, width: float, height: float, color: Tuple[int, int, int, int], tilt_angle: float = 0, num_segments: int = 128*) → *Shape*

Create a filled ellipse. Or circle if you use the same width and height.

The function returns a Shape object that can be drawn with `my_shape.draw()`. Don't create the shape in the draw method, create it in the setup method and then draw it in `on_draw`.

For even faster performance, add multiple shapes into a ShapeElementList and draw that list. This allows nearly unlimited shapes to be drawn just as fast as one.

`arcade.shape_list.`**`create_ellipse_filled_with_colors`**(*center_x: float, center_y: float, width: float, height: float, outside_color: Tuple[int, int, int, int], inside_color: Tuple[int, int, int, int], tilt_angle: float = 0, num_segments: int = 32*) → *Shape*

Draw an ellipse, and specify inside/outside color. Used for doing gradients.

The function returns a Shape object that can be drawn with `my_shape.draw()`. Don't create the shape in the draw method, create it in the setup method and then draw it in `on_draw`.

For even faster performance, add multiple shapes into a ShapeElementList and draw that list. This allows nearly unlimited shapes to be drawn just as fast as one.

Parameters

- **center_x** – X position of the center of the ellipse.
- **center_y** – Y position of the center of the ellipse.
- **width** – Width of the ellipse.
- **height** – Height of the ellipse.
- **outside_color** – Color of the outside of the ellipse.
- **inside_color** – Color of the inside of the ellipse.
- **tilt_angle** – Angle to tilt the ellipse.
- **num_segments** – Number of segments to use to draw the ellipse.

`arcade.shape_list.`**`create_ellipse_outline`**(*center_x: float, center_y: float, width: float, height: float, color: Tuple[int, int, int, int], border_width: float = 1, tilt_angle: float = 0, num_segments: int = 128*) → *Shape*

Create an outline of an ellipse.

The function returns a Shape object that can be drawn with `my_shape.draw()`. Don't create the shape in the draw method, create it in the setup method and then draw it in `on_draw`.

For even faster performance, add multiple shapes into a ShapeElementList and draw that list. This allows nearly unlimited shapes to be drawn just as fast as one.

`arcade.shape_list.`**`create_line`**(*start_x: float, start_y: float, end_x: float, end_y: float, color: Tuple[int, int, int, int], line_width: float = 1*) → *Shape*

Create a Shape object for a line.

Parameters

- **start_x** – Starting x position
- **start_y** – Starting y position
- **end_x** – Ending x position
- **end_y** – Ending y position
- **color** – Color of the line
- **line_width** – Width of the line

`arcade.shape_list.`**`create_line_generic`**(*point_list: Sequence[Tuple[float, float]], color: Tuple[int, int, int, int], shape_mode: int*) → *Shape*

This function is used by `create_line_strip` and `create_line_loop`, just changing the OpenGL type for the line drawing.

Parameters

- **point_list** – A list of points that make up the shape.
- **color** – A color such as a `Color`
- **shape_mode** – The OpenGL drawing mode. Defaults to GL_TRIANGLES.

`arcade.shape_list.`**`create_line_generic_with_colors`**(*point_list: Sequence[Tuple[float, float]], color_sequence: Sequence[Tuple[int, int, int, int]], shape_mode: int*) → *Shape*

This function is used by `create_line_strip` and `create_line_loop`, just changing the OpenGL type for the line drawing.

Parameters

- **point_list** – A list of points that make up the shape.
- **color_sequence** – A sequence of colors such as a `list`; each color must be either a `Color` instance or a 4-length RGBA `tuple`.
- **shape_mode** – The OpenGL drawing mode. Defaults to GL_TRIANGLES.

`arcade.shape_list.`**`create_line_loop`**(*point_list: Sequence[Tuple[float, float]], color: Tuple[int, int, int, int], line_width: float = 1*) → *Shape*

Create a multi-point line loop to be rendered later. This works faster than draw_line because the vertexes are only loaded to the graphics card once, rather than each frame.

Parameters

- **point_list** – A list of points that make up the shape.
- **color** – A color such as a `Color`
- **line_width** – Width of the line

`arcade.shape_list.`**`create_line_strip`**(*point_list: Sequence[Tuple[float, float]], color: Tuple[int, int, int, int], line_width: float = 1*) → *Shape*

Create a multi-point line to be rendered later. This works faster than draw_line because the vertexes are only loaded to the graphics card once, rather than each frame.

Internally, thick lines are created by two triangles.

Parameters

- **point_list** –
- **color** –
- **line_width** –

`arcade.shape_list.`**`create_lines`**(*point_list: Sequence[Tuple[float, float]], color: Tuple[int, int, int, int]*) → *Shape*

Create a multi-point line loop to be rendered later. This works faster than draw_line because the vertexes are only loaded to the graphics card once, rather than each frame.

Parameters

- **point_list** – A list of points that make up the shape.
- **color** – A color such as a `Color`
- **line_width** – Width of the line

`arcade.shape_list.`**`create_lines_with_colors`**(*point_list: Sequence[Tuple[float, float]], color_list: Sequence[Tuple[int, int, int, int]], line_width: float = 1*) → *Shape*

Create a line segments to be rendered later. This works faster than draw_line because the vertexes are only loaded to the graphics card once, rather than each frame.

Parameters

- **point_list** – Line segments start and end point tuples list
- **color_list** – Three or four byte tuples list for every point
- **line_width** – Width of the line

Returns Shape

`arcade.shape_list.`**`create_polygon`**(*point_list: Sequence[Tuple[float, float]], color: Tuple[int, int, int, int]*) → *Shape*

Draw a convex polygon. This will NOT draw a concave polygon. Because of this, you might not want to use this function.

The function returns a Shape object that can be drawn with `my_shape.draw()`. Don't create the shape in the draw method, create it in the setup method and then draw it in `on_draw`.

For even faster performance, add multiple shapes into a ShapeElementList and draw that list. This allows nearly unlimited shapes to be drawn just as fast as one.

Parameters

- **`point_list`** – A list of points that make up the shape.
- **`color`** – A color such as a `Color`

`arcade.shape_list.`**`create_rectangle`**(*center_x: float, center_y: float, width: float, height: float, color: Tuple[int, int, int, int], border_width: float = 1, tilt_angle: float = 0, filled=True*) → *Shape*

This function creates a rectangle using a vertex buffer object.

The function returns a Shape object that can be drawn with `my_shape.draw()`. Don't create the shape in the draw method, create it in the setup method and then draw it in `on_draw`.

For even faster performance, add multiple shapes into a ShapeElementList and draw that list. This allows nearly unlimited shapes to be drawn just as fast as one.

Parameters

- **`center_x`** – X position of the center of the rectangle
- **`center_y`** – Y position of the center of the rectangle
- **`width`** – Width of the rectangle
- **`height`** – Height of the rectangle
- **`color`** – A color such as a `Color`
- **`border_width`** – Width of the border
- **`tilt_angle`** – Angle to tilt the rectangle in degrees
- **`filled`** – If True, the rectangle is filled. If False, it is an outline.

`arcade.shape_list.`**`create_rectangle_filled`**(*center_x: float, center_y: float, width: float, height: float, color: Tuple[int, int, int, int], tilt_angle: float = 0*) → *Shape*

Create a filled rectangle.

The function returns a Shape object that can be drawn with `my_shape.draw()`. Don't create the shape in the draw method, create it in the setup method and then draw it in `on_draw`.

For even faster performance, add multiple shapes into a ShapeElementList and draw that list. This allows nearly unlimited shapes to be drawn just as fast as one.

Parameters

- **`center_x`** – X position of the center of the rectangle
- **`center_y`** – Y position of the center of the rectangle
- **`width`** – Width of the rectangle
- **`height`** – Height of the rectangle

- **color** – A color such as a *Color*
- **tilt_angle** – Angle to tilt the rectangle in degrees

`arcade.shape_list.`**`create_rectangle_filled_with_colors`**(*point_list*, *color_list*) → *Shape*

This function creates one rectangle/quad using a vertex buffer object.

The function returns a Shape object that can be drawn with `my_shape.draw()`. Don't create the shape in the draw method, create it in the setup method and then draw it in `on_draw`.

For even faster performance, add multiple shapes into a ShapeElementList and draw that list. This allows nearly unlimited shapes to be drawn just as fast as one.

Parameters

- **point_list** – List of points to create the rectangle from
- **color_list** – List of colors to create the rectangle from

`arcade.shape_list.`**`create_rectangle_outline`**(*center_x: float*, *center_y: float*, *width: float*, *height: float*, *color: Tuple[int, int, int, int]*, *border_width: float = 1*, *tilt_angle: float = 0*) → *Shape*

Create a rectangle outline.

The function returns a Shape object that can be drawn with `my_shape.draw()`. Don't create the shape in the draw method, create it in the setup method and then draw it in `on_draw`.

For even faster performance, add multiple shapes into a ShapeElementList and draw that list. This allows nearly unlimited shapes to be drawn just as fast as one.

Parameters

- **center_x** – X position of the center of the rectangle
- **center_y** – Y position of the center of the rectangle
- **width** – Width of the rectangle
- **height** – Height of the rectangle
- **color** – A color such as a *Color*
- **border_width** – Width of the border
- **tilt_angle** – Angle to tilt the rectangle in degrees

`arcade.shape_list.`**`create_rectangles_filled_with_colors`**(*point_list*, *color_list: Sequence[Tuple[int, int, int, int]]*) → *Shape*

This function creates multiple rectangle/quads using a vertex buffer object.

The function returns a Shape object that can be drawn with `my_shape.draw()`. Don't create the shape in the draw method, create it in the setup method and then draw it in `on_draw`.

For even faster performance, add multiple shapes into a ShapeElementList and draw that list. This allows nearly unlimited shapes to be drawn just as fast as one.

`arcade.shape_list.`**`create_triangles_filled_with_colors`**(*point_list: Sequence[Tuple[float, float]]*, *color_sequence: Sequence[Tuple[int, int, int, int]]*) → *Shape*

This function creates multiple triangles using a vertex buffer object. Triangles are build for every 3 sequential vertices with step of 3 vertex Total amount of triangles to be rendered: len(point_list) / 3

The function returns a Shape object that can be drawn with `my_shape.draw()`. Don't create the shape in the draw method, create it in the setup method and then draw it in `on_draw`.

For even faster performance, add multiple shapes into a ShapeElementList and draw that list. This allows nearly unlimited shapes to be drawn just as fast as one.

Parameters

- **`point_list`** – Triangles vertices tuples.
- **`color_sequence`** – A sequence of colors such as a `list`; each color must be either a *`Color`* instance or a 4-length RGBA `tuple`.

`arcade.shape_list.`**`create_triangles_strip_filled_with_colors`**(*point_list*, *color_sequence: Sequence[Tuple[int, int, int, int]]*) → *Shape*

This function creates multiple triangles using a vertex buffer object. Triangles are built for every 3 sequential vertices with step of 1 vertex Total amount of triangles to be rendered: len(point_list) - 2

The function returns a Shape object that can be drawn with `my_shape.draw()`. Don't create the shape in the draw method, create it in the setup method and then draw it in `on_draw`.

For even faster performance, add multiple shapes into a ShapeElementList and draw that list. This allows nearly unlimited shapes to be drawn just as fast as one.

Parameters

- **`point_list`** – Triangles vertices tuples.
- **`color_sequence`** – A sequence of colors such as a `list`; each color must be either a *`Color`* instance or a 4-length RGBA `tuple`.

`arcade.shape_list.`**`get_rectangle_points`**(*center_x: float*, *center_y: float*, *width: float*, *height: float*, *tilt_angle: float = 0*) → Sequence[Tuple[float, float]]

Utility function that will return all four coordinate points of a rectangle given the x, y center, width, height, and rotation.

Parameters

- **`center_x`** – X position of the center of the rectangle
- **`center_y`** – Y position of the center of the rectangle
- **`width`** – Width of the rectangle
- **`height`** – Height of the rectangle
- **`tilt_angle`** – Angle to tilt the rectangle in degrees

33.4 Drawing - Utility

`arcade.`**`get_points_for_thick_line`**(*start_x: float*, *start_y: float*, *end_x: float*, *end_y: float*, *line_width: float*)

Function used internally for Arcade. OpenGL draws triangles only, so a thick line must be two triangles that make up a rectangle. This calculates and returns those points.

33.5 Sprites

class arcade.**PyMunk**

Bases:

Object used to hold pymunk info for a sprite.

damping

gravity

max_horizontal_velocity

max_velocity

max_vertical_velocity

class arcade.**PymunkMixin**

Bases:

pymunk_moved(*physics_engine*, *dx*, *dy*, *d_angle*)

Called by the pymunk physics engine if this sprite moves.

class arcade.**SpriteCircle**(*radius: int*, *color: Tuple[int, int, int, int]*, *soft: bool = False*, ***kwargs*)

Bases: `Sprite`

A circle of the specified radius.

The texture is automatically generated instead of loaded from a file.

There may be a stutter the first time a combination of `radius`, `color`, and `soft` is used due to texture generation. All subsequent calls for the same combination will run faster because they will re-use the texture generated earlier.

For a gradient fill instead of a solid color, set `soft` to `True`. The circle will fade from an opaque center to transparent at the edges.

Parameters

- **radius** – Radius of the circle in pixels
- **color** – The Color of the sprite as an RGB or RGBA tuple
- **soft** – If `True`, the circle will fade from an opaque center to transparent edges.

boundary_bottom: float | None

boundary_left: float | None

boundary_right: float | None

boundary_top: float | None

change_angle: float

cur_texture_index: int

force

guid: str | None

physics_engines: List[Any]

textures: List[*Texture*]

class arcade.**SpriteSolidColor**(*width: int, height: int, center_x: float = 0, center_y: float = 0, color: Tuple[int, int, int, int] = (255, 255, 255, 255), angle: float = 0, **kwargs*)

Bases: *Sprite*

A rectangular sprite of the given `width`, `height`, and `color`.

The texture is automatically generated instead of loaded from a file. Internally only a single global texture is used for this sprite type, so concerns about memory usage non-existent regardless of size or number of sprite variations.

Parameters

- **width** – Width of the sprite in pixels
- **height** – Height of the sprite in pixels
- **center_x** – Initial x position of the sprite
- **center_y** – Initial y position of the sprite
- **color** – The color of the sprite as a *Color*, an RGBA tuple, or an RGB tuple.
- **angle** – Initial angle of the sprite in degrees

class arcade.**AnimatedWalkingSprite**(*scale: float = 1.0, center_x: float = 0.0, center_y: float = 0.0, **kwargs*)

Bases: *Sprite*

Deprecated Sprite for platformer games that supports walking animations. Make sure to call update_animation after loading the animations so the initial texture can be set. Or manually set it.

It is highly recommended you create your own version of this class rather than try to use this pre-packaged one.

For an example, see this section of the platformer tutorial: *Step 12 - Loading a Map From a Map Editor*.

Parameters

- **scale** – Initial scale of the sprite.
- **center_x** – Initial x position of the sprite.
- **center_y** – Initial y position of the sprite.

update_animation(*delta_time: float = 0.016666666666666666*) → None

Logic for texture animation.

Parameters
delta_time – Time since last update.

boundary_bottom: float | None

boundary_left: float | None

boundary_right: float | None

boundary_top: float | None

change_angle: float

cur_texture_index: int

force

guid: str | None

physics_engines: List[Any]

textures: List[*Texture*]

class arcade.**TextureAnimation**(*keyframes: List[TextureKeyframe] | None = None*)

Bases:

Animation class that holds a list of keyframes. The animation should not store any state related to the current time so it can be shared between multiple sprites.

Parameters

- **keyframes** – List of keyframes for the animation.
- **loop** – If the animation should loop.

append_keyframe(*keyframe: TextureKeyframe*) → None

Add a keyframe to the animation.

Parameters
keyframe – Keyframe to add.

get_keyframe(*time: float, loop: bool = True*) → Tuple[int, *TextureKeyframe*]

Get the frame at a given time.

Parameters

- **time** – Time in seconds.
- **loop** – If the animation should loop.

Returns
Tuple of frame index and keyframe.

remove_keyframe(*index: int*) → None

Remove a keyframe from the animation.

Parameters
index – Index of the keyframe to remove.

duration_ms

Total duration of the animation in milliseconds.

duration_seconds

Total duration of the animation in seconds.

keyframes

A tuple of keyframes in the animation. Keyframes should not be modified directly.

num_frames

Number of frames in the animation.

class arcade.**TextureAnimationSprite**(*center_x: float = 0.0, center_y: float = 0.0, scale: float = 1.0, animation: TextureAnimation | None = None, **kwargs*)

Bases: *Sprite*

Animated sprite based on keyframes. Primarily used internally by tilemaps.

Parameters

- **path_or_texture** – Path to the image file, or a Texture object.

- **center_x** – Initial x position of the sprite.
- **center_y** – Initial y position of the sprite.
- **scale** – Initial scale of the sprite.

update_animation(*delta_time: float = 0.016666666666666666*, ***kwargs*) → None

Logic for updating the animation.

Parameters
delta_time – Time since last update.

animation

Animation object for this sprite.

boundary_bottom: float | None

boundary_left: float | None

boundary_right: float | None

boundary_top: float | None

change_angle: float

cur_texture_index: int

force

guid: str | None

physics_engines: List[Any]

textures: List[*Texture*]

time

Get or set the current time of the animation in seconds.

class arcade.**TextureKeyframe**(*texture: Texture*, *duration: int = 100*, *tile_id: int | None = 0*, ***kwargs*)

Bases:

Keyframe for texture animations.

Parameters

- **texture** – Texture to display for this keyframe.
- **duration** – Duration in milliseconds to display this keyframe.
- **tile_id** – Tile ID for this keyframe (only used for tiled maps)

duration

Duration in milliseconds to display this keyframe.

texture

The texture to display for this keyframe.

tile_id

Tile ID for this keyframe (only used for tiled maps)

class arcade.**Sprite**(*path_or_texture: str | Path | Texture | None = None, scale: float = 1.0, center_x: float = 0.0, center_y: float = 0.0, angle: float = 0.0, **kwargs: Any*)

Bases: `BasicSprite`, `PymunkMixin`

Sprites are used to render image data to the screen & perform collisions.

Most games center around sprites. They are most frequently used as follows:

1. Create `Sprite` instances from image data
2. Add the sprites to a `SpriteList` instance
3. Call `SpriteList.draw()` on the instance inside your `on_draw` method.

For runnable examples of how to do this, please see arcade's *built-in Sprite examples*.

Tip: Advanced users should see `BasicSprite`

It uses fewer resources at the cost of having fewer features.

Parameters

- **path_or_texture** – Path to an image file, or a texture object.
- **center_x** – Location of the sprite in pixels.
- **center_y** – Location of the sprite in pixels.
- **scale** – Show the image at this many times its original size.
- **angle** – The initial rotation of the sprite in degrees

append_texture(*texture: Texture*)

Appends a new texture to the list of textures that can be applied to this sprite.

Parameters
texture – Texture to add to the list of available textures

draw(**, filter: Tuple[int, int] | None = None, pixelated: bool | None = None, blend_function: Tuple[int, int] | Tuple[int, int, int, int] | None = None*) → None

A debug method which draws the sprite into the current OpenGL context.

Warning: You are probably looking for `SpriteList.draw()`!

Drawing individual sprites is slow compared to using `SpriteList`. See *Why SpriteLists?* for more information.

This method should not be relied on. It may be removed one day.

Parameters

- **filter** – Optional parameter to set OpenGL filter, such as *gl.GL_NEAREST* to avoid smoothing.
- **pixelated** – `True` for pixelated and `False` for smooth interpolation. Shortcut for setting filter=GL_NEAREST.
- **blend_function** – Optional parameter to set the OpenGL blend function used for drawing the sprite list, such as 'arcade.Window.ctx.BLEND_ADDITIVE' or 'arcade.Window.ctx.BLEND_DEFAULT'

forward(*speed: float = 1.0*) → None

Adjusts a Sprites forward.

Parameters
speed – speed

register_physics_engine(*physics_engine: Any*) → None

Register a physics engine on the sprite. This is only needed if you actually need a reference to your physics engine in the sprite itself. It has no other purposes.

The registered physics engines can be accessed through the `physics_engines` attribute.

It can for example be the pymunk physics engine or a custom one you made.

remove_from_sprite_lists() → None

Remove this sprite from all sprite lists it is in including registered physics engines.

reverse(*speed: float = 1.0*) → None

Adjusts a Sprite backwards.

Parameters
speed – speed

set_texture(*texture_no: int*) → None

Set the current texture by texture number. The number is the index into `self.textures`.

Parameters
texture_no – Index into `self.textures`

stop() → None

Stop the Sprite's motion by setting the velocity and angle change to 0.

strafe(*speed: float = 1.0*) → None

Adjusts a Sprite sideways.

Parameters
speed – speed

sync_hit_box_to_texture()

Update the sprite's hit box to match the current texture's hit box.

turn_left(*theta: float = 90.0*) → None

Rotate the sprite left by the passed number of degrees.

Parameters
theta – change in angle, in degrees

turn_right(*theta: float = 90.0*) → None

Rotate the sprite right by the passed number of degrees.

Parameters
theta – change in angle, in degrees

update() → None

The default update method for a Sprite. Can be overridden by a subclass.

This method moves the sprite based on its velocity and angle change.

update_spatial_hash() → None

Update the sprites location in the spatial hash.

angle

Get or set the rotation or the sprite.

The value is in degrees and is clockwise.

boundary_bottom: float | None

boundary_left: float | None

boundary_right: float | None

boundary_top: float | None

change_angle: float

change_x

Get or set the velocity in the x plane of the sprite.

change_y

Get or set the velocity in the y plane of the sprite.

cur_texture_index: int

force

guid: str | None

hit_box

Get or set the hit box for this sprite.

physics_engines: List[Any]

properties

Get or set custom sprite properties.

radians

Get or set the rotation of the sprite in radians.

The value is in radians and is clockwise.

texture

Get or set the active texture for this sprite

textures: List[*Texture*]

velocity

Get or set the velocity of the sprite.

The x and y velocity can also be set separately using the `sprite.change_x` and `sprite.change_y` properties.

Example:

```
sprite.velocity = 1.0, 0.0
```

Returns
Tuple[float, float]

class arcade.**BasicSprite**(*texture: Texture*, *scale: float = 1.0*, *center_x: float = 0*, *center_y: float = 0*, ***kwargs: Any*)

Bases:

The absolute minimum needed for a sprite.

It does not support features like rotation or changing the hitbox after creation. For more built-in features, please see `Sprite`.

Parameters

- **texture** – The texture data to use for this sprite.
- **scale** – The scaling factor for drawing the texture.
- **center_x** – Location of the sprite along the X axis in pixels.
- **center_y** – Location of the sprite along the Y axis in pixels.

collides_with_list(*sprite_list: SpriteList*) → List[SpriteType]

Check if current sprite is overlapping with any other sprite in a list

Parameters
sprite_list – SpriteList to check against

Returns
List of all overlapping Sprites from the original SpriteList

collides_with_point(*point: Tuple[float, float]*) → bool

Check if point is within the current sprite.

Parameters
point – Point to check.

Returns
True if the point is contained within the sprite's boundary.

collides_with_sprite(*other: SpriteType*) → bool

Will check if a sprite is overlapping (colliding) another Sprite.

Parameters
other – the other sprite to check against.

Returns
True or False, whether or not they are overlapping.

draw_hit_box(*color: Tuple[int, int, int, int] = (0, 0, 0, 255)*, *line_thickness: float = 2.0*) → None

Draw a sprite's hit-box. This is useful for debugging.

Parameters

- **color** – Color of box
- **line_thickness** – How thick the box should be

kill() → None

Alias of `remove_from_sprite_lists()`.

on_update(*delta_time: float = 0.016666666666666666*) → None

Update the sprite. Similar to update, but also takes a delta-time. It can be called manually or by the SpriteList's on_update method.

Parameters
delta_time – Time since last update.

register_sprite_list(*new_list:* SpriteList) → None

Register this sprite as belonging to a list. We will automatically remove ourselves from the list when kill() is called.

remove_from_sprite_lists() → None

Remove the sprite from all sprite lists.

rescale_relative_to_point(*point: Tuple[float, float]*, *factor: float*) → None

Rescale the sprite and its distance from the passed point.

This function does two things:

1. Multiply both values in the sprite's *scale_xy* value by factor.
2. Scale the distance between the sprite and point by factor.

If point equals the sprite's *position*, the distance will be zero and the sprite will not move.

Parameters

- **point** – The reference point for rescaling.
- **factor** – Multiplier for sprite scale & distance to point.

Returns

rescale_xy_relative_to_point(*point: Tuple[float, float]*, *factors_xy: Iterable[float]*) → None

Rescale the sprite and its distance from the passed point.

This method can scale by different amounts on each axis. To scale along only one axis, set the other axis to 1.0 in factors_xy.

Internally, this function does the following:

1. Multiply the x & y of the sprite's *scale_xy* attribute by the corresponding part from factors_xy.
2. Scale the x & y of the difference between the sprite's position and point by the corresponding component from factors_xy.

If point equals the sprite's *position*, the distance will be zero and the sprite will not move.

Parameters

- **point** – The reference point for rescaling.
- **factors_xy** – A 2-length iterable containing x and y multipliers for scale & distance to point.

Returns

update() → None

Generic update method. It can be called manually or by the SpriteList's update method.

update_animation(*delta_time: float = 0.016666666666666666*) → None

Generic update animation method. Usually involves changing the active texture on the sprite.

This can be called manually or by the SpriteList's update_animation method.

Parameters
delta_time – Time since last update.

update_spatial_hash() → None

Update the sprites location in the spatial hash if present.

`alpha`
Get or set the alpha value of the sprite

`bottom`
The lowest y coordinate in the hit box.

When setting this property the sprite is positioned relative to the lowest y coordinate in the hit box.

`center_x`
Get or set the center x position of the sprite.

`center_y`
Get or set the center y position of the sprite.

`color`
Get or set the RGBA multiply color for the sprite.

When setting the color, it may be specified as any of the following:

- an RGBA `tuple` with each channel value between 0 and 255
- an instance of *`Color`*
- an RGB `tuple`, in which case the color will be treated as opaque

Example usage:

```
>>> print(sprite.color)
Color(255, 255, 255, 255)

>>> sprite.color = arcade.color.RED

>>> sprite.color = 255, 0, 0

>>> sprite.color = 255, 0, 0, 128
```

`depth`
Get or set the depth of the sprite.

This is really the z coordinate of the sprite and can be used with OpenGL depth testing with opaque sprites.

`height`
Get or set the height of the sprite in pixels.

`hit_box`

`left`
The leftmost x coordinate in the hit box.

When setting this property the sprite is positioned relative to the leftmost x coordinate in the hit box.

`position`
Get or set the center x and y position of the sprite.

Returns
(center_x, center_y)

`right`
The rightmost x coordinate in the hit box.

When setting this property the sprite is positioned relative to the rightmost x coordinate in the hit box.

scale

Get or set the sprite's x scale value or set both x & y scale to the same value.

Note: Negative values are supported. They will flip & mirror the sprite.

scale_xy

Get or set the x & y scale of the sprite as a pair of values.

sprite_lists: List['SpriteList']

texture

Get or set the visible texture for this sprite This property can be changed over time to animate a sprite.

Note that this doesn't change the hit box of the sprite.

top

The highest y coordinate in the hit box.

When setting this property the sprite is positioned relative to the highest y coordinate in the hit box.

visible

Get or set the visibility of this sprite. This is a shortcut for changing the alpha value of a sprite to 0 or 255:

```
# Make the sprite invisible
sprite.visible = False
# Change back to visible
sprite.visible = True
# Toggle visible
sprite.visible = not sprite.visible
```

width

Get or set width or the sprite in pixels

arcade.**load_animated_gif**(*resource_name*) → *TextureAnimationSprite*

Attempt to load an animated GIF as an `TextureAnimationSprite`.

Many older GIFs will load with incorrect transparency for every frame but the first. Until the Pillow library handles the quirks of the format better, loading animated GIFs will be pretty buggy. A good workaround is loading GIFs in another program and exporting them as PNGs, either as sprite sheets or a frame per file.

33.6 Sprite Lists

arcade.**check_for_collision**(*sprite1: BasicSprite*, *sprite2: BasicSprite*) → bool

Check for a collision between two sprites.

Parameters

- **sprite1** – First sprite
- **sprite2** – Second sprite

Returns

True or False depending if the sprites intersect.

`arcade.`**`check_for_collision_with_list`**(*sprite: SpriteType*, *sprite_list:* SpriteList, *method:* *int = 0*) → List[SpriteType]

Check for a collision between a sprite, and a list of sprites.

Parameters

- **`sprite`** – Sprite to check
- **`sprite_list`** – SpriteList to check against
- **`method`** – Collision check method. 0 is auto-select. (spatial if available, GPU if 1500+ sprites, else simple) 1 is Spatial Hashing if available, 2 is GPU based, 3 is simple check-everything. Defaults to 0.

Returns
List of sprites colliding, or an empty list.

`arcade.`**`check_for_collision_with_lists`**(*sprite:* BasicSprite, *sprite_lists:* *Iterable*[SpriteList*[SpriteType]]*, *method=1*) → List[SpriteType]

Check for a collision between a Sprite, and a list of SpriteLists.

Parameters

- **`sprite`** – Sprite to check
- **`sprite_lists`** – SpriteLists to check against
- **`method`** – Collision check method. 1 is Spatial Hashing if available, 2 is GPU based, 3 is slow CPU-bound check-everything. Defaults to 1.

Returns
List of sprites colliding, or an empty list.

`arcade.`**`get_closest_sprite`**(*sprite: SpriteType*, *sprite_list:* SpriteList) → Tuple[SpriteType, float] | None

Given a Sprite and SpriteList, returns the closest sprite, and its distance.

Parameters

- **`sprite`** – Target sprite
- **`sprite_list`** – List to search for closest sprite.

Returns
A tuple containing the closest sprite and the minimum distance. If the spritelist is empty we return `None`.

`arcade.`**`get_distance_between_sprites`**(*sprite1: SpriteType*, *sprite2: SpriteType*) → float

Returns the distance between the center of two given sprites

Parameters

- **`sprite1`** – Sprite one
- **`sprite2`** – Sprite two

Returns
Distance

`arcade.`**`get_sprites_at_exact_point`**(*point:* *Tuple[float, float]*, *sprite_list:* SpriteList*[SpriteType]*) → List[SpriteType]

Get a list of sprites whose center_x, center_y match the given point. This does NOT return sprites that overlap the point, the center has to be an exact match.

Parameters

- **`point`** – Point to check
- **`sprite_list`** – SpriteList to check against

Returns
List of sprites colliding, or an empty list.

`arcade.`**`get_sprites_at_point`**(*point: Tuple[float, float]*, *sprite_list: SpriteList[SpriteType]*) → List[SpriteType]

Get a list of sprites at a particular point. This function sees if any sprite overlaps the specified point. If a sprite has a different center_x/center_y but touches the point, this will return that sprite.

Parameters

- **`point`** – Point to check
- **`sprite_list`** – SpriteList to check against

Returns
List of sprites colliding, or an empty list.

`arcade.`**`get_sprites_in_rect`**(*rect: Tuple[int, int, int, int] | List[int]*, *sprite_list: SpriteList[SpriteType]*) → List[SpriteType]

Get a list of sprites in a particular rectangle. This function sees if any sprite overlaps the specified rectangle. If a sprite has a different center_x/center_y but touches the rectangle, this will return that sprite.

The rectangle is specified as a tuple of (left, right, bottom, top).

Parameters

- **`rect`** – Rectangle to check
- **`sprite_list`** – SpriteList to check against

Returns
List of sprites colliding, or an empty list.

`class` `arcade.`**`SpatialHash`**(*cell_size: int*)

Bases: `Generic`[`SpriteType`]

Structure for fast collision checking with sprites.

See: https://www.gamedev.net/articles/programming/general-and-gameplay-programming/spatial-hashing-r2697/

Parameters
`cell_size` – Size (width and height) of the cells in the spatial hash

`add`(*sprite: SpriteType*) → None

Add a sprite to the spatial hash.

Parameters
`sprite` – The sprite to add

`get_sprites_near_point`(*point: Tuple[float, float]*) → Set[SpriteType]

Return sprites in the same bucket as the given point.

Parameters
`point` – The point to check

Returns
A set of close-by sprites

`get_sprites_near_rect`(*rect: Tuple[int, int, int, int] | List[int]*) → Set[SpriteType]

Return sprites in the same buckets as the given rectangle.

Parameters
`rect` – The rectangle to check (left, right, bottom, top)

Returns
A set of sprites in the rectangle

`get_sprites_near_sprite`(*sprite: BasicSprite*) → Set[SpriteType]

Get all the sprites that are in the same buckets as the given sprite.

Parameters
`sprite` – The sprite to check

Returns
A set of close-by sprites

`hash`(*point: Tuple[int, int]*) → Tuple[int, int]

Convert world coordinates to cell coordinates

`move`(*sprite: SpriteType*) → None

Shortcut to remove and re-add a sprite.

Parameters
`sprite` – The sprite to move

`remove`(*sprite: SpriteType*) → None

Remove a Sprite.

Parameters
`sprite` – The sprite to remove

`reset`()

Clear all the sprites from the spatial hash.

`count`

Return the number of sprites in the spatial hash

class arcade.`SpriteList`(*use_spatial_hash: bool = False, spatial_hash_cell_size: int = 128, atlas: 'TextureAtlas' | None = None, capacity: int = 100, lazy: bool = False, visible: bool = True*)

Bases: `Generic[SpriteType]`

The purpose of the spriteList is to batch draw a list of sprites. Drawing single sprites will not get you anywhere performance wise as the number of sprites in your project increases. The spritelist contains many low level optimizations taking advantage of your graphics processor. To put things into perspective, a spritelist can contain tens of thousands of sprites without any issues. Sprites outside the viewport/window will not be rendered.

If the spritelist are going to be used for collision it's a good idea to enable spatial hashing. Especially if no sprites are moving. This will make collision checking **a lot** faster. In technical terms collision checking is `O(1)` with spatial hashing enabled and `O(N)` without. However, if you have a list of moving sprites the cost of updating the spatial hash when they are moved can be greater than what you save with spatial collision checks. This needs to be profiled on a case by case basis.

For the advanced options check the advanced section in the arcade documentation.

Parameters

- **use_spatial_hash** – If set to True, this will make creating a sprite, and moving a sprite in the SpriteList slower, but it will speed up collision detection with items in the SpriteList. Great for doing collision detection with static walls/platforms in large maps.
- **spatial_hash_cell_size** – The cell size of the spatial hash (default: 128)
- **atlas** – (Advanced) The texture atlas for this sprite list. If no atlas is supplied the global/default one will be used.
- **capacity** – (Advanced) The initial capacity of the internal buffer. It's a suggestion for the maximum amount of sprites this list can hold. Can normally be left with default value.
- **lazy** – (Advanced) `True` delays creating OpenGL resources for the sprite list until either its `draw()` or `initialize()` method is called. See *Lazy SpriteLists* to learn more.
- **visible** – Setting this to False will cause the SpriteList to not be drawn. When draw is called, the method will just return without drawing.

sprite: *Sprite* **in** self → bool

Return if the sprite list contains the given sprite

iter(self) → Iterator[SpriteType]

Return an iterable object of sprites.

len(self) → int

Return the length of the sprite list.

self **[**index: int**]** = sprite: SpriteType

Replace a sprite at a specific index

append(*sprite: SpriteType*)

Add a new sprite to the list.

Parameters
sprite – Sprite to add to the list.

clear(*deep: bool = True*)

Remove all the sprites resetting the spritelist to it's initial state.

The complexity of this method is `O(N)` with a deep clear (default). If ALL the sprites in the list gets garbage collected with the list itself you can do an `O(1)`` clear using `deep=False`. **Make sure you know exactly what you are doing before using this option.** Any lingering sprite reference will cause a massive memory leak. The `deep` option will iterate all the sprites and remove their references to this spritelist. Sprite and SpriteList have a circular reference for performance reasons.

disable_spatial_hashing() → None

Deletes the internal spatial hash object

draw(**, filter: int | Tuple[int, int] | None = None, pixelated: bool | None = None, blend_function: Tuple[int, int] | Tuple[int, int, int, int] | None = None*) → None

Draw this list of sprites.

Uninitialized sprite lists will first create OpenGL resources before drawing. This may cause a performance stutter when the following are true:

1. You created the sprite list with `lazy=True`
2. You did not call `initialize()` before drawing
3. You are initializing many sprites and/or lists at once

See *Lazy SpriteLists* to learn more.

Parameters

- **filter** – Optional parameter to set OpenGL filter, such as *gl.GL_NEAREST* to avoid smoothing.
- **pixelated** – `True` for pixelated and `False` for smooth interpolation. Shortcut for setting filter=GL_NEAREST.
- **blend_function** – Optional parameter to set the OpenGL blend function used for drawing the sprite list, such as 'arcade.Window.ctx.BLEND_ADDITIVE' or 'arcade.Window.ctx.BLEND_DEFAULT'

draw_hit_boxes(*color: Tuple[int, int, int, int] = (0, 0, 0, 255)*, *line_thickness: float = 1*)

Draw all the hit boxes in this list

enable_spatial_hashing(*spatial_hash_cell_size: int = 128*)

Turn on spatial hashing.

extend(*sprites: Iterable[SpriteType] | SpriteList*)

Extends the current list with the given iterable

Parameters
sprites – Iterable of Sprites to add to the list

index(*sprite: SpriteType*) → int

Return the index of a sprite in the spritelist

Parameters
sprite – Sprite to find and return the index of

initialize() → None

Request immediate creation of OpenGL resources for this list.

Calling this method is optional. It only has an effect for lists created with `lazy=True`. If this method is not called, uninitialized sprite lists will automatically initialize OpenGL resources on their first `draw()` call instead.

This method is useful for performance optimization, advanced techniques, and writing tests. Do not call it across thread boundaries. See *Lazy SpriteLists* to learn more.

insert(*index: int*, *sprite: SpriteType*)

Inserts a sprite at a given index.

Parameters

- **index** – The index at which to insert
- **sprite** – The sprite to insert

move(*change_x: float*, *change_y: float*) → None

Moves all Sprites in the list by the same amount. This can be a very expensive operation depending on the size of the sprite list.

Parameters

- **change_x** – Amount to change all x values by
- **change_y** – Amount to change all y values by

on_update(*delta_time: float = 0.016666666666666666*)

Update the sprite. Similar to update, but also takes a delta-time.

pop(*index: int = -1*) → SpriteType

Pop off the last sprite, or the given index, from the list

Parameters

index – Index of sprite to remove, defaults to -1 for the last item.

preload_textures(*texture_list: List['Texture']*) → None

Preload a set of textures that will be used for sprites in this sprite list.

Parameters

texture_list – List of textures.

remove(*sprite: SpriteType*)

Remove a specific sprite from the list. :param sprite: Item to remove from the list

rescale(*factor: float*) → None

Rescale all sprites in the list relative to the spritelists center.

reverse()

Reverses the current list in-place

shuffle()

Shuffles the current list in-place

sort(**, key: Callable, reverse: bool = False*)

Sort the spritelist in place using < comparison between sprites. This function is similar to python's `list.sort()`.

Example sorting sprites based on y-axis position using a lambda:

```
# Normal order
spritelist.sort(key=lambda x: x.position[1])
# Reversed order
spritelist.sort(key=lambda x: x.position[1], reverse=True)
```

Example sorting sprites using a function:

```
# More complex sorting logic can be applied, but let's just stick to y position
def create_y_pos_comparison(sprite):
    return sprite.position[1]

spritelist.sort(key=create_y_pos_comparison)
```

Parameters

- **key** – A function taking a sprite as an argument returning a comparison key
- **reverse** – If set to `True` the sprites will be sorted in reverse

swap(*index_1: int, index_2: int*)

Swap two sprites by index :param index_1: Item index to swap :param index_2: Item index to swap

update() → None

Call the update() method on each sprite in the list.

update_animation(*delta_time: float = 0.016666666666666666*)

Call the update_animation in every sprite in the sprite list.

write_sprite_buffers_to_gpu() → None

Ensure buffers are resized and fresh sprite data is written into the internal sprite buffers.

This is automatically called in *SpriteList.draw()*, but there are instances when using custom shaders we need to force this to happen since we might have not called *SpriteList.draw()* since the spritelist was modified.

If you have added, removed, moved or changed ANY sprite property this method will synchronize the data on the gpu side (buffer resizing and writing in new data).

alpha

Get or set the alpha/transparency of the entire spritelist. This is a byte value from 0 to 255 were 0 is completely transparent/invisible and 255 is opaque.

alpha_normalized

Get or set the alpha/transparency of all the sprites in the list. This is a floating point number from 0.0 to 1.0 were 0.0 is completely transparent/invisible and 1.0 is opaque.

This is a shortcut for setting the alpha value in the spritelist color.

atlas

Get the texture atlas for this sprite list

buffer_angles

Get the internal OpenGL angle buffer for the spritelist.

This buffer contains a series of 32 bit floats representing the rotation angle for each sprite in degrees.

This buffer is attached to the *geometry* instance with name `in_angle`.

buffer_colors

Get the internal OpenGL color buffer for this spritelist.

This buffer contains a series of 32 bit floats representing the RGBA color for each sprite. 4 x floats = RGBA.

This buffer is attached to the *geometry* instance with name `in_color`.

buffer_indices

Get the internal index buffer for this spritelist.

The data in the other buffers are not in the correct order matching `spritelist[i]`. The index buffer has to be used used to resolve the right order. It simply contains a series of integers referencing locations in the other buffers.

Also note that the length of this buffer might be bigger than the number of sprites. Rely on `len(spritelist)` for the correct length.

This index buffer is attached to the *geometry* instance and will be automatically be applied the the input buffers when rendering or transforming.

buffer_positions

Get the internal OpenGL position buffer for this spritelist.

The buffer contains 32 bit float values with x, y and z positions. These are the center positions for each sprite.

This buffer is attached to the *geometry* instance with name `in_pos`.

buffer_sizes

Get the internal OpenGL size buffer for this spritelist.

The buffer contains 32 bit float width and height values.

This buffer is attached to the *geometry* instance with name `in_size`.

buffer_textures

Get the internal openGL texture id buffer for the spritelist.

This buffer contains a series of single 32 bit floats referencing a texture ID. This ID references a texture in the texture atlas assigned to this spritelist. The ID is used to look up texture coordinates in a 32bit floating point texture the texter atlas provides. This system makes sure we can resize and rebuild a texture atlas without having to rebuild every single spritelist.

This buffer is attached to the *geometry* instance with name `in_texture`.

Note that it should ideally an unsigned integer, but due to compatibility we store them as 32 bit floats. We cast them to integers in the shader.

center

Get the mean center coordinates of all sprites in the list.

color

Get or set the multiply color for all sprites in the list RGBA integers

This will affect all sprites in the list, and each value must be between 0 and 255.

The color may be specified as any of the following:

- an RGBA `tuple` with each channel value between 0 and 255
- an instance of *Color*
- an RGB `tuple`, in which case the color will be treated as opaque

Each individual sprite can also be assigned a color via its *color* property.

When *SpriteList.draw()* is called, each pixel will default to a value equivalent to the following:

1. Convert the sampled texture, sprite, and list colors into normalized floats (0.0 to 1.0)
2. Multiply the color channels together: `texture_color * sprite_color * spritelist_color`
3. Multiply the floating point values by 255 and round the result

color_normalized

Get or set the spritelist color in normalized form (0.0 -> 1.0 floats). This property works the same as *color*.

geometry

Returns the internal OpenGL geometry for this spritelist. This can be used to execute custom shaders with the spritelist data.

One or multiple of the following inputs must be defined in your vertex shader:

```
in vec2 in_pos;
in float in_angle;
in vec2 in_size;
in float in_texture;
in vec4 in_color;
```

visible

Get or set the visible flag for this spritelist. If visible is `False` the `draw()` has no effect.

33.7 Sprite Scenes

`class arcade.Scene`

Bases:

Stores *SpriteList* instances as named layers, allowing bulk updates & drawing.

In addition to helping you update or draw multiple sprite lists at once, this class also provides the following convenience methods:

- *add_sprite()*, which adds sprites to layers by name
- *Scene.from_tilemap()*, which creates a scene from a *TileMap* already loaded from tiled data
- Fine-grained convenience methods for adding, deleting, and reordering sprite lists
- Flexible but slow general convenience methods
- Flexible but slow support for the `in` & `del` Python keywords.

For another example of how to use this class, see *Step 3 - Many Sprites with SpriteList*.

bool(self) → bool

Returns whether or not *_sprite_lists* contains anything

item: str | *SpriteList* **in** self → bool

True when *item* is in *_sprite_lists* or is a value in *_name_mapping*

del self[sprite_list: int | str | *SpriteList*] → None

Remove a sprite list from this scene by its index, name, or instance value.

Tip: Use a more specific method when speed is important!

This method uses `isinstance()`, which will slow down your program if used frequently!

Consider the following alternatives:

- *remove_sprite_list_by_index()*
- *remove_sprite_list_by_name()*
- *remove_sprite_list_by_object()*

Parameters
sprite_list – The index, name, or *SpriteList* instance to remove from this scene.

self[key: str] → *SpriteList*

Retrieve a sprite list by name.

This is here for ease of use to make sub-scripting the scene object directly to retrieve a SpriteList possible.

Parameters
key – The name of the sprite list to retrieve

len(self) → int

Return the number of sprite lists in this scene.

add_sprite(*name: str*, *sprite: Sprite*) → None

Add a Sprite to the SpriteList with the specified name.

If there is no SpriteList for the given `name`, one will be created with *SpriteList*'s default arguments and added to the end (top) of the scene's current draw order.

To fully customize the SpriteList's options, you should create it directly and add it to the scene with one of the following:

- *add_sprite_list_before()*
- *add_sprite_list()*
- *add_sprite_list_after()*

Parameters

- **name** – The name of the sprite list to add to or create.
- **sprite** – The sprite to add.

add_sprite_list(*name: str*, *use_spatial_hash: bool = False*, *sprite_list: SpriteList | None = None*) → None

Add a SpriteList to the scene with the specified name.

This will add a new SpriteList as a layer above the others in the scene.

If no SpriteList is supplied via the `sprite_list` parameter then a new one will be created, and the `use_spatial_hash` parameter will be respected for that creation.

Parameters

- **name** – The name to give the new layer.
- **use_spatial_hash** – If creating a new sprite list, whether to enable spatial hashing on it.
- **sprite_list** – Use a specific sprite list rather than creating a new one.

add_sprite_list_after(*name: str*, *after: str*, *use_spatial_hash: bool = False*, *sprite_list: SpriteList | None = None*) → None

Add a SpriteList to the scene with the specified name after a specific SpriteList.

If no sprite list is supplied via the `sprite_list` parameter, then a new one will be created. Aside from the value of `use_spatial_hash` passed to this method, it will use the default arguments for a new *SpriteList*.

The added sprite list will be drawn above the sprite list named in `after`.

Parameters

- **name** – The name to give the layer.
- **after** – The name of the layer to place the new one after.
- **use_spatial_hash** – If creating a new sprite list, selects whether to enable spatial hashing.
- **sprite_list** – If a sprite list is passed via this argument, it will be used instead of creating a new one.

add_sprite_list_before(*name: str*, *before: str*, *use_spatial_hash: bool = False*, *sprite_list: SpriteList | None = None*) → None

Add a sprite list to the scene with the specified name before another SpriteList.

If no sprite list is supplied via the `sprite_list` parameter, then a new one will be created. Aside from the value of `use_spatial_hash` passed to this method, it will use the default arguments for a new *SpriteList*.

The added sprite list will be drawn under the sprite list named in `before`.

Parameters

- **name** – The name to give the new layer.
- **before** – The name of the layer to place the new one before.
- **use_spatial_hash** – If creating a new sprite list, selects whether to enable spatial hashing.
- **sprite_list** – If a sprite list is passed via this argument, it will be used instead of creating a new one.

draw(*names: Iterable[str] | None = None*, *filter: Tuple[int, int] | None = None*, *pixelated: bool = False*, *blend_function: Tuple[int, int] | Tuple[int, int, int, int] | None = None*, ***kwargs*) → None

Call *draw()* on the scene's sprite lists.

By default, this method calls *draw()* on each sprite list in the scene in the default draw order.

You can limit and reorder the draw calls with the `names` argument by passing a list of names in the scene. The sprite lists will be drawn in the order of the passed iterable. If a name is not in the scene, a `KeyError` will be raised.

The other named keyword arguments are the same as those of *SpriteList.draw()*. The `**kwargs` option is for advanced users who have subclassed *SpriteList*.

Parameters

- **names** – Which layers to draw & what order to draw them in.
- **filter** – Optional parameter to set OpenGL filter, such as `gl.GL_NEAREST` to avoid smoothing.
- **pixelated** – `True` for pixel art and `False` for smooth scaling.
- **blend_function** – Use the specified OpenGL blend function while drawing the sprite list, such as `arcade.Window.ctx.BLEND_ADDITIVE` or `arcade.Window.ctx.BLEND_DEFAULT`.

draw_hit_boxes(*color: Tuple[int, int, int, int] = (0, 0, 0, 255)*, *line_thickness: float = 1.0*, *names: Iterable[str] | None = None*) → None

Draw debug hit box outlines for sprites in the scene's layers.

If `names` is a valid iterable of layer names in the scene, then hit boxes will be drawn for the specified layers in the order of the passed iterable.

If *names* is not provided, then every layer's hit boxes will be drawn in the order specified.

Parameters

- **color** – The RGBA color to use to draw the hit boxes with.
- **line_thickness** – How many pixels thick the hit box outlines should be
- **names** – Which layers & what order to draw their hit boxes in.

classmethod from_tilemap(*tilemap: TileMap*) → *Scene*

Create a new Scene from a *TileMap* object.

The SpriteLists will use the layer names and ordering as defined in the Tiled file.

Parameters

tilemap – The *TileMap* object to create the scene from.

get_sprite_list(*name: str*) → *SpriteList*

Retrieve a sprite list by name.

It is also possible to access sprite lists the following ways:

- `scene_instance[name]`
- directly accessing `scene_instance._name_mapping`, although this will get flagged by linters as bad style.

Parameters

name – The name of the sprite list to retrieve.

move_sprite_list_after(*name: str, after: str*) → None

Move a named SpriteList in the scene to be after another SpriteList in the scene.

A *SceneKeyError* will be raised if either `name` or `after` contain a name not currently in the scene. This exception can be handled as a `KeyError`.

Parameters

- **name** – The name of the SpriteList to move.
- **after** – The name of the SpriteList to place it after.

move_sprite_list_before(*name: str, before: str*) → None

Move a named SpriteList in the scene to be before another SpriteList in the scene.

A *SceneKeyError* will be raised if either `name` or `before` contain a name not currently in the scene. This exception can be handled as a `KeyError`.

Parameters

- **name** – The name of the SpriteList to move.
- **before** – The name of the SpriteList to place it before.

on_update(*delta_time: float = 0.016666666666666666, names: Iterable[str] | None = None*) → None

Call *on_update()* on the scene's sprite lists.

By default, this method calls *on_update()* on the scene's sprite lists in the default draw order.

You can limit and reorder the updates with the `names` argument by passing a list of names in the scene. The sprite lists will be drawn in the order of the passed iterable. If a name is not in the scene, a `KeyError` will be raised.

Parameters

- **delta_time** – The time step to update by in seconds.
- **names** – Which layers & what order to update them in.

remove_sprite_list_by_index(*index: int*) → None

Remove a layer from the scene by its index in the draw order.

Parameters
index – The index of the sprite list to remove.

remove_sprite_list_by_name(*name: str*) → None

Remove a layer from the scene by its name.

A KeyError will be raised if the SpriteList is not in the scene.

Parameters
name – The name of the sprite list to remove.

remove_sprite_list_by_object(*sprite_list:* SpriteList) → None

Remove the passed SpriteList instance from the Scene.

A ValueError will be raised if the passed sprite list is not in the scene.

Parameters
sprite_list – The sprite list to remove.

update(*names: Iterable[str] | None = None*) → None

Call *update()* on the scene's sprite lists.

By default, this method calls *update()* on the scene's sprite lists in the default draw order.

You can limit and reorder the updates with the `names` argument by passing a list of names in the scene. The sprite lists will be drawn in the order of the passed iterable. If a name is not in the scene, a KeyError will be raised.

Parameters
names – Which layers & what order to update them in.

update_animation(*delta_time: float, names: Iterable[str] | None = None*) → None

Call *update_animation()* on the scene's sprite lists.

By default, this method calls *update_animation()* on each sprite list in the scene in the default draw order.

You can limit and reorder the updates with the `names` argument by passing a list of names in the scene. The sprite lists will be drawn in the order of the passed iterable. If a name is not in the scene, a KeyError will be raised.

Parameters

- **delta_time** – The time step to update by in seconds.
- **names** – Which layers & what order to update them in.

class arcade.**SceneKeyError**(*name: str*)

Bases: KeyError

Raised when a py:class:*.Scene* cannot find a layer for a specified name.

It is a subclass of KeyError, and you can handle it as one if you wish:

```python
try:
    # this will raise a SceneKeyError
    scene_instance.add_sprite("missing_layer_name", arcade.SpriteSolidColor(10,10))

# We can handle it as a KeyError because it is a subclass of it
except KeyError as e:
    print("Your error handling should go here")
```

The main purpose of this class is to help arcade's developers keep error messages consistent.

Parameters
: **name** – the name of the missing *SpriteList*

33.8 Camera

class arcade.**Camera**(**, viewport: Tuple[int, int, int, int] | None = None, projection: Tuple[float, float, float, float] | None = None, zoom: float = 1.0, rotation: float = 0.0, anchor: Tuple[float, float] | None = None, window: Window | None = None*)

Bases: *SimpleCamera*

The Camera class is used for controlling the visible viewport, the projection, zoom and rotation. It is very useful for separating a scrolling screen of sprites, and a GUI overlay. For an example of this in action, see sprite_move_scrolling.

Parameters

- **viewport** – (left, bottom, width, height) size of the viewport. If None the window size will be used.
- **projection** – (left, right, bottom, top) size of the projection. If None the window size will be used.
- **zoom** – the zoom to apply to the projection
- **rotation** – the angle in degrees to rotate the projection
- **anchor** – the x, y point where the camera rotation will anchor. Default is the center of the viewport.
- **window** – Window to associate with this camera, if working with a multi-window program.

get_sprites_at_point(*point: Point, sprite_list: SpriteList*) → List['Sprite']

Get a list of sprites at a particular point when This function sees if any sprite overlaps the specified point. If a sprite has a different center_x/center_y but touches the point, this will return that sprite.

Parameters

- **point** – Point to check
- **sprite_list** – SpriteList to check against

Returns
: List of sprites colliding, or an empty list.

set_viewport(*viewport: Tuple[int, int, int, int]*) → None

Sets the viewport

shake(*velocity: Vec2 | tuple, speed: float = 1.5, damping: float = 0.9*) → None

Add a camera shake.

Parameters

- **velocity** – Vector to start moving the camera
- **speed** – How fast to shake
- **damping** – How fast to stop shaking

update() → None

Update the camera's viewport to the current settings.

use() → None

Select this camera for use. Do this right before you draw.

anchor

Get or set the rotation anchor for the camera.

By default, the anchor is the center of the screen and the anchor value is *None*. Assigning a custom anchor point will override this behavior. The anchor point is in world / global coordinates.

Example:

```
# Set the anchor to the center of the world
camera.anchor = 0, 0
# Set the anchor to the center of the player
camera.anchor = player.position
```

far

The far applied to the projection

near

The near applied to the projection

rotation

Get or set the rotation in degrees.

This will rotate the camera clockwise meaning the contents will rotate counter-clockwise.

scale

Returns the x, y scale.

zoom

The zoom applied to the projection. Just returns the x scale value.

class arcade.**SimpleCamera**(*, *viewport: Tuple[int, int, int, int] | None = None*, *projection: Tuple[float, float, float, float] | None = None*, *window: Window | None = None*)

Bases:

A simple camera that allows to change the viewport, the projection and can move around. That's it. See arcade.Camera for more advance stuff.

Parameters

- **viewport** – Size of the viewport: (left, bottom, width, height)
- **projection** – Space to allocate in the viewport of the camera (left, right, bottom, top)

center(*vector: Vec2 | tuple*, *speed: float = 1.0*) → None

Centers the camera on coordinates

get_map_coordinates(*camera_vector: Vec2 | tuple*) → Vec2

Returns map coordinates in pixels from screen coordinates based on the camera position

Parameters

camera_vector – Vector captured from the camera viewport `move`(*vector: Vec2 | tuple*) → None

Moves the camera with a speed of 1.0, aka instant move

This is equivalent to calling move_to(my_pos, 1.0)

`move_to`(*vector: Vec2 | tuple*, *speed: float = 1.0*) → None

Sets the goal position of the camera.

The camera will lerp towards this position based on the provided speed, updating its position every time the use() function is called.

Parameters

- **`vector`** – Vector to move the camera towards.
- **`speed`** – How fast to move the camera, 1.0 is instant, 0.1 moves slowly

`resize`(*viewport_width: int*, *viewport_height: int*, ***, *resize_projection: bool = True*) → None

Resize the camera's viewport. Call this when the window resizes.

Parameters

- **`viewport_width`** – Width of the viewport
- **`viewport_height`** – Height of the viewport
- **`resize_projection`** – if True the projection will also be resized

`set_viewport`(*viewport: Tuple[int, int, int, int]*) → None

Sets the viewport

`update`()

Update the camera's viewport to the current settings.

`use`() → None

Select this camera for use. Do this right before you draw.

`projection`

The dimensions of the space to project in the camera viewport (left, right, bottom, top). The projection is what you want to project into the camera viewport.

`projection_to_viewport_height_ratio`

The ratio of projection height to viewport height

`projection_to_viewport_width_ratio`

The ratio of projection width to viewport width

`viewport`

The space the camera will hold on the screen (left, bottom, width, height)

`viewport_height`

Returns the height of the viewport

`viewport_to_projection_height_ratio`

The ratio of viewport height to projection height

`viewport_to_projection_width_ratio`

The ratio of viewport width to projection width

`viewport_width`

Returns the width of the viewport

33.9 Text

class arcade.**Text**(*text: str, start_x: int, start_y: int, color: Tuple[int, int, int] | Tuple[int, int, int, int] = (255, 255, 255, 255), font_size: float = 12, width: int | None = 0, align: str = 'left', font_name: str | Tuple[str, ...] = ('calibri', 'arial'), bold: bool = False, italic: bool = False, anchor_x: str = 'left', anchor_y: str = 'baseline', multiline: bool = False, rotation: float = 0, batch: Batch | None = None, group: Group | None = None, start_z: int = 0*)

Bases:

An object-oriented way to draw text to the screen.

Tip: Use this class when performance matters!

Unlike `draw_text()`, this class does not risk wasting time recalculating and re-setting any text each time `draw()` is called. This makes it faster while:

- requiring you to manage instances and drawing yourself
- using negligible extra RAM

The speed advantage scales as more text needs to be drawn to the screen.

The constructor arguments work identically to those of `draw_text()`. See its documentation for in-depth explanation for how to use each of them. For example code, see drawing_text_objects.

Parameters

- **text** – Initial text to display. Can be an empty string
- **start_x** – x position to align the text's anchor point with
- **start_y** – y position to align the text's anchor point with
- **start_z** – z position to align the text's anchor point with
- **color** – Color of the text as an RGBA tuple or a `Color` instance.
- **font_size** – Size of the text in points
- **width** – A width limit in pixels
- **align** – Horizontal alignment; values other than "left" require width to be set
- **font_name** (`Union[str, Tuple[str, ...]]`) – A font name, path to a font file, or list of names
- **bold** – Whether to draw the text as bold
- **italic** – Whether to draw the text as italic
- **anchor_x** – How to calculate the anchor point's x coordinate. Options: "left", "center", or "right"
- **anchor_y** – How to calculate the anchor point's y coordinate. Options: "top", "bottom", "center", or "baseline".
- **multiline** – Requires width to be set; enables word wrap rather than clipping
- **rotation** – rotation in degrees, counter-clockwise from horizontal

All constructor arguments other than `text` have a corresponding property. To access the current text, use the `value` property instead.

By default, the text is placed so that:

- the left edge of its bounding box is at `start_x`
- its baseline is at `start_y`

The baseline is located along the line the bottom of the text would be written on, excluding letters with tails such as y:

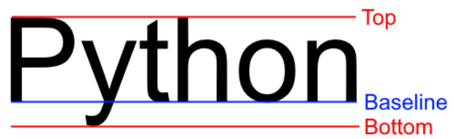


Fig. 1: The blue line is the baseline for the string `"Python"`

`rotation` allows for the text to be rotated around the anchor point by the passed number of degrees. Positive values rotate counter-clockwise from horizontal, while negative values rotate clockwise:

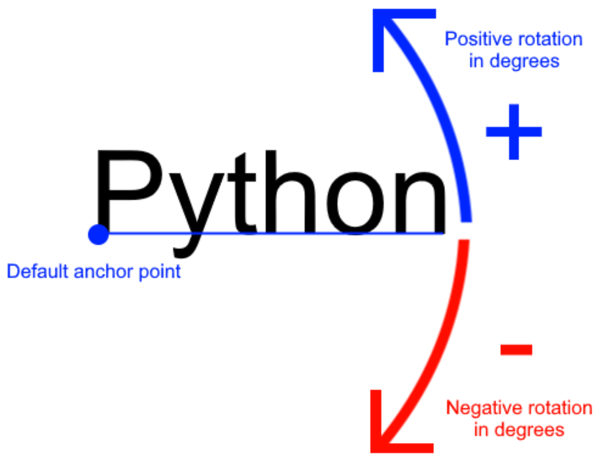


Fig. 2: Rotation around the default anchor (`anchor_y="baseline"` and `anchor_x="left"`)

`draw()` → None

Draw the label to the screen at its current `x` and `y` position.

`draw_debug`(*anchor_color: Tuple[int, int, int, int] = (255, 0, 0, 255), background_color: Tuple[int, int, int, int] = (0, 0, 139, 255), outline_color: Tuple[int, int, int, int] = (255, 255, 255, 255)*) → None

Draw test with debug geometry showing the content area, outline and the anchor point.

Parameters

- **`anchor_color`** – Color of the anchor point
- **`background_color`** – Color the content background
- **`outline_color`** – Color of the content outline

`align`

`anchor_x`

Get or set the horizontal anchor.

Options: “left”, “center”, or “right”

`anchor_y`

Get or set the vertical anchor.

Options : “top”, “bottom”, “center”, or “baseline”

`batch`

`bold`

Get or set bold state of the label

`bottom`

Pixel location of the bottom content border.

`color`

Get or set the text color for the label

`content_height`

Get the pixel height of the text content.

`content_size`

Get the pixel width and height of the text contents.

`content_width`

Get the pixel width of the text contents

`font_name`

Get or set the font name(s) for the label

`font_size`

Get or set the font size of the label

`group`

`height`

Get or set the height of the label in pixels This value affects text flow when multiline text is used. If you are looking for the physical size if the text, see *`content_height`*

`italic`

Get or set the italic state of the label

`left`

Pixel location of the left content border.

`multiline`

Get or set the multiline flag of the label.

`position`

The current x, y position as a tuple.

This is faster than setting x and y position separately because the underlying geometry only needs to change position once.

right
Pixel location of the right content border.

rotation

size
Get the size of the label

start_z
Get or set the z position of the label

text
Get or set the current text string to display.

The value assigned will be converted to a string.

This is an alias for `value`

top
Pixel location of the top content border.

value
Get or set the current text string to display.

The value assigned will be converted to a string.

width
Get or set the width of the label in pixels. This value affects text flow when multiline text is used. If you are looking for the physical size if the text, see `content_width`

x
Get or set the x position of the label

y
Get or set the y position of the label

arcade.**create_text_sprite**(*text: str, color: Tuple[int, int, int, int] = (255, 255, 255, 255), font_size: float = 12, width: int = 0, align: str = 'left', font_name: str | Tuple[str, ...] = ('calibri', 'arial'), bold: bool = False, italic: bool = False, anchor_x: str = 'left', multiline: bool = False, texture_atlas: TextureAtlas | None = None, background_color: Tuple[int, int, int, int] | None = None*) → *Sprite*

Creates a sprite containing text based off of `Text`.

Internally this creates a Text object and an empty texture. It then uses either the provided texture atlas, or gets the default one, and draws the Text object into the texture atlas.

It then creates a sprite referencing the newly created texture, and positions it accordingly, and that is final result that is returned from the function.

If you are providing a custom texture atlas, something important to keep in mind is that the resulting Sprite can only be added to SpriteLists which use that atlas. If it is added to a SpriteList which uses a different atlas, you will likely just see a black box drawn in its place.

Parameters

- **text** – Initial text to display. Can be an empty string
- **color** – Color of the text as a tuple or list of 3 (RGB) or 4 (RGBA) integers
- **font_size** – Size of the text in points
- **width** – A width limit in pixels

- **align** – Horizontal alignment; values other than "left" require width to be set
- **font_name** – A font name, path to a font file, or list of names
- **bold** – Whether to draw the text as bold
- **italic** – Whether to draw the text as italic
- **anchor_x** – How to calculate the anchor point's x coordinate. Options: "left", "center", or "right"
- **multiline** – Requires width to be set; enables word wrap rather than clipping
- **texture_atlas** – The texture atlas to use for the newly created texture. The default global atlas will be used if this is None.
- **background_color** – The background color of the text. If None, the background will be transparent.

`arcade.draw_text`(*text: Any, start_x: int, start_y: int, color: Tuple[int, int, int, int] = (255, 255, 255, 255), font_size: float = 12, width: int = 0, align: str = 'left', font_name: str | Tuple[str, ...] = ('calibri', 'arial'), bold: bool = False, italic: bool = False, anchor_x: str = 'left', anchor_y: str = 'baseline', multiline: bool = False, rotation: float = 0, start_z: int = 0*)

A simple way for beginners to draw text.

> **Warning:** Use `arcade.Text` objects instead.
>
> This method of drawing text is very slow and might be removed in the near future. Text objects can be 10-100 times faster depending on the use case.

> **Warning:** Cameras affect text drawing!
>
> If you want to draw a custom GUI that doesn't move with the game world, you will need a second camera. For information on how to do this, see sprite_move_scrolling.

This function lets you start draw text easily with better performance than the old pillow-based text. If you need even higher performance, consider using `Text`.

Example code can be found at drawing_text.

Parameters

- **text** – Text to display. The object passed in will be converted to a string
- **start_x** – x position to align the text's anchor point with
- **start_y** – y position to align the text's anchor point with
- **start_z** – z position to align the text's anchor point with
- **color** – Color of the text as an RGBA tuple or `Color` instance.
- **font_size** – Size of the text in points
- **width** – A width limit in pixels
- **align** – Horizontal alignment; values other than "left" require width to be set
- **font_name** (`Union[str, Tuple[str, ...]]`) – A font name, path to a font file, or list of names

- **bold** – Whether to draw the text as bold
- **italic** – Whether to draw the text as italic
- **anchor_x** – How to calculate the anchor point's x coordinate
- **anchor_y** – How to calculate the anchor point's y coordinate
- **multiline** – Requires width to be set; enables word wrap rather than clipping
- **rotation** – rotation in degrees, counter-clockwise from horizontal

By default, the text is placed so that:

- the left edge of its bounding box is at start_x
- its baseline is at start_y

The baseline of text is the line it would be written on:

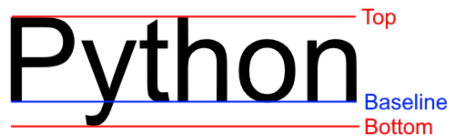


Fig. 3: The blue line is the baseline for the string "Python"

font_name can be any of the following:

- a built-in font in the *Built-In Resources*
- the name of a system font
- a path to a font on the system
- a *tuple* containing any mix of the previous three

Each entry provided will be tried in order until one is found. If none of the fonts are found, a default font will be chosen (usually Arial).

anchor_x and anchor_y specify how to calculate the anchor point, which affects how the text is:

- Placed relative to start_x and start_y
- Rotated

By default, the text is drawn so that start_x is at the left of the text's bounding box and start_y is at the baseline.

You can set a custom anchor point by passing combinations of the following values for anchor_x and anchor_y:

Table 1: Values allowed by anchor_x

String value	Practical Effect	Anchor Position
"left" *(default)*	Text drawn with its left side at start_x	Anchor point on the left side of the text's bounding box
"center"	Text drawn horizontally centered on start_x	Anchor point at horizontal center of text's bounding box
"right"	Text drawn with its right side at start_x	Anchor placed on the right side of the text's bounding box

Table 2: Values allowed by `anchor_y`

String value	Practical Effect	Anchor Position
`"baseline"` *(default)*	Text drawn with baseline on `start_y`.	Anchor placed at the text rendering baseline
`"top"`	Text drawn with its top aligned with `start_y`	Anchor point placed at the top of the text
`"bottom"`	Text drawn with its absolute bottom aligned with `start_y`, including the space for tails on letters such as y and g	Anchor point placed at the bottom of the text after the space allotted for letters such as y and g
`"center"`	Text drawn with its vertical center on `start_y`	Anchor placed at the vertical center of the text

`rotation` allows for the text to be rotated around the anchor point by the passed number of degrees. Positive values rotate counter-clockwise from horizontal, while negative values rotate clockwise:

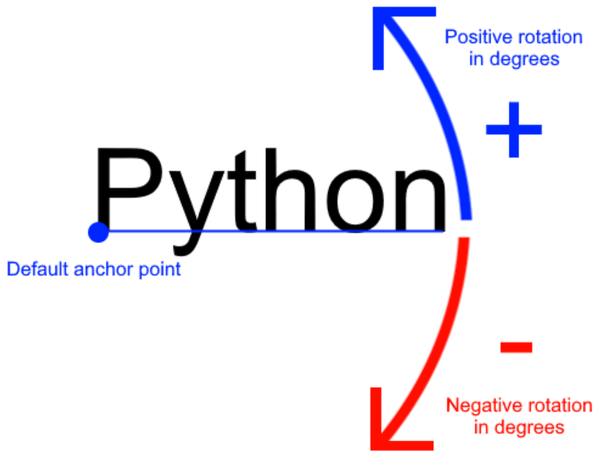


Fig. 4: Rotation around the default anchor point (`anchor_y="baseline"` and `anchor_x="left"`)

It can be helpful to think of this function working as follows:

1. Text layout and alignment are calculated:
   1. The text's characters are laid out within a bounding box according to the current styling options
   2. The anchor point on the text is calculated based on the text value, styling, as well as values for `anchor_x` and `anchor_y`
2. The text is placed so its anchor point is at `(start_x, start_y))`
3. The text is rotated around its anchor point before finally being drawn

This function is less efficient than using `Text` because some steps above can be repeated each time a call is made rather than fully cached as with the class.

`arcade.`**`load_font`**`(`*path: str* | *Path*`)` → None

Load fonts in a file (usually .ttf) adding them to a global font registry.

A file can contain one or multiple fonts. Each font has a name. Open the font file to find the actual name(s). These names are used to select font when drawing text.

Examples:

```
# Load a font in the current working directory
# (absolute path is often better)
arcade.load_font("Custom.ttf")
# Load a font using a custom resource handle
arcade.load_font(":font:Custom.ttf")
```

Parameters
`path` – A string, or an array of paths with fonts.

Raises
`FileNotFoundError` – if the font specified wasn't found

Returns

33.10 Tiled Map Reader

class `arcade.tilemap.TileMap`(*map_file: str | Path = '', scaling: float = 1.0, layer_options: Dict[str, Dict[str, Any]] | None = None, use_spatial_hash: bool = False, hit_box_algorithm: HitBoxAlgorithm | None = None, tiled_map: pytiled_parser.TiledMap | None = None, offset: Vec2 = Vec2(0, 0), texture_atlas: 'TextureAtlas' | None = None, lazy: bool = False*)

Bases:

Class that represents a fully parsed and loaded map from Tiled. For examples on how to use this class, see: https://api.arcade.academy/en/latest/examples/platform_tutorial/step_09.html

Parameters

- **`map_file`** (*`Union[str, Path]`*) – A JSON map file for a Tiled map to initialize from
- **`scaling`** – Global scaling to apply to all Sprites.
- **`layer_options`** (*`Dict[str, Dict[str, Any]]`*) – Extra parameters for each layer.
- **`use_spatial_hash`** – If set to True, this will make moving a sprite in the SpriteList slower, but it will speed up collision detection with items in the SpriteList. Great for doing collision detection with static walls/platforms.
- **`hit_box_algorithm`** – The hit box algorithm to use for the Sprite's in this layer.
- **`tiled_map`** – An already parsed pytiled-parser map object. Passing this means that the `map_file` argument will be ignored, and the pre-parsed map will instead be used. This can be helpful for working with Tiled World files.
- **`offset`** – Can be used to offset the position of all sprites and objects within the map. This will be applied in addition to any offsets from Tiled. This value can be overridden with the layer_options dict.
- **`texture_atlas`** – A default texture atlas to use for the SpriteLists created by this map. If not supplied the global default atlas will be used.
- **`lazy`** – SpriteLists will be created lazily.

The *layer_options* parameter can be used to specify per layer arguments.

The available options for this are:

use_spatial_hash - A boolean to enable spatial hashing on this layer's SpriteList. scaling - A float providing layer specific Sprite scaling. hit_box_algorithm - The hit box algorithm to use for the Sprite's in this layer. offset - A tuple containing X and Y position offsets for the layer custom_class - All objects in the layer are created from this class instead of Sprite. Must be subclass of Sprite. custom_class_args - Custom arguments, passed into the constructor of the custom_class texture_atlas - A texture atlas to use for the SpriteList from this layer, if none is supplied then the one defined at the map level will be used.

For example:

code-block:

```
layer_options = {
    "Platforms": {
        "use_spatial_hash": True,
        "scaling": 2.5,
        "offset": (-128, 64),
        "custom_class": Platform,
        "custom_class_args": {
            "health": 100
        }
    },
}
```

The keys and their values in each layer are passed to the layer processing functions using the ** operator on the dictionary.

get_cartesian(*x: float*, *y: float*) → Tuple[float, float]

Given a set of coordinates in pixel units, this returns the cartesian coordinates.

This assumes the supplied coordinates are pixel coordinates, and bases the cartesian grid off of the Map's tile size.

If you have a map with 128x128 pixel Tiles, and you supply coordinates 500, 250 to this function you'll receive back 3, 2

Parameters

- **x** – The X Coordinate to convert
- **y** – The Y Coordinate to convert

get_tilemap_layer(*layer_path: str*) → Layer | None

background_color: Color | None

The background color of the map.

height: float

The height of the map in tiles. This is the number of tiles, not pixels.

object_lists: Dict[str, List[*TiledObject*]]

A dictionary mapping TiledObjects to their layer names. This is used for all object layers of the map.

offset: Vec2

A tuple containing the X and Y position offset values.

scaling: float

A global scaling value to be applied to all Sprites in the map.

sprite_lists: Dict[str, *SpriteList*]

A dictionary mapping SpriteLists to their layer names. This is used for all tile layers of the map.

tile_height: float

The height in pixels of each tile.

tile_width: float

The width in pixels of each tile.

tiled_map: TiledMap

The pytiled-parser map object. This can be useful for implementing features that aren't supported by this class by accessing the raw map data directly.

width: float

The width of the map in tiles. This is the number of tiles, not pixels.

`arcade.tilemap.`**`load_tilemap`**(*map_file: str | Path*, *scaling: float = 1.0*, *layer_options: Dict[str, Dict[str, Any]] | None = None*, *use_spatial_hash: bool = False*, *hit_box_algorithm: HitBoxAlgorithm | None = None*, *offset: Vec2 = Vec2(0, 0)*, *texture_atlas: 'TextureAtlas' | None = None*, *lazy: bool = False*) → *TileMap*

Given a .json map file, loads in and returns a *TileMap* object.

A TileMap can be created directly using the classes *__init__* function. This function exists for ease of use.

For more clarification on the layer_options key, see the *__init__* function of the *TileMap* class

Parameters

- **map_file** (*Union[str, Path]*) – The JSON map file.
- **scaling** – The global scaling to apply to all Sprite's within the map.
- **use_spatial_hash** – If set to True, this will make moving a sprite in the SpriteList slower, but it will speed up collision detection with items in the SpriteList. Great for doing collision detection with static walls/platforms.
- **hit_box_algorithm** – The hit box algorithm to use for collision detection.
- **layer_options** (*Dict[str, Dict[str, Any]]*) – Layer specific options for the map.
- **offset** – Can be used to offset the position of all sprites and objects within the map. This will be applied in addition to any offsets from Tiled. This value can be overridden with the layer_options dict.
- **lazy** – SpriteLists will be created lazily.

`arcade.tilemap.`**`read_tmx`**(*map_file: str | Path*) → TiledMap

Deprecated function to raise a warning that it has been removed.

Exists to provide info for outdated code bases.

33.11 Texture Management

`arcade.`**`make_circle_texture`**(*diameter: int, color: Tuple[int, int, int, int], name: str | None = None, hitbox_algorithm: HitBoxAlgorithm | None = None*) → *Texture*

Return a Texture of a circle with the given diameter and color.

Parameters

- **`diameter`** – Diameter of the circle and dimensions of the square `Texture` returned.
- **`color`** – Color of the circle as a `Color` instance a 3 or 4 tuple.
- **`name`** – Custom or pre-chosen name for this texture

Returns

New `Texture` object.

`arcade.`**`make_soft_circle_texture`**(*diameter: int, color: Tuple[int, int, int, int], center_alpha: int = 255, outer_alpha: int = 0, name: str | None = None, hit_box_algorithm: HitBoxAlgorithm | None = None*) → *Texture*

Return a `Texture` of a circle with the given diameter and color, fading out at its edges.

Parameters

- **`diameter`** – Diameter of the circle and dimensions of the square `Texture` returned.
- **`color`** – Color of the circle as a 4-length tuple or `Color` instance.
- **`center_alpha`** – Alpha value of the circle at its center.
- **`outer_alpha`** – Alpha value of the circle at its edges.
- **`name`** – Custom or pre-chosen name for this texture
- **`hit_box_algorithm`** – The hit box algorithm

Returns

New `Texture` object.

`arcade.`**`make_soft_square_texture`**(*size: int, color: Tuple[int, int, int, int], center_alpha: int = 255, outer_alpha: int = 0, name: str | None = None*) → *Texture*

Return a `Texture` of a square with the given diameter and color, fading out at its edges.

Parameters

- **`size`** – Diameter of the square and dimensions of the square Texture returned.
- **`color`** – Color of the square.
- **`center_alpha`** – Alpha value of the square at its center.
- **`outer_alpha`** – Alpha value of the square at its edges.
- **`name`** – Custom or pre-chosen name for this texture

Returns

New `Texture` object.

`arcade.`**`cleanup_texture_cache`**()

This cleans up the cache of textures. Useful when running unit tests so that the next test starts clean.

`arcade.`**`get_default_image`**(*size: Tuple[int, int] = (128, 128)*) → ImageData

Generates and returns a default image and caches it internally for future use.

Parameters
`size` – Size of the image to create.

Returns
The default image.

`arcade.`**`get_default_texture`**(*size: Tuple[int, int] = (128, 128)*) → *Texture*

Creates and returns a default texture and caches it internally for future use.

Parameters
`size` – Size of the texture to create

Returns
The default texture.

`class arcade.`**`TextureManager`**

Bases:

This class is used to manage textures. It is used to keep track of textures that have been loaded, and to make sure we don't load the same texture twice.

Textures loaded through this manager is cached internally.

`flush`(*sprite_sheets: bool = True*, *textures: bool = True*, *image_data: bool = True*, *hit_boxes: bool = False*)

Remove contents from the texture manager.

Parameters

- **`sprite_sheets`** – If `True`, sprite sheets will be flushed.
- **`textures`** – If `True`, textures will be flushed.
- **`image_data`** – If `True`, image data will be flushed.
- **`hit_boxes`** – If `True`, hit boxes will be flushed.

`spritesheet`(*path: str | Path*, *cache: bool = True*) → *SpriteSheet*

Loads a spritesheet or returns a cached version.

Parameters

- **`path`** – Path to the file to load.
- **`cache`** – If `True`, the spritesheet will be cached. If `False`, the spite sheet will not be cached or returned from the cache.

`texture`(*path: str | Path*, *hit_box_algorithm: HitBoxAlgorithm | None = None*, *cache: bool = True*) → *Texture*

Loads a texture or returns a cached version.

Parameters

- **`path`** – Path to the file to load.
- **`hit_box_algorithm`** – Algorithm to use to create a hit box for the texture.

`arcade.`**`load_spritesheet`**(*file_name: str | Path*, *sprite_width: int*, *sprite_height: int*, *columns: int*, *count: int*, *margin: int = 0*, *hit_box_algorithm: HitBoxAlgorithm | None = None*) → List[*Texture*]

Parameters

- **file_name** – Name of the file to that holds the texture.
- **sprite_width** – Width of the sprites in pixels
- **sprite_height** – Height of the sprites in pixels
- **columns** – Number of tiles wide the image is.
- **count** – Number of tiles in the image.
- **margin** – Margin between images
- **hit_box_algorithm** – The hit box algorithm

Returns List
List of `Texture` objects.

`arcade.load_texture`(*file_path: str | Path*, ***, *x: int = 0*, *y: int = 0*, *width: int = 0*, *height: int = 0*, *hit_box_algorithm: HitBoxAlgorithm | None = None*) → *Texture*

Load an image from disk and create a texture.

The `x`, `y`, `width`, and `height` parameters are used to specify a sub-rectangle of the image to load. If not specified, the entire image is loaded.

Parameters

- **file_name** – Name of the file to that holds the texture.
- **x** – X coordinate of the texture in the image.
- **y** – Y coordinate of the texture in the image.
- **width** – Width of the texture in the image.
- **height** – Height of the texture in the image.
- **hit_box_algorithm** –

Returns
New `Texture` object.

Raises
ValueError

`arcade.load_texture_pair`(*file_name: str | Path*, *hit_box_algorithm: HitBoxAlgorithm | None = None*) → Tuple[*Texture*, *Texture*]

Load a texture pair, with the second being a mirror image of the first. Useful when doing animations and the character can face left/right.

Parameters

- **file_name** – Path to texture
- **hit_box_algorithm** – The hit box algorithm

`arcade.load_textures`(*file_name: str | Path*, *image_location_list: Tuple[Tuple[int, int, int, int] | List[int], ...] | List[Tuple[int, int, int, int] | List[int]]*, *mirrored: bool = False*, *flipped: bool = False*, *hit_box_algorithm: HitBoxAlgorithm | None = None*) → List[*Texture*]

Load a set of textures from a single image file.

Note: If the code is to load only part of the image, the given *x*, *y* coordinates will start with the origin *(0, 0)* in the upper left of the image. When drawing, Arcade uses *(0, 0)* in the lower left corner. Be careful with this reversal.

For a longer explanation of why computers sometimes start in the upper left, see: http://programarcadegames.com/index.php?chapter=introduction_to_graphics&lang=en#section_5

Parameters

- **file_name** – Name of the file.
- **image_location_list** – List of image sub-locations. Each rectangle should be a *List* of four floats: *[x, y, width, height]*.
- **mirrored** – If set to *True*, the image is mirrored left to right.
- **flipped** – If set to *True*, the image is flipped upside down.
- **hit_box_algorithm** – One of None, 'None', 'Simple' (default) or 'Detailed'.
- **hit_box_detail** – Float, defaults to 4.5. Used with 'Detailed' to hit box

Returns
List of `Texture`'s.

Raises
ValueError

`class arcade.Texture`(*image: Image | ImageData*, ***, *hit_box_algorithm: HitBoxAlgorithm | None = None*, *hit_box_points: Sequence[Tuple[float, float]] | None = None*, *hash: str | None = None*, ***kwargs*)

Bases:

An arcade.Texture is simply a wrapper for image data as a Pillow image and the hit box data for this image used in collision detection. Usually created by the `load_texture` or `load_textures` commands.

Parameters

- **image** – The image or ImageData for this texture
- **hit_box_algorithm** – The algorithm to use for calculating the hit box.
- **hit_box_points** – A list of hitbox points for the texture to use (Optional). Completely overrides the hit box algorithm.
- **hash** – Optional unique name for the texture. Can be used to make this texture globally unique. By default the hash of the pixel data is used.

`add_atlas_ref`(*atlas:* TextureAtlas) → None

Add a reference to an atlas that this texture is in.

`classmethod create_atlas_name`(*hash: str*, *vertex_order: Tuple[int, int, int, int] = (0, 1, 2, 3)*)

`classmethod create_cache_name`(***, *hash: str*, *hit_box_algorithm: HitBoxAlgorithm*, *vertex_order: Tuple[int, int, int, int] = (0, 1, 2, 3)*) → str

Create a cache name for the texture.

Parameters

- **image_data** – The image data
- **hit_box_algorithm** – The hit box algorithm
- **hit_box_args** – The hit box algorithm arguments
- **vertex_order** (`Tuple[int, int, int, int]`) – The vertex order

Returns
str

classmethod create_empty(*name: str*, *size: Tuple[int, int]*, *color: Tuple[int, int, int, int] = (0, 0, 0, 0)*) → *Texture*

Create a texture with all pixels set to transparent black.

The hit box of the returned Texture will be set to a rectangle with the dimensions in `size` because there is no non-blank pixel data to calculate a hit box.

Parameters

- **name** – The unique name for this texture
- **size** – The xy size of the internal image

This function has multiple uses, including:

- Allocating space in texture atlases
- Generating custom cached textures from component images

The internal image can be altered with Pillow draw commands and then written/updated to a texture atlas. This works best for infrequent changes such as generating custom cached sprites. For frequent texture changes, you should instead render directly into the texture atlas.

Warning: If you plan to alter images using Pillow, read its documentation thoroughly! Some of the functions can have unexpected behavior.

For example, if you want to draw one or more images that contain transparency onto a base image that also contains transparency, you will likely need to use PIL.Image.alpha_composite as part of your solution. Otherwise, blending may behave in unexpected ways.

This is especially important for customizable characters.

Be careful of your RAM usage when using this function. The Texture this method returns will have a new internal RGBA Pillow image which uses 4 bytes for every pixel in it. This will quickly add up if you create many large Textures.

If you want to create more than one blank texture with the same dimensions, you can save CPU time and RAM by calling this function once, then passing the `image` attribute of the resulting Texture object to the class constructor for each additional blank Texture instance you would like to create. This can be especially helpful if you are creating multiple large Textures.

classmethod create_filled(*name: str*, *size: Tuple[int, int]*, *color: Tuple[int, int, int, int]*) → *Texture*

Create a filled texture. This is an alias for `create_empty()`.

Parameters

- **name** – Name of the texture
- **size** (*Tuple[int, int]*) – Size of the texture
- **color** – Color of the texture

Returns

Texture

classmethod create_image_cache_name(*path: str | Path*, *crop: Tuple[int, int, int, int] = (0, 0, 0, 0)*)

crop(*x: int*, *y: int*, *width: int*, *height: int*) → *Texture*

Create a new texture from a sub-section of this texture.

If the crop is the same size as the original texture or the crop is 0 width or height, the original texture is returned.

Parameters

- **x** – X position to start crop
- **y** – Y position to start crop
- **width** – Width of crop
- **height** – Height of crop
- **cache** – If True, the cropped texture will be cached

Returns
Texture

draw_scaled(*center_x: float*, *center_y: float*, *scale: float = 1.0*, *angle: float = 0.0*, *alpha: int = 255*)

Draw the texture.

Warning: This is a very slow method of drawing a texture, and should be used sparingly. The method simply creates a sprite internally and draws it.

Parameters

- **center_x** – X location of where to draw the texture.
- **center_y** – Y location of where to draw the texture.
- **scale** – Scale to draw rectangle. Defaults to 1.
- **angle** – Angle to rotate the texture by.
- **alpha** – The transparency of the texture *(0-255)*.

draw_sized(*center_x: float*, *center_y: float*, *width: float*, *height: float*, *angle: float = 0.0*, *alpha: int = 255*)

Draw a texture with a specific width and height.

Warning: This is a very slow method of drawing a texture, and should be used sparingly. The method simply creates a sprite internally and draws it.

Parameters

- **center_x** – X position to draw texture
- **center_y** – Y position to draw texture
- **width** – Width to draw texture
- **height** – Height to draw texture
- **angle** – Angle to draw texture
- **alpha** – Alpha value to draw texture

flip_diagonally() → *Texture*

Returns a new texture that is flipped diagonally from this texture. This is an alias for `transpose()`.

This returns a new texture with the same image data, but has updated hit box data and a transform that will be applied to the image when it's drawn (GPU side).

Returns
Texture

`flip_horizontally()` → *Texture*

Flip the texture left to right / horizontally.

This returns a new texture with the same image data, but has updated hit box data and a transform that will be applied to the image when it's drawn (GPU side).

Returns
Texture

`flip_left_right()` → *Texture*

Flip the texture left to right / horizontally.

This returns a new texture with the same image data, but has updated hit box data and a transform that will be applied to the image when it's drawn (GPU side).

Returns
Texture

`flip_top_bottom()` → *Texture*

Flip the texture top to bottom / vertically.

This returns a new texture with the same image data, but has updated hit box data and a transform that will be applied to the image when it's drawn (GPU side).

Returns
Texture

`flip_vertically()` → *Texture*

Flip the texture top to bottom / vertically.

This returns a new texture with the same image data, but has updated hit box data and a transform that will be applied to the image when it's drawn (GPU side).

Returns
Texture

`remove_atlas_ref`(*atlas:* TextureAtlas) → None

Remove a reference to an atlas that this texture is in.

`remove_from_atlases()` → None

Remove this texture from all atlases.

`remove_from_cache`(*ignore_error: bool = True*) → None

Remove this texture from the cache.

Parameters
`ignore_error` – If True, ignore errors if the texture is not in the cache

Returns
None

`rotate_180()` → *Texture*

Rotate the texture 180 degrees.

This returns a new texture with the same image data, but has updated hit box data and a transform that will be applied to the image when it's drawn (GPU side).

Returns
Texture

`rotate_270()` → *Texture*

Rotate the texture 270 degrees.

This returns a new texture with the same image data, but has updated hit box data and a transform that will be applied to the image when it's drawn (GPU side).

Returns
Texture

`rotate_90`(*count: int = 1*) → *Texture*

Rotate the texture by a given number of 90 degree steps.

This returns a new texture with the same image data, but has updated hit box data and a transform that will be applied to the image when it's drawn (GPU side).

Parameters
`count` – Number of 90 degree steps to rotate.

Returns
Texture

`transform`(*transform: Type[*Transform*]*) → *Texture*

Create a new texture with the given transform applied.

Parameters
`transform` – Transform to apply

Returns
New texture

`transpose()` → *Texture*

Returns a new texture that is transposed from this texture. This flips the texture diagonally from lower right to upper left.

This returns a new texture with the same image data, but has updated hit box data and a transform that will be applied to the image when it's drawn (GPU side).

Returns
Texture

`transverse()` → *Texture*

Returns a new texture that is transverse from this texture. This flips the texture diagonally from lower left to upper right.

This returns a new texture with the same image data, but has updated hit box data and a transform that will be applied to the image when it's drawn (GPU side).

Returns
Texture

`static validate_crop`(*image: Image, x: int, y: int, width: int, height: int*) → None

Validate the crop values for a given image.

`atlas_name`

The name of the texture used for the texture atlas (read only).

Returns
str

`cache_name`

The name of the texture used for caching (read only).

Returns
str

`crop_values`

The crop values used to create this texture in the referenced file

Returns
Tuple[int, int, int, int]

`file_path`

A Path object to the file this texture was loaded from

Returns
Path

`height`

The virtual width of the texture in pixels.

This can be different from the actual width if the texture has been transformed or the size have been set manually.

`hit_box_algorithm`

(read only) The algorithm used to calculate the hit box for this texture.

`hit_box_points`

Get the hit box points for this texture.

Custom hit box points must be supplied during texture creation and should ideally not be changed after creation.

Returns
PointList

`image`

Get or set the image of the texture.

Warning: This is an advanced function. Be absolutely sure you know the consequences of changing the image. It can cause problems with the texture atlas and hit box points.

Parameters
`image` – The image to set

`image_data`

The image data of the texture (read only).

This is a simple wrapper around the image containing metadata like hash and is used to determine the uniqueness of the image in texture atlases.

Returns
ImageData

`properties`

A dictionary of properties for this texture. This can be used to store any data you want.

Returns
Dict[str, Any]

size
The virtual size of the texture in pixels.

This can be different from the actual width if the texture has been transformed or the size have been set manually.

width
The virtual width of the texture in pixels. This can be different from the actual width if the texture has been transformed or the size have been set manually.

class arcade.SpriteSheet(*path: str | Path | None = None*, *image: Image | None = None*)
Bases:

Class to hold a sprite sheet. A sprite sheet is a single image that contains multiple textures. Textures can be created from the sprite sheet by cropping out sections of the image.

This is only a utility class. It does not have any special functionality

Parameters

- **path** – Path to the file to load.
- **image** – PIL image to use.

crop(*area: Tuple[int, int, int, int] | List[int]*)
Crop a texture from the sprite sheet.

Parameters
area – Area to crop `(x, y, width, height)`

crop_grid(*size: Tuple[int, int]*, *columns: int*, *count: int*, *margin: Tuple[int, int, int, int] | List[int] = (0, 0, 0, 0)*, *hit_box_algorithm: HitBoxAlgorithm | None = None*) → List[*Texture*]
Crop a grid of textures from the sprite sheet.

Parameters

- **size** – Size of each texture `(width, height)`
- **columns** – Number of columns in the grid
- **count** – Number of textures to crop
- **margin** – The margin around each texture `(left, right, bottom, top)`
- **hit_box_algorithm** – Hit box algorithm to use for the textures.

crop_sections(*sections: List[Tuple[int, int, int, int] | List[int]]*)
Crop multiple textures from the sprite sheet by specifying a list of areas to crop.

Parameters
sections – List of areas to crop `[(x, y, width, height), ...]`

flip_left_right() → None
Flip the sprite sheet left/right.

flip_top_bottom()
Flip the sprite sheet up/down.

classmethod from_image(*image: Image*)
Create a sprite sheet from a PIL image.

Parameters
image – PIL image to use.

flip_flags

Query the orientation of the sprite sheet. This can be used to determine if the sprite sheet needs to be flipped.

Default values are `(False, False)`. Will be modified when *flip_left_right()* or *flip_top_bottom()* is called.

Returns
Tuple of booleans `(flip_left_right, flip_top_bottom)`.

image

Get or set the PIL image for this sprite sheet.

path

The path to the sprite sheet.

Returns
The path.

33.12 Texture Transforms

class arcade.texture.transforms.**FlipLeftRightTransform**

Bases: *Transform*

Flip texture horizontally / left to right.

static **transform_hit_box_points**(*points: Sequence[Tuple[float, float]]*) → Sequence[Tuple[float, float]]

order = (1, 0, 3, 2)

How texture coordinates order should be changed for this transform. upper_left, upper_right, lower_left, lower_right

class arcade.texture.transforms.**FlipTopBottomTransform**

Bases: *Transform*

Flip texture vertically / top to bottom.

static **transform_hit_box_points**(*points: Sequence[Tuple[float, float]]*) → Sequence[Tuple[float, float]]

order = (2, 3, 0, 1)

How texture coordinates order should be changed for this transform. upper_left, upper_right, lower_left, lower_right

class arcade.texture.transforms.**Rotate180Transform**

Bases: *Transform*

Rotate 180 degrees clockwise.

static **transform_hit_box_points**(*points: Sequence[Tuple[float, float]]*) → Sequence[Tuple[float, float]]

order = (3, 2, 1, 0)

How texture coordinates order should be changed for this transform. upper_left, upper_right, lower_left, lower_right

class arcade.texture.transforms.**Rotate270Transform**

Bases: *Transform*

Rotate 270 degrees clockwise.

static transform_hit_box_points(*points: Sequence[Tuple[float, float]]*) → Sequence[Tuple[float, float]]

order = (1, 3, 0, 2)

How texture coordinates order should be changed for this transform. upper_left, upper_right, lower_left, lower_right

class arcade.texture.transforms.**Rotate90Transform**

Bases: *Transform*

Rotate 90 degrees clockwise.

static transform_hit_box_points(*points: Sequence[Tuple[float, float]]*) → Sequence[Tuple[float, float]]

order = (2, 0, 3, 1)

How texture coordinates order should be changed for this transform. upper_left, upper_right, lower_left, lower_right

class arcade.texture.transforms.**Transform**

Bases:

Base class for all texture transforms.

Transforms are responsible for transforming the texture coordinates and hit box points.

static transform_hit_box_points(*points: Sequence[Tuple[float, float]]*) → Sequence[Tuple[float, float]]

Transforms hit box points.

classmethod transform_texture_coordinates_order(*texture_coordinates: Tuple[float, float, float, float, float, float, float, float], order: Tuple[int, int, int, int]*) → Tuple[float, float, float, float, float, float, float, float]

Change texture coordinates order.

Parameters

- **texture_coordinates** – Texture coordinates to transform
- **order** – The new order

classmethod transform_vertex_order(*order: Tuple[int, int, int, int]*) → Tuple[int, int, int, int]

Transforms and exiting vertex order with this transform. This gives us important metadata on how to quickly transform the texture coordinates without iterating all applied transforms.

order = (0, 1, 2, 3)

How texture coordinates order should be changed for this transform. upper_left, upper_right, lower_left, lower_right

class arcade.texture.transforms.**TransposeTransform**

Bases: *Transform*

Transpose texture.

static transform_hit_box_points(*points: Sequence[Tuple[float, float]]*) → Sequence[Tuple[float, float]]

order = (0, 2, 1, 3)

How texture coordinates order should be changed for this transform. upper_left, upper_right, lower_left, lower_right

class arcade.texture.transforms.**TransverseTransform**

Bases: `Transform`

Transverse texture.

static transform_hit_box_points(*points: Sequence[Tuple[float, float]]*) → Sequence[Tuple[float, float]]

order = (3, 1, 2, 0)

How texture coordinates order should be changed for this transform. upper_left, upper_right, lower_left, lower_right

class arcade.texture.transforms.**VertexOrder**(*value*)

Bases: Enum

Order for texture coordinates.

LOWER_LEFT = 2

LOWER_RIGHT = 3

UPPER_LEFT = 0

UPPER_RIGHT = 1

arcade.texture.transforms.**get_orientation**(*order: Tuple[int, int, int, int]*) → int

Get orientation info from the vertex order

33.13 Texture Atlas

class arcade.texture_atlas.**AtlasRegion**(*atlas:* TextureAtlas, *x: int*, *y: int*, *width: int*, *height: int*, *texture_coordinates: Tuple[float, float, float, float, float, float, float, float] | None = None*)

Bases:

Stores information about where a texture is located.

The texture coordinates are stored as a tuple of 8 floats (4 points, 2 floats each) in the following order: upper_left, upper_right, lower_left, lower_right.

Layout:

```
(0, 1)                                  (1, 1)
+--------------------------------------+
|          Atlas Texture               |
|                                      |
| (2)                (3)               |
+------------------+                   |
|   Image          |                   |
```

(continues on next page)

(continued from previous page)

```
|                   |                      |
|                   |                      |
|                   |                      |
|                   |                      |
| (0)               | (1)                  |
+-------------------+----------------------+
(0, 0)                                     (1, 0)
```

Parameters

- **atlas** – The atlas this region belongs to
- **texture** – The arcade texture
- **x** – The x position of the texture
- **y** – The y position of the texture
- **width** – The width of the texture in pixels
- **height** – The height of the texture in pixels
- **texture_coordinates** – The texture coordinates (optional)

verify_image_size(*image_data: ImageData*)

Verify the image has the right size. The internal image of a texture can be tampered with at any point causing an atlas update to fail.

height

texture_coordinates

width

x

y

class arcade.texture_atlas.**TextureAtlas**(*size: Tuple[int, int], *, border: int = 1, textures: Sequence[Texture] | None = None, auto_resize: bool = True, ctx: ArcadeContext | None = None, capacity: int = 2*)

Bases: `TextureAtlasBase`

A texture atlas with a size in a context.

A texture atlas is a large texture containing several textures so OpenGL can easily batch draw thousands or hundreds of thousands of sprites on one draw operation.

This is a fairly simple atlas that stores horizontal strips were the height of the strip is the texture/image with the larges height.

Adding a texture to this atlas generates a texture id. This id is used the sprite list vertex data to reference what texture each sprite is using. The actual texture coordinates are located in a float32 texture this atlas is responsible for keeping up to date.

The atlas deals with image and textures. The image is the actual image data. The texture is the arcade texture object that contains the image and other information about such as transforms. Several textures can share the same image with different transforms applied. The transforms are simply changing the order of the texture coordinates to flip, rotate or mirror the image.

Parameters

- **size** (*Tuple[int, int]*) – The width and height of the atlas in pixels
- **border** – Currently no effect; Should always be 1 to avoid textures bleeding
- **textures** – The texture for this atlas
- **auto_resize** – Automatically resize the atlas when full
- **ctx** – The context for this atlas (will use window context if left empty)
- **capacity** – The number of textures the atlas keeps track of. This is multiplied by 4096. Meaning capacity=2 is 8192 textures. This value can affect the performance of the atlas.

add(*texture:* Texture) → Tuple[int, *AtlasRegion*]

Add a texture to the atlas.

Parameters
texture – The texture to add

Returns
texture_id, AtlasRegion tuple

Raises
AllocatorException – If there are no room for the texture

classmethod **calculate_minimum_size**(*textures: Sequence[*Texture*], border: int = 1*)

Calculate the minimum atlas size needed to store the the provided sequence of textures

Parameters

- **textures** – Sequence of textures
- **border** – The border around each texture in pixels

Returns
An estimated minimum size as a (width, height) tuple

classmethod **create_from_texture_sequence**(*textures: Sequence[*Texture*], border: int = 1*) → *TextureAtlas*

Create a texture atlas of a reasonable size from a sequence of textures.

Parameters

- **textures** – A sequence of textures (list, set, tuple, generator etc.)
- **border** – The border for the atlas in pixels (space between each texture)

get_image_region_info(*hash: str*) → *AtlasRegion*

Get the region info for and image by has

Parameters
hash – The hash of the image

Returns
The AtlasRegion for the given texture name

get_texture_id(*texture:* Texture) → int

Get the internal id for a Texture in the atlas

Parameters
atlas_name – The name of the texture in the atlas

Returns
The texture id for the given texture name

Raises
`Exception` – If the texture is not in the atlas

get_texture_region_info(*atlas_name: str*) → *AtlasRegion*

Get the region info for a texture by atlas name

Returns
The AtlasRegion for the given texture name

has_image(*image_data: ImageData*) → bool

Check if a image is already in the atlas

has_texture(*texture:* Texture) → bool

Check if a texture is already in the atlas

has_unique_texture(*texture:* Texture) → bool

Check if the atlas already have a texture with the same image data and vertex order

read_texture_image_from_atlas(*texture:* Texture) → Image

Read the pixel data for a texture directly from the atlas texture on the GPU. The contents of this image can be altered by rendering into the atlas and is useful in situations were you need the updated pixel data on the python side.

Parameters
texture – The texture to get the image for

Returns
A pillow image containing the pixel data in the atlas

rebuild() → None

Rebuild the underlying atlas texture.

This method also tries to organize the textures more efficiently ordering them by size. The texture ids will persist so the sprite list don't need to be rebuilt.

remove(*texture:* Texture) → None

Remove a texture from the atlas.

This doesn't erase the pixel data from the atlas texture itself, but leaves the area unclaimed. The area will be reclaimed when the atlas is rebuilt.

Parameters
texture – The texture to remove

render_into(*texture:* Texture, *projection: Tuple[float, float, float, float] | None = None*)

Render directly into a sub-section of the atlas. The sub-section is defined by the already allocated space of the texture supplied in this method.

By default the projection will be set to match the texture area size were *0, 0* is the lower left corner and *width, height* (of texture) is the upper right corner.

This method should should be used with the `with` statement:

```python
with atlas.render_into(texture):
    # Draw commands here

# Specify projection
```

(continues on next page)

(continued from previous page)

```
with atlas.render_into(texture, projection=(0, 100, 0, 100))
    # Draw geometry
```

Parameters

- **texture** – The texture area to render into
- **projection** – The ortho projection to render with. This parameter can be left blank if no projection changes are needed. The tuple values are: (left, right, button, top)

resize(*size: Tuple[int, int]*) → None

Resize the atlas on the gpu.

This will copy the pixel data from the old to the new atlas retaining the exact same data. This is useful if the atlas was rendered into directly and we don't have to transfer each texture individually from system memory to graphics memory.

Parameters
size – The new size

save(*path: str | Path*, *flip: bool = False*, *components: int = 4*, *draw_borders: bool = False*, *border_color: Tuple[int, int, int] = (255, 0, 0)*) → None

Save the texture atlas to a png.

Borders can also be drawn into the image to visualize the regions of the atlas.

Parameters

- **path** – The path to save the atlas on disk
- **flip** – Flip the image horizontally
- **components** – Number of components. (3 = RGB, 4 = RGBA)
- **color** – RGB color of the borders

Returns
A pillow image containing the atlas texture

show(*flip: bool = False*, *components: int = 4*, *draw_borders: bool = False*, *border_color: Tuple[int, int, int] = (255, 0, 0)*) → None

Show the texture atlas using Pillow.

Borders can also be drawn into the image to visualize the regions of the atlas.

Parameters

- **flip** – Flip the image horizontally
- **components** – Number of components. (3 = RGB, 4 = RGBA)
- **draw_borders** – Draw region borders into image
- **color** – RGB color of the borders

to_image(*flip: bool = False*, *components: int = 4*, *draw_borders: bool = False*, *border_color: Tuple[int, int, int] = (255, 0, 0)*) → Image

Convert the atlas to a Pillow image.

Borders can also be drawn into the image to visualize the regions of the atlas.

Parameters

- **flip** – Flip the image horizontally
- **components** – Number of components. (3 = RGB, 4 = RGBA)
- **draw_borders** – Draw region borders into image
- **color** – RGB color of the borders

Returns
A pillow image containing the atlas texture

update_texture_image(*texture:* Texture)

Updates the internal image of a texture in the atlas texture. The new image needs to be the exact same size as the original one meaning the texture already need to exist in the atlas.

This can be used in cases were the image is manipulated in some way and we need a quick way to sync these changes to graphics memory. This operation is fairly expensive, but still orders of magnitude faster than removing the old texture, adding the new one and re-building the entire atlas.

Parameters
texture – The texture to update

update_texture_image_from_atlas(*texture:* Texture) → None

Update the arcade Texture's internal image with the pixel data content from the atlas texture on the GPU. This can be useful if you render into the atlas and need to update the texture with the new pixel data.

Parameters
texture – The texture to update

use_uv_texture(*unit: int = 0*) → None

Bind the texture coordinate texture to a channel. In addition this method writes the texture coordinate to the texture if the data is stale. This is to avoid a full update every time a texture is added to the atlas.

Parameters
unit – The texture unit to bind the uv texture

write_image(*image: Image*, *x: int*, *y: int*) → None

Write a PIL image to the atlas in a specific region.

Parameters

- **image** – The pillow image
- **x** – The x position to write the texture
- **y** – The y position to write the texture

auto_resize

Get or set the auto resize flag for the atlas. If enabled the atlas will resize itself when full.

border

The texture border in pixels

fbo

The framebuffer object for this atlas

height

The height of the texture atlas in pixels

image_uv_texture

Texture coordinate texture for images.

images

Return a list of all the images in the atlas.

A new list is constructed from the internal weak set of images.

max_height

The maximum height of the atlas in pixels

max_size

The maximum size of the atlas in pixels (x, y)

max_width

The maximum width of the atlas in pixels

size

The width and height of the texture atlas in pixels

texture

The atlas texture.

texture_uv_texture

Texture coordinate texture for textures.

textures

All textures instance added to the atlas regardless of their internal state. See `unique_textures`()` for textures with unique image data and transformation.

unique_textures

All unique textures in the atlas.

These are textures using an image with the same hash and the same vertex order. The full list of all textures can be found in `textures()`.

width

The width of the texture atlas in pixels

class arcade.texture_atlas.**TextureAtlasBase**(*ctx: 'ArcadeContext' | None*)

Bases: `ABC`

Generic base for texture atlases.

abstract remove(*texture:* Texture) → None

Remove a texture from the atlas.

ctx

33.14 Performance Information

class arcade.**PerfGraph**(*width: int, height: int, graph_data: str = 'FPS', update_rate: float = 0.1, background_color: Tuple[int, int, int, int] = (0, 0, 0, 255), data_line_color: Tuple[int, int, int, int] = (255, 255, 255, 255), axis_color: Tuple[int, int, int, int] = (155, 135, 12, 255), grid_color: Tuple[int, int, int, int] = (155, 135, 12, 255), font_color: Tuple[int, int, int, int] = (255, 255, 255, 255), font_size: int = 10, y_axis_num_lines: int = 4, view_y_scale_step: float = 20.0*)

Bases: `Sprite`

An auto-updating line chart of FPS or event handler execution times.

You must use *`arcade.enable_timings()`* to turn on performance tracking for the chart to display data.

Aside from instantiation and updating the chart, this class behaves like other *`arcade.Sprite`* instances. You can use it with *`SpriteList`* normally. See performance_statistics_example for an example of how to use this class.

Unlike other *`Sprite`* instances, this class neither loads an *`arcade.Texture`* nor accepts one as a constructor argument. Instead, it creates a new internal *`Texture`* instance. The chart is automatically redrawn to this internal *`Texture`* every `update_rate` seconds.

Parameters

- **`width`** – The width of the chart texture in pixels
- **`height`** – The height of the chart texture in pixels
- **`graph_data`** – The pyglet event handler or statistic to track
- **`update_rate`** – How often the graph updates, in seconds
- **`background_color`** – The background color of the chart
- **`data_line_color`** – Color of the line tracking drawn
- **`axis_color`** – The color to draw the x & y axes in
- **`font_color`** – The color of the label font
- **`font_size`** – The size of the label font in points
- **`y_axis_num_lines`** – How many grid lines should be used to divide the y scale of the graph.
- **`view_y_scale_step`** – The graph's view area will be scaled to a multiple of this value to fit to the data currently displayed.

`remove_from_sprite_lists()`

Remove the sprite from all lists and cancel the update event.

Returns

`update_graph`(*delta_time: float*)

Update the graph by redrawing the internal texture data.

> **Warning:** You do not need to call this method! It will be called automatically!

Parameters

`delta_time` – Elapsed time in seconds. Passed by the pyglet scheduler.

`axis_color`

`background_color`

`boundary_bottom`: `float | None`

`boundary_left`: `float | None`

`boundary_right`: `float | None`

`boundary_top`: `float | None`

`change_angle`: `float`

cur_texture_index: int

font_color

font_size

force

grid_color

guid: str | None

physics_engines: List[Any]

textures: List[*Texture*]

arcade.**clear_timings**() → None

Reset the count & average time for each event type to zero.

Performance tracking must be enabled with *arcade.enable_timings()* before calling this function.

See performance_statistics_example for an example of how to use function.

arcade.**disable_timings**() → None

Disable collection of timing information.

Performance tracking must be enabled with *arcade.enable_timings()* before calling this function.

arcade.**enable_timings**(*max_history: int = 100*) → None

Enable recording of performance information.

This function must be called before using any other performance features, except for *arcade.timings_enabled()*, which can be called at any time.

See performance_statistics_example for an example of how to use function.

Parameters
max_history – How many frames to keep performance info for.

arcade.**get_fps**(*frame_count: int = 60*) → float

Get the FPS over the last `frame_count` frames.

Performance tracking must be enabled with *arcade.enable_timings()* before calling this function.

To get the FPS over the last 30 frames, you would pass 30 instead of the default 60.

See performance_statistics_example for an example of how to use function.

Parameters
frame_count – How many frames to calculate the FPS over.

arcade.**get_timings**() → Dict

Get a dict of the current dispatch event timings.

Performance tracking must be enabled with *arcade.enable_timings()* before calling this function.

Returns
A dict of event timing data, consisting of counts and average handler duration.

`arcade.print_timings()`

Print event handler statistics to stdout as a table.

Performance tracking must be enabled with `arcade.enable_timings()` before calling this function.

See performance_statistics_example for an example of how to use function.

The statistics consist of:

- how many times each registered event was called
- the average time for handling each type of event in seconds

The table looks something like:

```
Event          Count Average Time
-------------- ----- ------------
on_update         60       0.0000
on_mouse_enter     1       0.0000
on_mouse_motion   39       0.0000
on_expose          1       0.0000
on_draw           60       0.0020
```

`arcade.timings_enabled()` → bool

Return true if timings are enabled, false otherwise.

This function can be used at any time to check if timings are enabled. See `arcade.enable_timings()` for more information.

Returns
Whether timings are currently enabled.

33.15 Physics Engines

`class arcade.PymunkException`

Bases: `Exception`

Exception raised for errors in the PymunkPhysicsEngine.

`class arcade.PymunkPhysicsEngine`(*gravity=(0, 0)*, *damping: float = 1.0*, *maximum_incline_on_ground: float = 0.708*)

Bases:

Pymunk Physics Engine

Parameters

- **`gravity`** – The direction where gravity is pointing
- **`damping`** – The amount of speed which is kept to the next tick. A value of 1.0 means no speed loss, while 0.9 has 10% loss of speed etc.
- **`maximum_incline_on_ground`** – The maximum incline the ground can have, before is_on_ground() becomes False default = 0.708 or a little bit over 45° angle

`add_collision_handler`(*first_type: str*, *second_type: str*, *begin_handler: Callable | None = None*, *pre_handler: Callable | None = None*, *post_handler: Callable | None = None*, *separate_handler: Callable | None = None*)

Add code to handle collisions between objects.

add_sprite(*sprite: Sprite, mass: float = 1, friction: float = 0.2, elasticity: float | None = None, moment_of_inertia: float | None = None, body_type: int = 0, damping: float | None = None, gravity: Vec2d | Tuple[float, float] | Vec2 | None = None, max_velocity: int | None = None, max_horizontal_velocity: int | None = None, max_vertical_velocity: int | None = None, radius: float = 0, collision_type: str | None = 'default'*)

Add a sprite to the physics engine.

Parameters

- **sprite** – The sprite to add.
- **mass** – The mass of the object. Defaults to 1.
- **friction** – The friction the object has. Defaults to 0.2.
- **elasticity** – How bouncy this object is. 0 is no bounce. Values of 1.0 and higher may behave badly.
- **moment_of_inertia** – The moment of inertia, or force needed to change angular momentum. Providing infinite makes this object stuck in its rotation.
- **body_type** – The type of the body. Defaults to Dynamic, meaning, the body can move, rotate etc. Providing STATIC makes it fixed to the world.
- **damping** – See class docs.
- **gravity** – See class docs.
- **max_velocity** – The maximum velocity of the object.
- **max_horizontal_velocity** – Maximum velocity on the x axis in pixels.
- **max_vertical_velocity** – Maximum velocity on the y axis in pixels.
- **radius** – Radius for the shape created for the sprite in pixels.
- **collision_type** – Assign a name to the sprite, use this name when adding collision handler.

add_sprite_list(*sprite_list, mass: float = 1, friction: float = 0.2, elasticity: float | None = None, moment_of_inertia: float | None = None, body_type: int = 0, damping: float | None = None, collision_type: str | None = None*)

Add all sprites in a sprite list to the physics engine.

apply_force(*sprite: Sprite, force: Tuple[float, float]*)

Apply force to a Sprite.

apply_impulse(*sprite: Sprite, impulse: Tuple[float, float]*)

Apply an impulse force on a sprite

apply_opposite_running_force(*sprite: Sprite*)

If a sprite goes left while on top of a dynamic sprite, that sprite should get pushed to the right.

check_grounding(*sprite: Sprite*)

See if the player is on the ground. Used to see if we can jump.

get_physics_object(*sprite: Sprite*) → *PymunkPhysicsObject*

Get the shape/body for a sprite.

get_sprite_for_shape(*shape: Shape | None*) → *Sprite* | None

Given a shape, what sprite is associated with it?

get_sprites_from_arbiter(*arbiter: Arbiter*) → Tuple[*Sprite* | None, *Sprite* | None]

Given a collision arbiter, return the sprites associated with the collision.

is_on_ground(*sprite:* Sprite) → bool

Return true of sprite is on top of something.

remove_sprite(*sprite:* Sprite)

Remove a sprite from the physics engine.

resync_sprites()

Set visual sprites to be the same location as physics engine sprites. Call this after stepping the pymunk physics engine

set_friction(*sprite:* Sprite, *friction: float*)

Apply force to a Sprite.

set_horizontal_velocity(*sprite:* Sprite, *velocity: float*)

Set a sprite's velocity

set_position(*sprite:* Sprite, *position: Vec2d* | *Tuple[float, float]*)

Apply an impulse force on a sprite

set_rotation(*sprite:* Sprite, *rotation: float*)

set_velocity(*sprite:* Sprite, *velocity: Tuple[float, float]*)

Apply an impulse force on a sprite

step(*delta_time: float = 0.016666666666666666*, *resync_sprites: bool = True*)

Tell the physics engine to perform calculations.

Parameters

- **delta_time** – Time to move the simulation forward. Keep this value constant, do not use varying values for each step.
- **resync_sprites** – Resynchronize Arcade graphical sprites to be at the same location as their Pymunk counterparts. If running multiple steps per frame, set this to false for the first steps, and true for the last step that's part of the update.

DYNAMIC = 0

KINEMATIC = 1

MOMENT_INF = inf

STATIC = 2

class arcade.**PymunkPhysicsObject**(*body: Body* | *None* = *None*, *shape: Shape* | *None* = *None*)

Bases:

Object that holds pymunk body/shape for a sprite.

class arcade.**PhysicsEnginePlatformer**(*player_sprite:* Sprite, *platforms:* SpriteList | *Iterable*[SpriteList] | *None* = *None*, *gravity_constant: float* = *0.5*, *ladders:* SpriteList | *Iterable*[SpriteList] | *None* = *None*, *walls:* SpriteList | *Iterable*[SpriteList] | *None* = *None*)

Bases:

Simplistic physics engine for use in a platformer. It is easier to get started with this engine than more sophisticated engines like PyMunk.

Note: Sending static sprites to the `walls` parameter and moving sprites to the `platforms` parameter will have very extreme benefits to performance.

Note: This engine will automatically move any Sprites sent to the `platforms` parameter between a `boundary_top` and `boundary_bottom` or a `boundary_left` and `boundary_right` attribute of the Sprite. You need only set an initial `change_x` or `change_y` on it.

Parameters

- **`player_sprite`** – The moving sprite
- **`platforms`** (*`Optional[Union[SpriteList, Iterable[SpriteList]]]`*) – Sprites the player can't move through. This value should only be used for moving Sprites. Static sprites should be sent to the `walls` parameter.
- **`gravity_constant`** – Downward acceleration per frame
- **`ladders`** (*`Optional[Union[SpriteList, Iterable[SpriteList]]]`*) – Ladders the user can climb on
- **`walls`** (*`Optional[Union[SpriteList, Iterable[SpriteList]]]`*) – Sprites the player can't move through. This value should only be used for static Sprites. Moving sprites should be sent to the `platforms` parameter.

`can_jump`(*y_distance: float = 5*) → bool

Method that looks to see if there is a floor under the player_sprite. If there is a floor, the player can jump and we return a True.

Returns
True if there is a platform below us

`disable_multi_jump`()

Disables multi-jump.

Calling this function also removes the requirement to call increment_jump_counter() every time the player jumps.

`enable_multi_jump`(*allowed_jumps: int*)

Enables multi-jump. allowed_jumps should include the initial jump. (1 allows only a single jump, 2 enables double-jump, etc)

If you enable multi-jump, you MUST call increment_jump_counter() every time the player jumps. Otherwise they can jump infinitely.

Parameters
`allowed_jumps` –

`increment_jump_counter`()

Updates the jump counter for multi-jump tracking

`is_on_ladder`()

Return 'true' if the player is in contact with a sprite in the ladder list.

`jump`(*velocity: int*)

Have the character jump.

`update`()

Move everything and resolve collisions.

Returns
SpriteList with all sprites contacted. Empty list if no sprites.

`ladders`
The ladder list registered with the physics engine.

`class arcade.`**`PhysicsEngineSimple`**`(`*player_sprite:* Sprite, *walls:* SpriteList*[*BasicSprite*]* | *Iterable[*SpriteList*[*BasicSprite*]]*`)`

Bases:

Simplistic physics engine for use in games without gravity, such as top-down games. It is easier to get started with this engine than more sophisticated engines like PyMunk.

Parameters

- **`player_sprite`** – The moving sprite
- **`walls`** (`Union[SpriteList, Iterable[SpriteList]]`) – The sprites it can't move through. This can be one or multiple spritelists.

`update()`
Move everything and resolve collisions.

Returns
SpriteList with all sprites contacted. Empty list if no sprites.

33.16 Misc Utility Functions

`class arcade.utils.`**`ByteRangeError`**`(`*var_name: str*, *value: int*`)`

Bases: `IntOutsideRangeError`

An int was outside the range of 0 to 255 inclusive

Parameters

- **`var_name`** – the name of the variable or argument
- **`value`** – the value to fall outside the expected range

`class arcade.utils.`**`FloatOutsideRangeError`**`(`*var_name: str*, *value: float*, *lower: float*, *upper: float*`)`

Bases: `OutsideRangeError`

A float value was outside an expected range

Parameters

- **`var_name`** – the name of the variable or argument
- **`value`** – the value to fall outside the expected range
- **`lower`** – the lower bound, inclusive, of the range
- **`upper`** – the upper bound, inclusive, of the range

`class arcade.utils.`**`IntOutsideRangeError`**`(`*var_name: str*, *value: int*, *lower: int*, *upper: int*`)`

Bases: `OutsideRangeError`

An integer was outside an expected range

This class was originally intended to assist deserialization from data packed into ints, such as `Color`.

Parameters

- **`var_name`** – the name of the variable or argument
- **`value`** – the value to fall outside the expected range

- **lower** – the lower bound, inclusive, of the range
- **upper** – the upper bound, inclusive, of the range

class arcade.utils.**NormalizedRangeError**(*var_name: str*, *value: float*)

Bases: `FloatOutsideRangeError`

A float was not between 0.0 and 1.0, inclusive

Note that normalized floats should not normally be bound-checked as before drawing as this is taken care of on the GPU side.

The exceptions to this are when processing data on the Python side, especially when it is cheaper to bound check two floats than call clamping functions.

Parameters

- **var_name** – the name of the variable or argument
- **value** – the value to fall outside the expected range

class arcade.utils.**OutsideRangeError**(*var_name: str*, *value: _CT*, *lower: _CT*, *upper: _CT*)

Bases: `ValueError`

Raised when a value is outside and expected range

This class and its subclasses are intended to be arcade-internal helpers to clearly signal exactly what went wrong. Each helps type annotate and template a string describing exactly what went wrong.

Parameters

- **var_name** – the name of the variable or argument
- **value** – the value to fall outside the expected range
- **lower** – the lower bound, inclusive, of the range
- **upper** – the upper bound, inclusive, of the range

class arcade.utils.**PerformanceWarning**

Bases: `Warning`

Use this for issuing performance warnings.

class arcade.utils.**ReplacementWarning**

Bases: `Warning`

Use this for issuing warnings about naming and functionality changes.

arcade.utils.**generate_uuid_from_kwargs**(***kwargs*) → str

Given key/pair combos, returns a string in uuid format. Such as *text='hi', size=32* it will return "text-hi-size-32". Called with no parameters, id does NOT return a random unique id.

arcade.utils.**get_raspberry_pi_info**() → Tuple[bool, str, str]

Determine if the host is a raspberry pi with additional info.

Returns

3 component tuple. bool (is host a raspi) str (architecture) str (model name)

arcade.utils.**is_raspberry_pi**() → bool

Determine if the host is a raspberry pi.

Returns

bool

`arcade.utils.`**`warning`**(*warning_type: Type[Warning]*, *message: str = ''*, ***kwargs*)

`arcade.`**`configure_logging`**(*level: int | None = None*)

Set up basic logging. :param level: The log level. Defaults to DEBUG.

33.17 Geometry Support

`arcade.geometry.`**`are_lines_intersecting`**(*p1: Tuple[float, float]*, *q1: Tuple[float, float]*, *p2: Tuple[float, float]*, *q2: Tuple[float, float]*) → bool

Given two lines defined by points p1, q1 and p2, q2, the function returns true if the two lines intersect.

Parameters

- **`p1`** – Point 1
- **`q1`** – Point 2
- **`p2`** – Point 3
- **`q2`** – Point 4

Returns

True or false depending if lines intersect

`arcade.geometry.`**`are_polygons_intersecting`**(*poly_a: Sequence[Tuple[float, float]]*, *poly_b: Sequence[Tuple[float, float]]*) → bool

Return True if two polygons intersect.

Parameters

- **`poly_a`** – List of points that define the first polygon.
- **`poly_b`** – List of points that define the second polygon.

Returns

True or false depending if polygons intersect

`arcade.geometry.`**`get_triangle_orientation`**(*p: Tuple[float, float]*, *q: Tuple[float, float]*, *r: Tuple[float, float]*) → int

Find the orientation of a triangle defined by (p, q, r)

The function returns following integer values

- 0 –> p, q and r are collinear
- 1 –> Clockwise
- 2 –> Counterclockwise

Parameters

- **`p`** – Point 1
- **`q`** – Point 2
- **`r`** – Point 3

Returns

0, 1, or 2 depending on orientation

`arcade.geometry.`**`is_point_in_box`**`(`*p: Tuple[float, float], q: Tuple[float, float], r: Tuple[float, float]*`)` → bool

Return True if point q is inside the box defined by p and r.

Parameters

- **`p`** – Point 1
- **`q`** – Point 2
- **`r`** – Point 3

Returns

True or false depending if points are collinear

`arcade.geometry.`**`is_point_in_polygon`**`(`*x: float, y: float, polygon: Sequence[Tuple[float, float]]*`)` → bool

Checks if a point is inside a polygon of three or more points.

Parameters

- **`x`** – X coordinate of point
- **`y`** – Y coordinate of point
- **`polygon_point_list`** – List of points that define the polygon.

Returns

True or false depending if point is inside polygon

33.18 Game Controller Support

`class` `arcade.`**`ControllerManager`**

Bases: `ControllerManager`

A ControllerManager provides an interface for handling connect/disconnect events.

Please see Pyglet docs: https://pyglet.readthedocs.io/en/latest/programming_guide/input.html#controllermanager

`arcade.`**`get_controllers`**`()`

This returns a list of controllers, it is synonymous with calling `pyglet.input.get_controllers()`

33.19 Joystick Support

`arcade.`**`get_game_controllers`**`()` → List[Joystick]

Get a list of all the game controllers

Returns

List of game controllers

`arcade.`**`get_joysticks`**`()` → List[Joystick]

Get a list of all the game controllers

This is an alias of `get_game_controllers`, which is better worded.

Returns

List of game controllers

33.20 Window and View

`class arcade.NoOpenGLException`

Bases: `Exception`

Exception when we can't get an OpenGL 3.3+ context

`class arcade.View`(*window: Window | None = None*)

Bases:

Support different views/screens in a window.

`add_section`(*section, at_index: int | None = None, at_draw_order: int | None = None*) → None

Adds a section to the view Section Manager.

Parameters

- **section** – the section to add to this section manager
- **at_index** – inserts the section at that index for event capture and update events. If None at the end
- **at_draw_order** – inserts the section in a specific draw order. Overwrites section.draw_order

`clear`(*color: Tuple[int, int, int, int] | Tuple[float, float, float, float] | None = None, normalized: bool = False, viewport: Tuple[int, int, int, int] | None = None*)

Clears the View's Window with the configured background color set through `arcade.Window.background_color`.

Parameters

- **color** – (Optional) override the current background color with one of the following:

1. A `Color` instance
2. A 4-length RGBA `tuple` of byte values (0 to 255)
3. A 4-length RGBA `tuple` of normalized floats (0.0 to 1.0)

- **normalized** – If the color format is normalized (0.0 -> 1.0) or byte values
- **viewport** (`Tuple[int, int, int, int]`) – The viewport range to clear

`on_draw`()

Called when this view should draw

`on_hide_view`()

Called once when this view is hidden.

`on_key_press`(*symbol: int, modifiers: int*)

Override this function to add key press functionality.

Parameters

- **symbol** – Key that was hit
- **modifiers** – Bitwise 'and' of all modifiers (shift, ctrl, num lock) active during this event. See *Modifiers*.

on_key_release(*_symbol: int*, *_modifiers: int*)

Override this function to add key release functionality.

Parameters

- **_symbol** – Key that was hit
- **_modifiers** – Bitwise 'and' of all modifiers (shift, ctrl, num lock) active during this event. See *Modifiers*.

on_mouse_drag(*x: int*, *y: int*, *dx: int*, *dy: int*, *_buttons: int*, *_modifiers: int*)

Override this function to add mouse button functionality.

Parameters

- **x** – x position of mouse
- **y** – y position of mouse
- **dx** – Change in x since the last time this method was called
- **dy** – Change in y since the last time this method was called
- **_buttons** – Which button is pressed
- **_modifiers** – Bitwise 'and' of all modifiers (shift, ctrl, num lock) active during this event. See *Modifiers*.

on_mouse_enter(*x: int*, *y: int*)

Called when the mouse was moved into the window. This event will not be triggered if the mouse is currently being dragged.

Parameters

- **x** – x position of mouse
- **y** – y position of mouse

on_mouse_leave(*x: int*, *y: int*)

Called when the mouse was moved outside of the window. This event will not be triggered if the mouse is currently being dragged. Note that the coordinates of the mouse pointer will be outside of the window rectangle.

Parameters

- **x** – x position of mouse
- **y** – y position of mouse

on_mouse_motion(*x: int*, *y: int*, *dx: int*, *dy: int*)

Override this function to add mouse functionality.

Parameters

- **x** – x position of mouse
- **y** – y position of mouse
- **dx** – Change in x since the last time this method was called
- **dy** – Change in y since the last time this method was called

on_mouse_press(*x: int*, *y: int*, *button: int*, *modifiers: int*)

Override this function to add mouse button functionality.

Parameters

- **`x`** – x position of the mouse
- **`y`** – y position of the mouse
- **`button`** – What button was hit. One of: arcade.MOUSE_BUTTON_LEFT, arcade.MOUSE_BUTTON_RIGHT, arcade.MOUSE_BUTTON_MIDDLE
- **`modifiers`** – Bitwise 'and' of all modifiers (shift, ctrl, num lock) active during this event. See *Modifiers*.

`on_mouse_release`(*x: int*, *y: int*, *button: int*, *modifiers: int*)

Override this function to add mouse button functionality.

Parameters

- **`x`** – x position of mouse
- **`y`** – y position of mouse
- **`button`** – What button was hit. One of: arcade.MOUSE_BUTTON_LEFT, arcade.MOUSE_BUTTON_RIGHT, arcade.MOUSE_BUTTON_MIDDLE
- **`modifiers`** – Bitwise 'and' of all modifiers (shift, ctrl, num lock) active during this event. See *Modifiers*.

`on_mouse_scroll`(*x: int*, *y: int*, *scroll_x: int*, *scroll_y: int*)

User moves the scroll wheel.

Parameters

- **`x`** – x position of mouse
- **`y`** – y position of mouse
- **`scroll_x`** – ammout of x pixels scrolled since last call
- **`scroll_y`** – ammout of y pixels scrolled since last call

`on_resize`(*width: int*, *height: int*)

Called when the window is resized while this view is active. `on_resize()` is also called separately. By default this method does nothing and can be overridden to handle resize logic.

`on_show`()

Deprecated. Use `on_show_view()` instead.

`on_show_view`()

Called once when the view is shown.

See also:

`on_hide_view()`

`on_update`(*delta_time: float*)

To be overridden

`has_sections`

Return if the View has sections

`section_manager`

lazy instantiation of the section manager `class arcade.Window`(*width: int = 800, height: int = 600, title: str | None = 'Arcade Window', fullscreen: bool = False, resizable: bool = False, update_rate: float = 0.016666666666666666, antialiasing: bool = True, gl_version: Tuple[int, int] = (3, 3), screen: Screen | None = None, style: str | None = None, visible: bool = True, vsync: bool = False, gc_mode: str = 'context_gc', center_window: bool = False, samples: int = 4, enable_polling: bool = True, gl_api: str = 'gl', draw_rate: float = 0.016666666666666666*)

Bases: `BaseWindow`

The Window class forms the basis of most advanced games that use Arcade. It represents a window on the screen, and manages events.

Parameters

- **`width`** – Window width
- **`height`** – Window height
- **`title`** – Title (appears in title bar)
- **`fullscreen`** – Should this be full screen?
- **`resizable`** – Can the user resize the window?
- **`update_rate`** – How frequently to run the on_update event.
- **`draw_rate`** – How frequently to run the on_draw event. (this is the FPS limit)
- **`antialiasing`** – Should OpenGL's anti-aliasing be enabled?
- **`gl_version`** – What OpenGL version to request. This is `(3, 3)` by default and can be overridden when using more advanced OpenGL features.
- **`screen`** – Pass a pyglet `Screen` to request the window be placed on it. See pyglet's window size & position guide to learn more.
- **`style`** – Request a non-default window style, such as borderless. Some styles only work in certain situations. See pyglet's guide to window style to learn more.
- **`visible`** – Should the window be visible immediately
- **`vsync`** – Wait for vertical screen refresh before swapping buffer This can make animations and movement look smoother.
- **`gc_mode`** – Decides how OpenGL objects should be garbage collected ("context_gc" (default) or "auto")
- **`center_window`** – If true, will center the window.
- **`samples`** – Number of samples used in antialiasing (default 4). Usually this is 2, 4, 8 or 16.
- **`enable_polling`** – Enabled input polling capability. This makes the `keyboard` and `mouse` attributes available for use.

`activate`()

Activate this window.

`center_window`() → None

Center the window on the screen.

`clear`(*color: Tuple[int, int, int, int] | Tuple[float, float, float, float] | None = None, normalized: bool = False, viewport: Tuple[int, int, int, int] | None = None*)

Clears the window with the configured background color set through *`arcade.Window.background_color`*.

Parameters

- **color** – (Optional) override the current background color with one of the following:

  1. A *Color* instance
  2. A 4-length RGBA `tuple` of byte values (0 to 255)
  3. A 4-length RGBA `tuple` of normalized floats (0.0 to 1.0)

- **normalized** – If the color format is normalized (0.0 -> 1.0) or byte values
- **viewport** (*Tuple[int, int, int, int]*) – The viewport range to clear

close()

Close the Window.

dispatch_events()

Dispatch events

flip()

Window framebuffers normally have a back and front buffer. This method makes the back buffer visible and hides the front buffer. A frame is rendered into the back buffer, so this method displays the frame we currently worked on.

This method also garbage collect OpenGL resources before swapping the buffers.

get_location() → Tuple[int, int]

Return the X/Y coordinates of the window

Returns
x, y of window location

get_size() → Tuple[int, int]

Get the size of the window.

Returns
(width, height)

get_system_mouse_cursor(*name*)

Get the system mouse cursor

get_viewport() → Tuple[float, float, float, float]

Get the viewport. (What coordinates we can see.)

hide_view()

Hide the currently active view (if any) returning us back to `on_draw` and `on_update` functions in the window.

This is not necessary to call if you are switching views. Simply call `show_view` again.

maximize()

Maximize the window.

minimize()

Minimize the window.

on_draw()

Override this function to add your custom drawing code.

on_key_press(*symbol: int*, *modifiers: int*)

Called once when a key gets pushed down.

Override this function to add key press functionality.

Tip: If you want the length of key presses to affect gameplay, you also need to override `on_key_release()`.

Parameters

- **symbol** – Key that was just pushed down
- **modifiers** – Bitwise 'and' of all modifiers (shift, ctrl, num lock) active during this event. See *Modifiers*.

on_key_release(*symbol: int*, *modifiers: int*)

Called once when a key gets released.

Override this function to add key release functionality.

Situations that require handling key releases include:

- Rythm games where a note must be held for a certain amount of time
- 'Charging up' actions that change strength depending on how long a key was pressed
- Showing which keys are currently pressed down

Parameters

- **symbol** – Key that was just released
- **modifiers** – Bitwise 'and' of all modifiers (shift, ctrl, num lock) active during this event. See *Modifiers*.

on_mouse_drag(*x: int*, *y: int*, *dx: int*, *dy: int*, *buttons: int*, *modifiers: int*)

Called repeatedly while the mouse moves with a button down.

Override this function to handle dragging.

Parameters

- **x** – x position of mouse
- **y** – y position of mouse
- **dx** – Change in x since the last time this method was called
- **dy** – Change in y since the last time this method was called
- **buttons** – Which button is pressed
- **modifiers** – Bitwise 'and' of all modifiers (shift, ctrl, num lock) active during this event. See *Modifiers*.

on_mouse_enter(*x: int*, *y: int*)

Called once whenever the mouse enters the window area on screen.

This event will not be triggered if the mouse is currently being dragged.

Parameters

- **x** –
- **y** –

`on_mouse_leave`(*x: int*, *y: int*)

Called once whenever the mouse leaves the window area on screen.

This event will not be triggered if the mouse is currently being dragged. Note that the coordinates of the mouse pointer will be outside of the window rectangle.

Parameters

- **x** –
- **y** –

`on_mouse_motion`(*x: int*, *y: int*, *dx: int*, *dy: int*)

Called repeatedly while the mouse is moving over the window.

Override this function to respond to changes in mouse position.

Parameters

- **x** – x position of mouse within the window in pixels
- **y** – y position of mouse within the window in pixels
- **dx** – Change in x since the last time this method was called
- **dy** – Change in y since the last time this method was called

`on_mouse_press`(*x: int*, *y: int*, *button: int*, *modifiers: int*)

Called once whenever a mouse button gets pressed down.

Override this function to handle mouse clicks. For an example of how to do this, see arcade's built-in aiming and shooting bullets demo.

See also:

`on_mouse_release()`

Parameters

- **x** – x position of the mouse
- **y** – y position of the mouse
- **button** – What button was pressed. This will always be one of the following:
  - `arcade.MOUSE_BUTTON_LEFT`
  - `arcade.MOUSE_BUTTON_RIGHT`
  - `arcade.MOUSE_BUTTON_MIDDLE`
- **modifiers** – Bitwise 'and' of all modifiers (shift, ctrl, num lock) active during this event. See *Modifiers*.

`on_mouse_release`(*x: int*, *y: int*, *button: int*, *modifiers: int*)

Called once whenever a mouse button gets released.

Override this function to respond to mouse button releases. This may be useful when you want to use the duration of a mouse click to affect gameplay.

Parameters

- **x** – x position of mouse
- **y** – y position of mouse
- **button** – What button was hit. One of: arcade.MOUSE_BUTTON_LEFT, arcade.MOUSE_BUTTON_RIGHT, arcade.MOUSE_BUTTON_MIDDLE
- **modifiers** – Bitwise 'and' of all modifiers (shift, ctrl, num lock) active during this event. See *Modifiers*.

`on_mouse_scroll`(*x: int*, *y: int*, *scroll_x: int*, *scroll_y: int*)

Called repeatedly while a mouse scroll wheel moves.

Override this function to respond to scroll events. The scroll arguments may be positive or negative to indicate direction, but the units are unstandardized. How many scroll steps you recieve may vary wildly between computers depending a number of factors, including system settings and the input devices used (i.e. mouse scrollwheel, touchpad, etc).

Warning: Not all users can scroll easily!

Only some input devices support horizontal scrolling. Standard vertical scrolling is common, but some laptop touchpads are hard to use.

This means you should be careful about how you use scrolling. Consider making it optional to maximize the number of people who can play your game!

Parameters

- **x** – x position of mouse
- **y** – y position of mouse
- **scroll_x** – number of steps scrolled horizontally since the last call of this function
- **scroll_y** – number of steps scrolled vertically since the last call of this function

`on_resize`(*width: int*, *height: int*)

Override this function to add custom code to be called any time the window is resized. The main responsibility of this method is updating the projection and the viewport.

If you are not changing the default behavior when overriding, make sure you call the parent's `on_resize` first:

```python
def on_resize(self, width: int, height: int):
    super().on_resize(width, height)
    # Add extra resize logic here
```

Parameters

- **width** – New width
- **height** – New height

`on_update`(*delta_time: float*)

Move everything. Perform collision checks. Do all the game logic here.

Parameters

delta_time – Time interval since the last time the function was called.

run() → None

Run the main loop. After the window has been set up, and the event hooks are in place, this is usually one of the last commands on the main program. This is a blocking function starting pyglet's event loop meaning it will start to dispatch events such as `on_draw` and `on_update`.

set_caption(*caption*)

Set the caption for the window.

set_draw_rate(*rate: float*)

Set how often the on_draw function should be run. For example, set.set_draw_rate(1 / 60) will set the draw rate to 60 frames per second.

set_exclusive_keyboard(*exclusive=True*)

Capture all keyboard input.

set_exclusive_mouse(*exclusive=True*)

Capture the mouse.

set_fullscreen(*fullscreen: bool = True*, *screen: Window | None = None*, *mode: ScreenMode | None = None*, *width: float | None = None*, *height: float | None = None*)

Set if we are full screen or not.

Parameters

- **fullscreen** –
- **screen** – Which screen should we display on? See `get_screens()`
- **mode** – The screen will be switched to the given mode. The mode must have been obtained by enumerating *Screen.get_modes*. If None, an appropriate mode will be selected from the given *width* and *height*.
- **width** –
- **height** –

set_location(*x*, *y*)

Set location of the window.

set_max_size(*width: int*, *height: int*)

Wrap the Pyglet window call to set maximum size

Parameters

- **width** – width in pixels.
- **height** – height in pixels.

Raises ValueError

set_maximum_size(*width*, *height*)

Set largest window size.

set_min_size(*width: int*, *height: int*)

Wrap the Pyglet window call to set minimum size

Parameters

- **width** – width in pixels.
- **height** – height in pixels.

`set_minimum_size`(*width: int*, *height: int*)

Set smallest window size.

`set_mouse_platform_visible`(*platform_visible=None*)

Warning: You are probably looking for *`set_mouse_visible()`*!

This method was implemented to prevent PyCharm from displaying linter warnings. Most users will never need to set platform-specific visibility as the defaults from pyglet will usually handle their needs automatically.

For more information on what this means, see the documentation for `pyglet.window.Window.set_mouse_platform_visible()`.

`set_mouse_visible`(*visible: bool = True*)

Set whether to show the system's cursor while over the window

By default, the system mouse cursor is visible whenever the mouse is over the window. To hide the cursor, pass `False` to this function. Pass `True` to make the cursor visible again.

The window will continue receiving mouse events while the cursor is hidden, including movements and clicks. This means that functions like *`on_mouse_motion()`* and t'*`on_mouse_press()`* will continue to work normally.

You can use this behavior to visually replace the system mouse cursor with whatever you want. One example is a game character that is always at the most recent mouse position in the window.

Note: Advanced users can try using system cursor state icons

It may be possible to use system icons representing cursor interaction states such as hourglasses or resize arrows by using features `arcade.Window` inherits from the underlying pyglet window class. See the pyglet overview on cursors for more information.

Parameters
`visible` – Whether to hide the system mouse cursor

`set_size`(*width: int*, *height: int*)

Ignore the resizable flag and set the size

Parameters

- **`width`** –
- **`height`** –

`set_update_rate`(*rate: float*)

Set how often the on_update function should be dispatched. For example, self.set_update_rate(1 / 60) will set the update rate to 60 times per second.

Parameters
`rate` – Update frequency in seconds

`set_viewport`(*left: float*, *right: float*, *bottom: float*, *top: float*)

Set the viewport. (What coordinates we can see. Used to scale and/or scroll the screen).

See *`arcade.set_viewport()`* for more detailed information.

Parameters

- **left** –
- **right** –
- **bottom** –
- **top** –

set_visible(*visible: bool = True*)

Set if the window is visible or not. Normally, a program's window is visible.

Parameters
visible –

set_vsync(*vsync: bool*)

Set if we sync our draws to the monitors vertical sync rate.

show_view(*new_view: View*)

Select the view to show in the next frame. This is not a blocking call showing the view. Your code will continue to run after this call and the view will appear in the next dispatch of `on_update/on_draw``.

Calling this function is the same as setting the *arcade.Window.current_view* attribute.

Parameters
new_view – View to show

switch_to()

Switch the this window.

test(*frames: int = 10*)

Used by unit test cases. Runs the event loop a few times and stops.

Parameters
frames –

use()

Bind the window's framebuffer for rendering commands

background_color

Get or set the background color for this window. This affects what color the window will contain when *clear()* is called.

Examples:

```
# Use Arcade's built in Color values
window.background_color = arcade.color.AMAZON

# Set the background color with a custom Color instance
MY_RED = arcade.types.Color(255, 0, 0)
window.background_color = MY_RED

# Set the backgrund color directly from an RGBA tuple
window.background_color = 255, 0, 0, 255

# (Discouraged)
# Set the background color directly from an RGB tuple
# RGB tuples will assume 255 as the opacity / alpha value
window.background_color = 255, 0, 0
```

Type
Color

ctx

The OpenGL context for this window.

Type
arcade.ArcadeContext

current_view

This property returns the current view being shown. To set a different view, call the *arcade.Window.show_view()* method.

headless

If this is a headless window

Type
bool

arcade.**get_screens**() → List

Return a list of screens. So for a two-monitor setup, this should return a list of two screens. Can be used with arcade.Window to select which window we full-screen on.

Returns
List of screens, one for each monitor.

arcade.**open_window**(*width: int*, *height: int*, *window_title: str | None = None*, *resizable: bool = False*, *antialiasing: bool = True*) → *Window*

This function opens a window. For ease-of-use we assume there will only be one window, and the programmer does not need to keep a handle to the window. This isn't the best architecture, because the window handle is stored in a global, but it makes things easier for programmers if they don't have to track a window pointer.

Parameters

- **width** – Width of the window.
- **height** – Height of the window.
- **window_title** – Title of the window.
- **resizable** – Whether the user can resize the window.
- **antialiasing** – Smooth the graphics?

Returns
Handle to window

arcade.**close_window**() → None

Closes the current window, and then runs garbage collection. The garbage collection is necessary to prevent crashing when opening/closing windows rapidly (usually during unit tests).

arcade.**exit**() → None

Exits the application.

arcade.**finish_render**()

Swap buffers and displays what has been drawn.

Warning: If you are extending the *Window* class, this function should not be called. The event loop will automatically swap the window framebuffer for you after `on_draw`.

`arcade.get_display_size`(*screen_id: int = 0*) → Tuple[int, int]

Return the width and height of a monitor.

The size of the primary monitor is returned by default.

Parameters
`screen_id` – The screen number

Returns
Tuple containing the width and height of the screen

`arcade.get_window`() → *Window*

Return a handle to the current window.

Returns
Handle to the current window.

`arcade.pause`(*seconds: float*) → None

Pause for the specified number of seconds. This is a convenience function that just calls time.sleep().

Warning: This is mostly used for unit tests and is not likely to be a good solution for pausing an application or game.

Parameters
`seconds` – Time interval to pause in seconds.

`arcade.run`()

Run the main loop. After the window has been set up, and the event hooks are in place, this is usually one of the last commands on the main program. This is a blocking function starting pyglet's event loop meaning it will start to dispatch events such as `on_draw` and `on_update`.

`arcade.schedule`(*function_pointer: Callable*, *interval: float*)

Schedule a function to be automatically called every `interval` seconds. The function/callable needs to take a delta time argument similar to `on_update`. This is a float representing the number of seconds since it was scheduled or called.

A function can be scheduled multiple times, but this is not recommended.

Warning: Scheduled functions should **always** be unscheduled using *`arcade.unschedule()`*. Having lingering scheduled functions will lead to crashes.

Example:

```python
def some_action(delta_time):
    print(delta_time)

# Call the function every second
arcade.schedule(some_action, 1)
# Unschedule
```

Parameters

- **`function_pointer`** – Pointer to the function to be called.
- **`interval`** – Interval to call the function (float or integer)

`arcade.schedule_once(function_pointer: Callable, delay: float)`

Schedule a function to be automatically called once after `delay` seconds. The function/callable needs to take a delta time argument similar to `on_update`. This is a float representing the number of seconds since it was scheduled or called.

Example:

```python
def some_action(delta_time):
    print(delta_time)

# Call the function once after 1 second
arcade.schedule_one(some_action, 1)
```

Parameters

- **`function_pointer`** – Pointer to the function to be called.
- **`delay`** – Delay in seconds

`arcade.set_background_color(color: Tuple[int, int, int, int]) → None`

Set the color `arcade.Window.clear()` will use when clearing the window. This only needs to be called when the background color changes.

Note: A shorter and faster way to set background color is using `arcade.Window.background_color`.

Examples:

```python
# Use Arcade's built in color values
arcade.set_background_color(arcade.color.AMAZON)

# Specify RGB value directly (red)
arcade.set_background_color((255, 0, 0))
```

Parameters

`RGBA255` – List of 3 or 4 values in RGB/RGBA format.

`arcade.set_viewport(left: float, right: float, bottom: float, top: float) → None`

This sets what coordinates the window will cover.

Tip: Beginners will want to use `Camera`. It provides easy to use support for common tasks such as screen shake and movement to a destination.

If you are making a game with complex control over the viewport, this function can help.

By default, the lower left coordinate will be `(0, 0)`, the top y coordinate will be the height of the window in pixels, and the right x coordinate will be the width of the window in pixels.

Warning: Be careful of fractional or non-multiple values!

It is recommended to only set the viewport to integer values that line up with the pixels on the screen. Otherwise, tiled pixel art may not line up well during render, creating rectangle artifacts.

Note: *Window.on_resize* calls `set_viewport` by default. If you want to set your own custom viewport during the game, you may need to override the *Window.on_resize* method.

Note: For more advanced users

This functions sets the orthogonal projection used by shapes and sprites. It also updates the viewport to match the current screen resolution. `window.ctx.projection_2d` (*projection_2d()*) and `window.ctx.viewport` (*viewport()*) can be used to set viewport and projection separately.

Parameters

- **left** – Left-most (smallest) x value.
- **right** – Right-most (largest) x value.
- **bottom** – Bottom (smallest) y value.
- **top** – Top (largest) y value.

arcade.**set_window**(*window: 'Window' | None*) → None

Set a handle to the current window.

Parameters
window – Handle to the current window.

arcade.**start_render**() → None

Clears the window.

More practical alternatives to this function is *arcade.Window.clear()* or *arcade.View.clear()*.

arcade.**unschedule**(*function_pointer: Callable*)

Unschedule a function being automatically called.

Example:

```python
def some_action(delta_time):
    print(delta_time)

arcade.schedule(some_action, 1)
arcade.unschedule(some_action)
```

Parameters
function_pointer – Pointer to the function to be unscheduled.

class arcade.**Section**(*left: int, bottom: int, width: int, height: int, *, name: str | None = None, accept_keyboard_keys: bool | Iterable = True, accept_mouse_events: bool | Iterable = True, prevent_dispatch: Iterable | None = None, prevent_dispatch_view: Iterable | None = None, local_mouse_coordinates: bool = False, enabled: bool = True, modal: bool = False, draw_order: int = 1*)

Bases:

A Section represents a rectangular portion of the viewport Events are dispatched to the section based on it's position on the screen.

Parameters

- **left** – the left position of this section
- **bottom** – the bottom position of this section
- **width** – the width of this section
- **height** – the height of this section
- **name** – the name of this section
- **accept_keyboard_keys** (*Union[bool, Iterable]*) – whether or not this section captures keyboard keys through. keyboard events. If the param is an iterable means the keyboard keys that are captured in press/release events: for example: [arcade.key.UP, arcade.key.DOWN] will only capture this two keys
- **accept_mouse_events** (*Union[bool, Iterable]*) – whether or not this section captures mouse events. If the param is an iterable means the mouse events that are captured. for example: ['on_mouse_press', 'on_mouse_release'] will only capture this two events.
- **prevent_dispatch** – a list of event names that will not be dispatched to subsequent sections. You can pass None (default) or {True} to prevent the dispatch of all events.
- **prevent_dispatch_view** – a list of event names that will not be dispatched to the view. You can pass None (default) or {True} to prevent the dispatch of all events to the view.
- **local_mouse_coordinates** – if True the section mouse events will receive x, y coordinates section related to the section dimensions and position (not related to the screen)
- **enabled** – if False the section will not capture any events
- **modal** – if True the section will be a modal section: will prevent updates and event captures on other sections. Will also draw last (on top) but capture events first.
- **draw_order** – The order this section will have when on_draw is called. The lower the number the earlier this will get draw. This can be different from the event capture order or the on_update order which is defined by the insertion order.

get_xy_screen_relative(*section_x: int*, *section_y: int*)

Returns screen coordinates from section coordinates

get_xy_section_relative(*screen_x: int*, *screen_y: int*)

returns section coordinates from screen coordinates

mouse_is_on_top(*x: int*, *y: int*) → bool

Check if the current mouse position is on top of this section

on_draw()

on_hide_section()

on_key_press(*symbol: int*, *modifiers: int*)

on_key_release(*_symbol: int*, *_modifiers: int*)

on_mouse_drag(*x: int*, *y: int*, *dx: int*, *dy: int*, *_buttons: int*, *_modifiers: int*)

on_mouse_enter(*x: int*, *y: int*)

on_mouse_leave(*x: int*, *y: int*)

on_mouse_motion(*x: int*, *y: int*, *dx: int*, *dy: int*)

on_mouse_press(*x: int*, *y: int*, *button: int*, *modifiers: int*)

on_mouse_release(*x: int*, *y: int*, *button: int*, *modifiers: int*)

on_mouse_scroll(*x: int*, *y: int*, *scroll_x: int*, *scroll_y: int*)

on_resize(*width: int*, *height: int*)

on_show_section()

on_update(*delta_time: float*)

overlaps_with(*section: Section*) → bool

Checks if this section overlaps with another section

should_receive_mouse_event(*x: int*, *y: int*) → bool

Check if the current section should receive a mouse event at a given position

bottom

The bottom edge of this section

draw_order

Returns the draw order state The lower the number the earlier this section will get draw

enabled

Enables or disables this section

height

The height of this section

left

Left edge of this section

modal

Returns the modal state (Prevent the following sections from receiving input events and updating)

right

Right edge of this section

section_manager

Returns the section manager

top

Top edge of this section

view

The view this section is set on

width

The width of this section

window

The view window

class arcade.**SectionManager**(*view: View*)

Bases:

This manages the different Sections a View has. Actions such as dispatching the events to the correct Section, draw order, etc.

`add_section`(*section: Section*, *at_index: int | None = None*, *at_draw_order: int | None = None*) → None

Adds a section to this Section Manager Will trigger section.on_show_section if section is enabled

Parameters

- **`section`** – the section to add to this section manager
- **`at_index`** – inserts the section at that index for event capture and update events. If None at the end
- **`at_draw_order`** – inserts the section in a specific draw order. Overwrites section.draw_order

`clear_sections`() → None

Removes all sections and calls on_hide_section for each one if enabled

`disable`() → None

Disable all sections Disabling a section will trigger section.on_hide_section

`disable_all_keyboard_events`() → None

Removes the keyboard event handling from all sections

`dispatch_keyboard_event`(*event: str*, **args*, ***kwargs*) → bool | None

Generic method to dispatch keyboard events to the correct sections

Parameters

- **`event`** – the keyboard event name to dispatch
- **`args`** – any other position arguments that should be deliverd to the dispatched event
- **`kwargs`** – any other keyword arguments that should be delivered to the dispatched event

Returns

EVENT_HANDLED or EVENT_UNHANDLED, or whatever the dispatched method returns

`dispatch_mouse_enter_leave_events`(*event_origin: str*, *x: int*, *y: int*, **args*, ***kwargs*) → bool | None

This helper method will dispatch mouse enter / leave events to sections based on 'on_mouse_motion' and 'on_mouse_drag' events. Will also dispatch the event (event_origin) that called this method

Parameters

- **`event_origin`** – the mouse event name that called this method. This event will be called here.
- **`x`** – the x axis coordinate
- **`y`** – the y axis coordinate
- **`args`** – any other position arguments that should be deliverd to the dispatched event
- **`kwargs`** – any other keyword arguments that should be delivered to the dispatched event

Returns

EVENT_HANDLED or EVENT_UNHANDLED, or whatever the dispatched method returns

`dispatch_mouse_event`(*event: str*, *x: int*, *y: int*, **args*, *current_section: Section | None = None*, ***kwargs*) → bool | None

Generic method to dispatch mouse events to the correct Sections

Parameters

- **`event`** – the mouse event name to dispatch
- **`x`** – the x axis coordinate

- **y** – the y axis coordinate
- **args** – any other position arguments that should be deliverd to the dispatched event
- **current_section** – the section this mouse event should be delivered to. If None, will retrive all sections that should recieve this event based on x, y coordinates
- **kwargs** – any other keyword arguments that should be delivered to the dispatched event

Returns
EVENT_HANDLED or EVENT_UNHANDLED, or whatever the dispatched method returns

enable() → None

Enables all sections Enabling a section will trigger section.on_show_section

get_first_section(*x: int*, *y: int*, *, *event_capture: bool = True*) → *Section* | None

Returns the first section based on x,y position

Parameters

- **x** – the x axis coordinate
- **y** – the y axis coordinate
- **event_capture** – True will use event capture dimensions, False will use section draw size

Returns
a section if match the params otherwise None

get_section_by_name(*name: str*) → *Section* | None

Returns the first section with the given name :param name: the name of the section you want :return: the first section with the provided name. None otherwise

get_sections(*x: int*, *y: int*, *, *event_capture: bool = True*) → Generator[*Section*, None, None]

Returns a list of sections based on x,y position

Parameters

- **x** – the x axis coordinate
- **y** – the y axis coordinate
- **event_capture** – True will use event capture dimensions, False will use section draw size

Returns
a generator with the sections that match the params

on_draw() → None

Called on each event loop to draw It automatically calls camera.use() for each section that has a camera and resets the camera effects by calling the default SectionManager camera afterwards if needed. The SectionManager camera defaults to a camera that has the viewport and projection for the whole screen

on_hide_view() → None

Called when the view is hide The View.on_hide_view is called before this by the Window.hide_view method

on_key_press(**args*, ***kwargs*) → bool | None

Triggers the on_key_press event on the appropiate sections or view

Parameters

- **args** – any other position arguments that should be deliverd to the dispatched event
- **kwargs** – any other keyword arguments that should be delivered to the dispatched event

Returns
EVENT_HANDLED or EVENT_UNHANDLED, or whatever the dispatched method returns

`on_key_release`(**args*, ***kwargs*) → bool | None

Triggers the on_key_release event on the appropiate sections or view

Parameters

- **args** – any other position arguments that should be deliverd to the dispatched event
- **kwargs** – any other keyword arguments that should be delivered to the dispatched event

Returns
EVENT_HANDLED or EVENT_UNHANDLED, or whatever the dispatched method returns

`on_mouse_drag`(*x: int*, *y: int*, **args*, ***kwargs*) → bool | None

This method dispatches the on_mouse_drag and also calculates if on_mouse_enter/leave should be fired

Parameters

- **x** – the x axis coordinate
- **y** – the y axis coordinate
- **args** – any other position arguments that should be deliverd to the dispatched event
- **kwargs** – any other keyword arguments that should be delivered to the dispatched event

Returns
EVENT_HANDLED or EVENT_UNHANDLED, or whatever the dispatched method returns

`on_mouse_enter`(*x: int*, *y: int*, **args*, ***kwargs*) → bool | None

Triggered when the mouse enters the window space Will trigger on_mouse_enter on the appropiate sections or view

Parameters

- **x** – the x axis coordinate
- **y** – the y axis coordinate
- **args** – any other position arguments that should be deliverd to the dispatched event
- **kwargs** – any other keyword arguments that should be delivered to the dispatched event

Returns
EVENT_HANDLED or EVENT_UNHANDLED, or whatever the dispatched method returns

`on_mouse_leave`(*x: int*, *y: int*, **args*, ***kwargs*) → bool | None

Triggered when the mouse leaves the window space Will trigger on_mouse_leave on the appropiate sections or view

Parameters

- **x** – the x axis coordinate
- **y** – the y axis coordinate
- **args** – any other position arguments that should be deliverd to the dispatched event
- **kwargs** – any other keyword arguments that should be delivered to the dispatched event

Returns
EVENT_HANDLED or EVENT_UNHANDLED, or whatever the dispatched method returns

on_mouse_motion(*x: int*, *y: int*, **args*, ***kwargs*) → bool | None

This method dispatches the on_mouse_motion and also calculates if on_mouse_enter/leave should be fired

Parameters

- **x** – the x axis coordinate
- **y** – the y axis coordinate
- **args** – any other position arguments that should be deliverd to the dispatched event
- **kwargs** – any other keyword arguments that should be delivered to the dispatched event

Returns

EVENT_HANDLED or EVENT_UNHANDLED, or whatever the dispatched method returns

on_mouse_press(*x: int*, *y: int*, **args*, ***kwargs*) → bool | None

Triggers the on_mouse_press event on the appropiate sections or view

Parameters

- **x** – the x axis coordinate
- **y** – the y axis coordinate
- **args** – any other position arguments that should be deliverd to the dispatched event
- **kwargs** – any other keyword arguments that should be delivered to the dispatched event

Returns

EVENT_HANDLED or EVENT_UNHANDLED, or whatever the dispatched method returns

on_mouse_release(*x: int*, *y: int*, **args*, ***kwargs*) → bool | None

Triggers the on_mouse_release event on the appropiate sections or view

Parameters

- **x** – the x axis coordinate
- **y** – the y axis coordinate
- **args** – any other position arguments that should be deliverd to the dispatched event
- **kwargs** – any other keyword arguments that should be delivered to the dispatched event

Returns

EVENT_HANDLED or EVENT_UNHANDLED, or whatever the dispatched method returns

on_mouse_scroll(*x: int*, *y: int*, **args*, ***kwargs*) → bool | None

Triggers the on_mouse_scroll event on the appropiate sections or view

Parameters

- **x** – the x axis coordinate
- **y** – the y axis coordinate
- **args** – any other position arguments that should be deliverd to the dispatched event
- **kwargs** – any other keyword arguments that should be delivered to the dispatched event

Returns

EVENT_HANDLED or EVENT_UNHANDLED, or whatever the dispatched method returns `on_resize`(*width: int*, *height: int*) → None

Called when the window is resized.

Parameters

- **width** – the new width of the screen
- **height** – the new height of the screen

`on_show_view`() → None

Called when the view is shown The View.on_show_view is called before this by the Window.show_view method

`on_update`(*delta_time: float*) → None

Called on each event loop.

Parameters

delta_time – the delta time since this method was called last time

`remove_section`(*section: Section*) → None

Removes a section from this section manager

Parameters

section – the section to remove

`sort_section_event_order`() → None

This will sort sections on event capture order (and update) based on insertion order and section.modal

`sort_sections_draw_order`() → None

This will sort sections on draw order based on section.draw_order and section.modal

`has_sections`

Returns true if this section manager has sections

`is_current_view`

Returns if this section manager view is the current on the view window a.k.a.: is the view that is currently being shown

`sections`

Property that returns the list of sections

33.21 Sound

`class arcade.Sound`(*file_name: str | Path*, *streaming: bool = False*)

Bases:

This class represents a sound you can play.

`get_length`() → float

Get length of audio in seconds

`get_stream_position`(*player: Player*) → float

Return where we are in the stream. This will reset back to zero when it is done playing.

Parameters

player – Player returned from `play_sound()`.

`get_volume`(*player: Player*) → float

Get the current volume.

Parameters
`player` – Player returned from `play_sound()`.

Returns
A float, 0 for volume off, 1 for full volume.

`is_complete`(*player: Player*) → bool

Return true if the sound is done playing.

`is_playing`(*player: Player*) → bool

Return if the sound is currently playing or not

Parameters
`player` – Player returned from `play_sound()`.

Returns
A boolean, `True` if the sound is playing.

`play`(*volume: float = 1.0*, *pan: float = 0.0*, *loop: bool = False*, *speed: float = 1.0*) → Player

Play the sound.

Parameters

- `volume` – Volume, from 0=quiet to 1=loud
- `pan` – Pan, from -1=left to 0=centered to 1=right
- `loop` – Loop, false to play once, true to loop continuously
- `speed` – Change the speed of the sound which also changes pitch, default 1.0

`set_volume`(*volume*, *player: Player*) → None

Set the volume of a sound as it is playing.

Parameters

- `volume` – Floating point volume. 0 is silent, 1 is full.
- `player` – Player returned from `play_sound()`.

`stop`(*player: Player*) → None

Stop a currently playing sound.

`arcade.load_sound`(*path: str | Path*, *streaming: bool = False*) → *Sound* | None

Load a sound.

Parameters

- `path` – Name of the sound file to load.
- `streaming` – Boolean for determining if we stream the sound or load it all into memory. Set to `True` for long sounds to save memory, `False` for short sounds to speed playback.

Returns
Sound object which can be used by the `play_sound()` function.

`arcade.play_sound`(*sound: Sound*, *volume: float = 1.0*, *pan: float = 0.0*, *loop: bool = False*, *speed: float = 1.0*) → Player | None

Play a sound.

Parameters

- **sound** – Sound loaded by *load_sound()*. Do NOT use a string here for the filename.
- **volume** – Volume, from 0=quiet to 1=loud
- **pan** – Pan, from -1=left to 0=centered to 1=right
- **loop** – Should we loop the sound over and over?
- **speed** – Change the speed of the sound which also changes pitch, default 1.0

arcade.**stop_sound**(*player: Player*)

Stop a sound that is currently playing.

Parameters

player – Player returned from *play_sound()*.

33.22 Pathfinding

class arcade.**AStarBarrierList**(*moving_sprite: Sprite, blocking_sprites: SpriteList, grid_size: int, left: int, right: int, bottom: int, top: int*)

Bases:

Class that manages a list of barriers that can be encountered during A* path finding.

Parameters

- **moving_sprite** – Sprite that will be moving
- **blocking_sprites** – Sprites that can block movement
- **grid_size** – Size of the grid, in pixels
- **left** – Left border of playing field
- **right** – Right border of playing field
- **bottom** – Bottom of playing field
- **top** – Top of playing field
- **barrier_list** – SpriteList of barriers to use in _AStarSearch, None if not recalculated

recalculate()

Recalculate blocking sprites.

arcade.**astar_calculate_path**(*start_point: Tuple[float, float], end_point: Tuple[float, float], astar_barrier_list: AStarBarrierList, diagonal_movement: bool = True*) → List[Tuple[float, float]] | None

Calculates the path using AStarSearch Algorithm and returns the path

Parameters

- **start_point** – Where it starts
- **end_point** – Where it ends
- **astar_barrier_list** – AStarBarrierList with the boundries to use in the AStarSearch Algorithm
- **diagonal_movement** – Whether of not to use diagonals in the AStarSearch Algorithm

Returns

List of points(the path), or None if no path is found `arcade.`**`has_line_of_sight`**`(`*observer: Tuple[float, float], target: Tuple[float, float], walls:* SpriteList, *max_distance: float = inf, check_resolution: int = 2*`)` → bool

Determine if we have line of sight between two points.

Parameters

- **`observer`** – Start position
- **`target`** – End position position
- **`walls`** – List of all blocking sprites
- **`max_distance`** – Max distance point 1 can see
- **`check_resolution`** – Check every x pixels for a sprite. Trade-off between accuracy and speed.

Warning: Try to make sure spatial hashing is enabled on `walls`!

If spatial hashing is not enabled, this function may run very slowly!

Returns

Whether or not oberver to target is blocked by any wall in walls

33.23 Isometric Map Support (incomplete)

`arcade.isometric.`**`create_isometric_grid_lines`**`(`*width: int, height: int, tile_width: int, tile_height: int, color: Tuple[int, int, int, int], line_width: int*`)` → *ShapeElementList*

`arcade.isometric.`**`isometric_grid_to_screen`**`(`*tile_x: int, tile_y: int, width: int, height: int, tile_width: int, tile_height: int*`)` → Tuple[int, int]

`arcade.isometric.`**`screen_to_isometric_grid`**`(`*screen_x: int, screen_y: int, width: int, height: int, tile_width: int, tile_height: int*`)` → Tuple[int, int]

33.24 Earclip

`arcade.earclip.`**`earclip`**`(`*polygon: Sequence[Tuple[float, float]]*`)` → List[Tuple[Tuple[float, float], Tuple[float, float], Tuple[float, float]]]

Simple earclipping algorithm for a given polygon p. polygon is expected to be an array of 2-tuples of the cartesian points of the polygon For a polygon with n points it will return n-2 triangles. The triangles are returned as an array of 3-tuples where each item in the tuple is a 2-tuple of the cartesian point.

Implementation Reference:

- https://www.geometrictools.com/Documentation/TriangulationByEarClipping.pdf

33.25 Easing

class `arcade.easing.`**`EasingData`**(*start_period: float*, *cur_period: float*, *end_period: float*, *start_value: float*, *end_value: float*, *ease_function: Callable*)

Bases:

Data class for holding information about easing.

`reset`()

`cur_period:` `float`

`ease_function:` `Callable`

`end_period:` `float`

`end_value:` `float`

`start_period:` `float`

`start_value:` `float`

`arcade.easing.`**`ease_angle`**(*start_angle: float*, *end_angle: float*, ***, *time=None*, *rate=None*, *ease_function: ~typing.Callable = <function linear>*)

Set up easing for angles.

`arcade.easing.`**`ease_angle_update`**(*easing_data:* EasingData, *delta_time: float*) → Tuple

Update angle easing.

`arcade.easing.`**`ease_in`**(*percent: float*) → float

Function for quadratic ease-in easing.

`arcade.easing.`**`ease_in_back`**(*percent: float*) → float

Function for ease_in easing which moves back before moving forward.

`arcade.easing.`**`ease_in_out`**(*percent: float*) → float

Function for quadratic easing in and out.

`arcade.easing.`**`ease_in_out_sin`**(*percent: float*) → float

Function for easing in and out using a sin wave

`arcade.easing.`**`ease_in_sin`**(*percent: float*) → float

Function for ease_in easing using a sin wave

`arcade.easing.`**`ease_out`**(*percent: float*) → float

Function for quadratic ease-out easing.

`arcade.easing.`**`ease_out_back`**(*percent: float*) → float

Function for ease_out easing which moves back before moving forward.

`arcade.easing.`**`ease_out_bounce`**(*percent: float*) → float

Function for a bouncy ease-out easing.

`arcade.easing.`**`ease_out_elastic`**(*percent: float*) → float

Function for elastic ease-out easing.

`arcade.easing.`**`ease_out_sin`**(*percent: float*) → float

Function for ease_out easing using a sin wave `arcade.easing.`**`ease_position`**(*start_position*, *end_position*, *, *time=None*, *rate=None*, *ease_function=<function linear>*)

Get an easing position

`arcade.easing.`**`ease_update`**(*easing_data:* EasingData, *delta_time:* *float*) → Tuple

Update easing between two values/

`arcade.easing.`**`ease_value`**(*start_value: float*, *end_value: float*, *, *time=None*, *rate=None*, *ease_function=<function linear>*)

Get an easing value

`arcade.easing.`**`easing`**(*percent:* *float*, *easing_data:* EasingData) → float

Function for calculating return value for easing, given percent and easing data.

`arcade.easing.`**`linear`**(*percent:* *float*) → float

Function for linear easing.

`arcade.easing.`**`smoothstep`**(*percent:* *float*) → float

Function for smoothstep easing.

33.26 OpenGL Context

`class` `arcade.`**`ArcadeContext`**(*window:* *BaseWindow*, *gc_mode:* *str* = *'context_gc'*, *gl_api:* *str* = *'gl'*)

Bases: `Context`

An OpenGL context implementation for Arcade with added custom features. This context is normally accessed through `arcade.Window.ctx`.

Pyglet users can use the base Context class and extend that as they please.

This is part of the low level rendering API in arcade and is mainly for more advanced usage

Parameters

- **`window`** – The pyglet window
- **`gc_mode`** – The garbage collection mode for opengl objects. `auto` is just what we would expect in python while `context_gc` (default) requires you to call `Context.gc()`. The latter can be useful when using multiple threads when it's not clear what thread will gc the object.

`delattr`(self*name*, /)

Implement delattr(self, name).

`dir`(self)

Default dir() implementation.

self == value

Return self==value.

`format`(self*format_spec*, /)

Default object formatter.

self >= value

Return self>=value.

self > value

Return self>value.

hash(self)

Return hash(self).

self **<=** value

Return self<=value.

self **<** value

Return self<value.

self **!=** value

Return self!=value.

repr(self)

Return repr(self).

setattr(self*name*, *value*, */*)

Implement setattr(self, name, value).

sys.getsizeof(self)

Size of object in memory, in bytes.

str(self)

Return str(self).

classmethod activate(*ctx:* Context)

Mark a context as the currently active one.

> **Warning:** Never call this unless you know exactly what you are doing.

bind_window_block() → None

Binds the projection and view uniform buffer object. This should always be bound to index 0 so all shaders have access to them.

buffer(**, data: ByteString | memoryview | array | Array | None = None, reserve: int = 0, usage: str = 'static'*) → *Buffer*

Create an OpenGL Buffer object. The buffer will contain all zero-bytes if no data is supplied.

Examples:

```python
# Create 1024 byte buffer
ctx.buffer(reserve=1024)
# Create a buffer with 1000 float values using python's array.array
from array import array
ctx.buffer(data=array('f', [i for in in range(1000)]))
# Create a buffer with 1000 random 32 bit floats using numpy
self.ctx.buffer(data=np.random.random(1000).astype("f4"))
```

The `data` parameter can be anything that implements the Buffer Protocol.

This includes `bytes`, `bytearray`, `array.array`, and more. You may need to use typing workarounds for non-builtin types. See *Writing Raw Bytes to GL Buffers & Textures* for more information.

The `usage` parameter enables the GL implementation to make more intelligent decisions that may impact buffer object performance. It does not add any restrictions. If in doubt, skip this parameter and revisit when optimizing. The result are likely to be different between vendors/drivers or may not have any effect.

The available values mean the following:

```
stream
    The data contents will be modified once and used at most a few times.
static
    The data contents will be modified once and used many times.
dynamic
    The data contents will be modified repeatedly and used many times.
```

Parameters

- **data** – The buffer data. This can be a `bytes` instance or any any other object supporting the buffer protocol.
- **reserve** – The number of bytes to reserve
- **usage** – Buffer usage. 'static', 'dynamic' or 'stream'

compute_shader(**, source: str, common: Iterable[str] = ()*) → *ComputeShader*

Create a compute shader.

Parameters

- **source** – The glsl source
- **common** – Common / library source injected into compute shader

copy_framebuffer(*src:* Framebuffer, *dst:* Framebuffer)

Copies/blits a framebuffer to another one.

This operation has many restrictions to ensure it works across different platforms and drivers:

- The source and destination framebuffer must be the same size
- The formats of the attachments must be the same
- Only the source framebuffer can be multisampled
- Framebuffers cannot have integer attachments

Parameters

- **src** – The framebuffer to copy from
- **dst** – The framebuffer we copy to

depth_texture(*size: Tuple[int, int], *, data: ByteString | memoryview | array | Array | None = None*) → *Texture2D*

Create a 2D depth texture. Can be used as a depth attachment in a `Framebuffer`.

Parameters

- **size** (*Tuple[int, int]*) – The size of the texture
- **data** – The texture data (optional). Can be `bytes` or any object supporting the buffer protocol.

disable(**args*)

Disable one or more context flags:

```
# Single flag
ctx.disable(ctx.BLEND)
# Multiple flags
ctx.disable(ctx.DEPTH_TEST, ctx.CULL_FACE)
```

enable(**flags*)

Enables one or more context flags:

```
# Single flag
ctx.enable(ctx.BLEND)
# Multiple flags
ctx.enable(ctx.DEPTH_TEST, ctx.CULL_FACE)
```

enable_only(**args*)

Enable only some flags. This will disable all other flags. This is a simple way to ensure that context flag states are not lingering from other sections of your code base:

```
# Ensure all flags are disabled (enable no flags)
ctx.enable_only()
# Make sure only blending is enabled
ctx.enable_only(ctx.BLEND)
# Make sure only depth test and culling is enabled
ctx.enable_only(ctx.DEPTH_TEST, ctx.CULL_FACE)
```

enabled(**flags*)

Temporarily change enabled flags.

Flags that was enabled initially will stay enabled. Only new enabled flags will be reversed when exiting the context.

Example:

```
with ctx.enabled(ctx.BLEND, ctx.CULL_FACE):
    # Render something
```

enabled_only(**flags*)

Temporarily change enabled flags.

Only the supplied flags with be enabled in in the context. When exiting the context the old flags will be restored.

Example:

```
with ctx.enabled_only(ctx.BLEND, ctx.CULL_FACE):
    # Render something
```

finish() → None

Wait until all OpenGL rendering commands are completed.

This function will actually stall until all work is done and may have severe performance implications.

flush() → None

A suggestion to the driver to execute all the queued drawing calls even if the queue is not full yet. This is not a blocking call and only a suggestion. This can potentially be used for speedups when we don't have anything else to render.

framebuffer(**, color_attachments: Texture2D | List[Texture2D] | None = None, depth_attachment: Texture2D | None = None*) → *Framebuffer*

Create a Framebuffer.

Parameters

- **color_attachments** – List of textures we want to render into
- **depth_attachment** – Depth texture

gc() → int

Run garbage collection of OpenGL objects for this context. This is only needed when `gc_mode` is `context_gc`.

Returns

The number of resources destroyed

geometry(*content: Sequence[BufferDescription] | None = None, index_buffer: Buffer | None = None, mode: int | None = None, index_element_size: int = 4*)

Create a Geometry instance. This is Arcade's version of a vertex array adding a lot of convenience for the user. Geometry objects are fairly light. They are mainly responsible for automatically map buffer inputs to your shader(s) and provide various methods for rendering or processing this geometry,

The same geometry can be rendered with different programs as long as your shader is using one or more of the input attribute. This means geometry with positions and colors can be rendered with a program only using the positions. We will automatically map what is necessary and cache these mappings internally for performace.

In short, the geometry object is a light object that describes what buffers contains and automatically negotiate with shaders/programs. This is a very complex field in OpenGL so the Geometry object provides substantial time savings and greatly reduces the complexity of your code.

Geometry also provide rendering methods supporting the following:

- Rendering geometry with and without index buffer
- Rendering your geometry using instancing. Per instance buffers can be provided or the current instance can be looked up using `gl_InstanceID` in shaders.
- Running transform feedback shaders that writes to buffers instead the screen. This can write to one or multiple buffer.
- Render your geometry with indirect rendering. This means packing multiple meshes into the same buffer(s) and batch drawing them.

Examples:

```
# Single buffer geometry with a vec2 vertex position attribute
ctx.geometry([BufferDescription(buffer, '2f', ["in_vert"])], mode=ctx.TRIANGLES)

# Single interlaved buffer with two attributes. A vec2 position and vec2␣
↪velocity
ctx.geometry([
        BufferDescription(buffer, '2f 2f', ["in_vert", "in_velocity"])
    ],
    mode=ctx.POINTS,
)

# Geometry with index buffer
```

(continues on next page)

(continued from previous page)

```
ctx.geometry(
    [BufferDescription(buffer, '2f', ["in_vert"])],
    index_buffer=ibo,
    mode=ctx.TRIANGLES,
)

# Separate buffers
ctx.geometry([
        BufferDescription(buffer_pos, '2f', ["in_vert"])
        BufferDescription(buffer_vel, '2f', ["in_velocity"])
    ],
    mode=ctx.POINTS,
)

# Providing per-instance data for instancing
ctx.geometry([
        BufferDescription(buffer_pos, '2f', ["in_vert"])
        BufferDescription(buffer_instance_pos, '2f', ["in_offset"],␣
↪instanced=True)
    ],
    mode=ctx.POINTS,
)
```

Parameters

- **content** – List of *BufferDescription* (optional)
- **index_buffer** – Index/element buffer (optional)
- **mode** – The default draw mode (optional)
- **mode** – The default draw mode (optional)
- **index_element_size** – Byte size of a single index/element in the index buffer. In other words, the index buffer can be 8, 16 or 32 bit integers. Can be 1, 2 or 4 (8, 16 or 32 bit unsigned integer)

get_framebuffer_image(*fbo: Framebuffer*, *components: int = 4*, *flip: bool = True*) → Image

Shortcut method for reading data from a framebuffer and converting it to a PIL image.

Parameters

- **fbo** – Framebuffer to get image from
- **components** – Number of components to read
- **flip** – Flip the image upside down

is_enabled(*flag*) → bool

Check if a context flag is enabled

Type
bool

load_compute_shader(*path: str | Path*, *common: Iterable[str | Path] = ()*) → *ComputeShader*

Loads a compute shader from file. This methods supports resource handles.

Example:

```
ctx.load_compute_shader(":shader:compute/do_work.glsl")
```

Parameters

- **path** – Path to texture
- **common** – Common source injected into compute shader

load_program(**, vertex_shader: str | Path, fragment_shader: str | Path | None = None, geometry_shader: str | Path | None = None, tess_control_shader: str | Path | None = None, tess_evaluation_shader: str | Path | None = None, common: Iterable[str | Path] = (), defines: Dict[str, Any] | None = None, varyings: Sequence[str] | None = None, varyings_capture_mode: str = 'interleaved'*) → *Program*

Create a new program given a file names that contain the vertex shader and fragment shader. Note that fragment and geometry shader are optional for when transform shaders are loaded.

This method also supports the resource handles.

Example:

```
# The most common use case if having a vertex and fragment shader
program = window.ctx.load_program(
    vertex_shader="vert.glsl",
    fragment_shader="frag.glsl",
)
```

Parameters

- **vertex_shader** – path to vertex shader
- **fragment_shader** – path to fragment shader (optional)
- **geometry_shader** – path to geometry shader (optional)
- **tess_control_shader** – Tessellation Control Shader
- **tess_evaluation_shader** – Tessellation Evaluation Shader
- **common** – Common files to be included in all shaders
- **defines** – Substitute `#define` values in the source
- **varyings** – The name of the out attributes in a transform shader. This is normally not necessary since we auto detect them, but some more complex out structures we can't detect.
- **varyings_capture_mode** – The capture mode for transforms. `"interleaved"` means all out attribute will be written to a single buffer. `"separate"` means each out attribute will be written separate buffers. Based on these settings the *transform()* method will accept a single buffer or a list of buffer.

load_texture(*path: str | Path, *, flip: bool = True, build_mipmaps: bool = False*) → *Texture2D*

Loads and creates an OpenGL 2D texture. Currently, all textures are converted to RGBA for simplicity.

Example:

```
# Load a texture in current working directory
texture = window.ctx.load_texture("background.png")
# Load a texture using Arcade resource handle
texture = window.ctx.load_texture(":textures:background.png")
```

Parameters

- **path** – Path to texture
- **flip** – Flips the image upside down
- **build_mipmaps** – Build mipmaps for the texture

program(*, *vertex_shader: str*, *fragment_shader: str | None = None*, *geometry_shader: str | None = None*, *tess_control_shader: str | None = None*, *tess_evaluation_shader: str | None = None*, *common: List[str] | None = None*, *defines: Dict[str, str] | None = None*, *varyings: Sequence[str] | None = None*, *varyings_capture_mode: str = 'interleaved'*) → *Program*

Create a `Program` given the vertex, fragment and geometry shader.

Parameters

- **vertex_shader** – vertex shader source
- **fragment_shader** – fragment shader source (optional)
- **geometry_shader** – geometry shader source (optional)
- **tess_control_shader** – tessellation control shader source (optional)
- **tess_evaluation_shader** – tessellation evaluation shader source (optional)
- **common** – Common shader sources injected into all shaders
- **defines** – Substitute #defines values in the source (optional)
- **varyings** – The name of the out attributes in a transform shader. This is normally not necessary since we auto detect them, but some more complex out structures we can't detect.
- **varyings_capture_mode** – The capture mode for transforms. `"interleaved"` means all out attribute will be written to a single buffer. `"separate"` means each out attribute will be written separate buffers. Based on these settings the *transform()* method will accept a single buffer or a list of buffer.

pyglet_rendering()

Context manager for doing rendering with pyglet ensuring context states are reverted. This affects things like blending.

query(*, *samples=True*, *time=True*, *primitives=True*) → *Query*

Create a query object for measuring rendering calls in opengl.

Parameters

- **samples** – Collect written samples
- **time** – Measure rendering duration
- **primitives** – Collect the number of primitives emitted

reset() → None

Reset context flags and other states. This is mostly used in unit testing.

shader_inc(*source: str*) → str

Parse a shader source looking for `#include` directives and replace them with the contents of the included file.

The `#include` directive must be on its own line and the file and the path should use a resource handle.

Example:

```
#include :my_shader:lib/common.glsl
```

Parameters
source – Shader

texture(*size: Tuple[int, int], *, components: int = 4, dtype: str = 'f1', data: ByteString | memoryview | array | Array | None = None, wrap_x: int | None = None, wrap_y: int | None = None, filter: Tuple[int, int] | None = None, samples: int = 0, immutable: bool = False*) → *Texture2D*

Create a 2D Texture.

Wrap modes: `GL_REPEAT`, `GL_MIRRORED_REPEAT`, `GL_CLAMP_TO_EDGE`, `GL_CLAMP_TO_BORDER`

Minifying filters: `GL_NEAREST`, `GL_LINEAR`, `GL_NEAREST_MIPMAP_NEAREST`, `GL_LINEAR_MIPMAP_NEAREST` `GL_NEAREST_MIPMAP_LINEAR`, `GL_LINEAR_MIPMAP_LINEAR`

Magnifying filters: `GL_NEAREST`, `GL_LINEAR`

Parameters

- **size** (`Tuple[int, int]`) – The size of the texture
- **components** – Number of components (1: R, 2: RG, 3: RGB, 4: RGBA)
- **dtype** – The data type of each component: f1, f2, f4 / i1, i2, i4 / u1, u2, u4
- **data** – The texture data (optional). Can be `bytes` or any object supporting the buffer protocol.
- **wrap_x** – How the texture wraps in x direction
- **wrap_y** – How the texture wraps in y direction
- **filter** – Minification and magnification filter
- **samples** – Creates a multisampled texture for values > 0
- **immutable** – Make the storage (not the contents) immutable. This can sometimes be required when using textures with compute shaders.

BLEND = 3042

Blending

Type
Context flag

BLEND_ADDITIVE = (1, 1)

`ONE, ONE`

Type
Blend mode shortcut for additive blending

BLEND_DEFAULT = (770, 771)

`SRC_ALPHA, ONE_MINUS_SRC_ALPHA`

Type
Blend mode shortcut for default blend mode

BLEND_PREMULTIPLIED_ALPHA = (770, 1)

`SRC_ALPHA, ONE`

Type
Blend mode shortcut for pre-multiplied alpha

`CLAMP_TO_BORDER = 33069`

`CLAMP_TO_EDGE = 33071`

`CULL_FACE = 2884`
: Face culling

  Type
  Context flag

`DEPTH_TEST = 2929`
: Depth testing

  Type
  Context flag

`DST_ALPHA = 772`
: Blend function

`DST_COLOR = 774`
: Blend function

`FUNC_ADD = 32774`
: source + destination

`FUNC_REVERSE_SUBTRACT = 32779`
: destination - source

  Type
  Blend equations

`FUNC_SUBTRACT = 32778`
: source - destination

  Type
  Blend equations

`LINEAR = 9729`
: Linear interpolate

  Type
  Texture interpolation

`LINEAR_MIPMAP_LINEAR = 9987`
: Minification filter for mipmaps

  Type
  Texture interpolation

`LINEAR_MIPMAP_NEAREST = 9985`
: Minification filter for mipmaps

  Type
  Texture interpolation

`LINES = 1`
: Primitive mode

`LINES_ADJACENCY = 10`
: Primitive mode

`LINE_LOOP = 2`
Primitive mode

`LINE_STRIP = 3`
Primitive mode

`LINE_STRIP_ADJACENCY = 11`
Primitive mode

`MAX = 32776`
Maximum of source and destination

Type
Blend equations

`MIN = 32775`
Minimum of source and destination

Type
Blend equations

`MIRRORED_REPEAT = 33648`

`NEAREST = 9728`
Nearest pixel

Type
Texture interpolation

`NEAREST_MIPMAP_LINEAR = 9986`
Minification filter for mipmaps

Type
Texture interpolation

`NEAREST_MIPMAP_NEAREST = 9984`
Minification filter for mipmaps

Type
Texture interpolation

`ONE = 1`
Blend function

`ONE_MINUS_DST_ALPHA = 773`
Blend function

`ONE_MINUS_DST_COLOR = 775`
Blend function

`ONE_MINUS_SRC_ALPHA = 771`
Blend function

`ONE_MINUS_SRC_COLOR = 769`
Blend function

`PATCHES = 14`
Patch mode (tessellation)

POINTS = 0

Primitive mode

PROGRAM_POINT_SIZE = 34370

Enables `gl_PointSize` in vertex or geometry shaders.

When enabled we can write to `gl_PointSize` in the vertex shader to specify the point size for each individual point.

If this value is not set in the shader the behavior is undefined. This means the points may or may not appear depending if the drivers enforce some default value for `gl_PointSize`.

When disabled `Context.point_size` is used.

Type
Context flag

REPEAT = 10497

Repeat

Type
Texture wrap mode

SRC_ALPHA = 770

Blend function

SRC_COLOR = 768

Blend function

TRIANGLES = 4

Primitive mode

TRIANGLES_ADJACENCY = 12

Primitive mode

TRIANGLE_FAN = 6

Primitive mode

TRIANGLE_STRIP = 5

Primitive mode

TRIANGLE_STRIP_ADJACENCY = 13

Primitive mode

ZERO = 0

Blend function

active: 'Context' | None = None

The active context

atlas_size: Tuple[int, int] = (512, 512)

blend_func

Get or set the blend function. This is tuple specifying how the color and alpha blending factors are computed for the source and destination pixel.

When using a two component tuple you specify the blend function for the source and the destination.

When using a four component tuple you specify the blend function for the source color, source alpha destination color and destination alpha. (separate blend functions for color and alpha)

Supported blend functions are:

```
ZERO
ONE
SRC_COLOR
ONE_MINUS_SRC_COLOR
DST_COLOR
ONE_MINUS_DST_COLOR
SRC_ALPHA
ONE_MINUS_SRC_ALPHA
DST_ALPHA
ONE_MINUS_DST_ALPHA

# Shortcuts
DEFAULT_BLENDING     # (SRC_ALPHA, ONE_MINUS_SRC_ALPHA)
ADDITIVE_BLENDING    # (ONE, ONE)
PREMULTIPLIED_ALPHA  # (SRC_ALPHA, ONE)
```

These enums can be accessed in the `arcade.gl` module or simply as attributes of the context object. The raw enums from `pyglet.gl` can also be used.

Example:

```
# Using constants from the context object
ctx.blend_func = ctx.ONE, ctx.ONE
# from the gl module
from arcade import gl
ctx.blend_func = gl.ONE, gl.ONE
```

Type
tuple (src, dst)

`cull_face`

The face side to cull when face culling is enabled.

By default the back face is culled. This can be set to front, back or front_and_back:

```
ctx.cull_face = "front"
ctx.cull_face = "back"
ctx.cull_face = "front_and_back"
```

`default_atlas`

The default texture atlas. This is created when arcade is initialized. All sprite lists will use use this atlas unless a different atlas is passed in the `arcade.SpriteList` constructor.

Type
TextureAtlas

`error`

Check OpenGL error

Returns a string representation of the occurring error or `None` of no errors has occurred.

Example:

```
err = ctx.error
if err:
    raise RuntimeError("OpenGL error: {err}")
```

Type
str

fbo

Get the currently active framebuffer. This property is read-only

Type
arcade.gl.Framebuffer

front_face

Configure front face winding order of triangles.

By default the counter-clockwise winding side is the front face. This can be set set to clockwise or counter-clockwise:

```
ctx.front_face = "cw"
ctx.front_face = "ccw"
```

gc_mode

Set the garbage collection mode for OpenGL resources. Supported modes are:

```
# Default:
# Defer garbage collection until ctx.gc() is called
# This can be useful to enforce the main thread to
# run garbage collection of opengl resources
ctx.gc_mode = "context_gc"

# Auto collect is similar to python garbage collection.
# This is a risky mode. Know what you are doing before using this.
ctx.gc_mode = "auto"
```

gl_api: str = 'gl'

The OpenGL api. Usually "gl" or "gles".

gl_version

The OpenGL version as a 2 component tuple. This is the reported OpenGL version from drivers and might be a higher version than you requested.

Type
tuple (major, minor) version

info

Get the Limits object for this context containing information about hardware/driver limits and other context information.

Example:

```
>> ctx.info.MAX_TEXTURE_SIZE
(16384, 16384)
>> ctx.info.VENDOR
NVIDIA Corporation
>> ctx.info.RENDERER
NVIDIA GeForce RTX 2080 SUPER/PCIe/SSE2
```

limits

Get the Limits object for this context containing information about hardware/driver limits and other context information.

Warning: This an old alias for *info* and is only around for backwards compatibility.

Example:

```
>> ctx.limits.MAX_TEXTURE_SIZE
(16384, 16384)
>> ctx.limits.VENDOR
NVIDIA Corporation
>> ctx.limits.RENDERER
NVIDIA GeForce RTX 2080 SUPER/PCIe/SSE2
```

objects: Deque[Any]

Collected objects to gc when gc_mode is "context_gc". This can be used during debugging.

patch_vertices

Get or set number of vertices that will be used to make up a single patch primitive. Patch primitives are consumed by the tessellation control shader (if present) and subsequently used for tessellation.

Type
int

point_size

Set or get the point size. Default is *1.0*.

Point size changes the pixel size of rendered points. The min and max values are limited by `POINT_SIZE_RANGE`. This value usually at least `(1, 100)`, but this depends on the drivers/vendors.

If variable point size is needed you can enable *PROGRAM_POINT_SIZE* and write to `gl_PointSize` in the vertex or geometry shader.

Note: Using a geometry shader to create triangle strips from points is often a safer way to render large points since you don't have have any size restrictions.

primitive_restart_index

Get or set the primitive restart index. Default is `-1`. The primitive restart index can be used in index buffers to restart a primitive. This is for example useful when you use triangle strips or line strips and want to start on a new strip in the same buffer / draw call.

projection_2d

Get or set the global orthogonal projection for arcade.

This projection is used by sprites and shapes and is represented by four floats: `(left, right, bottom, top)`

When reading this property we reconstruct the projection parameters from pyglet's projection matrix. When setting this property we construct an orthogonal projection matrix and set it in pyglet.

Type
Tuple[float, float, float, float]

projection_2d_matrix

Get the current projection matrix. This 4x4 float32 matrix is calculated when setting *projection_2d*.

This property simply gets and sets pyglet's projection matrix.

Type
pyglet.math.Mat4

scissor

Get or set the scissor box for the active framebuffer. This is a shortcut for *scissor()*.

By default the scissor box is disabled and has no effect and will have an initial value of `None`. The scissor box is enabled when setting a value and disabled when set to `None`.

Example:

```
# Set and enable scissor box only drawing
# in a 100 x 100 pixel lower left area
ctx.scissor = 0, 0, 100, 100
# Disable scissoring
ctx.scissor = None
```

Type
tuple (x, y, width, height)

screen

The framebuffer for the window.

Type
`Framebuffer`

stats

Get the stats instance containing runtime information about creation and destruction of OpenGL objects.

Example:

```
>> ctx.limits.MAX_TEXTURE_SIZE
(16384, 16384)
>> ctx.limits.VENDOR
NVIDIA Corporation
>> ctx.limits.RENDERER
NVIDIA GeForce RTX 2080 SUPER/PCIe/SSE2
```

view_matrix_2d

Get the current view matrix. This 4x4 float32 matrix is calculated when setting *view_matrix_2d*.

This property simply gets and sets pyglet's view matrix.

Type
pyglet.math.Mat4

viewport

Get or set the viewport for the currently active framebuffer. The viewport simply describes what pixels of the screen OpenGL should render to. Normally it would be the size of the window's framebuffer:

```
# 4:3 screen
ctx.viewport = 0, 0, 800, 600
# 1080p
ctx.viewport = 0, 0, 1920, 1080
# Using the current framebuffer size
ctx.viewport = 0, 0, *ctx.screen.size
```

Type
tuple (x, y, width, height)

`window`

The window this context belongs to.

Type

`pyglet.Window`

`wireframe`

Get or set the wireframe mode. When enabled all primitives will be rendered as lines.

Type

bool

33.27 Math

`arcade.math.clamp`(*a*, *low: float*, *high: float*) → float

Clamp a number between a range.

`arcade.math.get_angle_degrees`(*x1: float*, *y1: float*, *x2: float*, *y2: float*) → float

Get the angle in degrees between two points.

Parameters

- **x1** – x coordinate of the first point
- **y1** – y coordinate of the first point
- **x2** – x coordinate of the second point
- **y2** – y coordinate of the second point

`arcade.math.get_angle_radians`(*x1: float*, *y1: float*, *x2: float*, *y2: float*) → float

Get the angle in radians between two points.

Parameters

- **x1** – x coordinate of the first point
- **y1** – y coordinate of the first point
- **x2** – x coordinate of the second point
- **y2** – y coordinate of the second point

`arcade.math.get_distance`(*x1: float*, *y1: float*, *x2: float*, *y2: float*) → float

Get the distance between two points.

Parameters

- **x1** – x coordinate of the first point
- **y1** – y coordinate of the first point
- **x2** – x coordinate of the second point
- **y2** – y coordinate of the second point

Returns

Distance between the two points

`arcade.math.lerp`(*v1: float*, *v2: float*, *u: float*) → float

linearly interpolate between two values

`arcade.math.`**`lerp_angle`**(*start_angle: float*, *end_angle: float*, *u: float*) → float

Linearly interpolate between two angles in degrees, following the shortest path.

Parameters

- **`start_angle`** – The starting angle
- **`end_angle`** – The ending angle
- **`u`** – The interpolation value

Returns
The interpolated angle

`arcade.math.`**`lerp_vec`**(*v1: Tuple[float, float]*, *v2: Tuple[float, float]*, *u: float*) → Tuple[float, float]

`arcade.math.`**`rand_angle_360_deg`**() → float

Returns a random angle in degrees.

`arcade.math.`**`rand_angle_spread_deg`**(*angle: float*, *half_angle_spread: float*) → float

Returns a random angle in degrees, within a spread of the given angle.

Parameters

- **`angle`** – The angle to spread from
- **`half_angle_spread`** – The half angle spread

Returns
A random angle in degrees

`arcade.math.`**`rand_in_circle`**(*center: Tuple[float, float]*, *radius: float*) → Tuple[float, float]

Generate a point in a circle, or can think of it as a vector pointing a random direction with a random magnitude <= radius.

Reference: https://stackoverflow.com/a/30564123

Note: This algorithm returns a higher concentration of points around the center of the circle

Parameters

- **`center`** – The center of the circle
- **`radius`** – The radius of the circle

Returns
A random point in the circle

`arcade.math.`**`rand_in_rect`**(*bottom_left: Tuple[float, float]*, *width: float*, *height: float*) → Tuple[float, float]

Calculate a random point in a rectangle.

Parameters

- **`bottom_left`** – The bottom left corner of the rectangle
- **`width`** – The width of the rectangle
- **`height`** – The height of the rectangle

Returns
A random point in the rectangle

`arcade.math.`**`rand_on_circle`**(*center: Tuple[float, float], radius: float*) → Tuple[float, float]

Generate a point on a circle.

Parameters

- **`center`** – The center of the circle
- **`radius`** – The radius of the circle

Returns

A random point on the circle

`arcade.math.`**`rand_on_line`**(*pos1: Tuple[float, float], pos2: Tuple[float, float]*) → Tuple[float, float]

Given two points defining a line, return a random point on that line.

Parameters

- **`pos1`** – The first point
- **`pos2`** – The second point

Returns

A random point on the line

`arcade.math.`**`rand_vec_magnitude`**(*angle: float, lo_magnitude: float, hi_magnitude: float*) → Tuple[float, float]

Returns a random vector, within a spread of the given angle.

Parameters

- **`angle`** – The angle to spread from
- **`lo_magnitude`** – The lower magnitude
- **`hi_magnitude`** – The higher magnitude

Returns

A random vector

`arcade.math.`**`rand_vec_spread_deg`**(*angle: float, half_angle_spread: float, length: float*) → Tuple[float, float]

Returns a random vector, within a spread of the given angle.

Parameters

- **`angle`** – The angle to spread from
- **`half_angle_spread`** – The half angle spread
- **`length`** – The length of the vector

Returns

A random vector

`arcade.math.`**`rotate_point`**(*x: float, y: float, cx: float, cy: float, angle_degrees: float*) → Tuple[float, float]

Rotate a point around a center.

Parameters

- **`x`** – x value of the point you want to rotate
- **`y`** – y value of the point you want to rotate
- **`cx`** – x value of the center point you want to rotate around
- **`cy`** – y value of the center point you want to rotate around
- **`angle_degrees`** – Angle, in degrees, to rotate

Returns
Return rotated (x, y) pair

`arcade.math.round_fast`(*value: float*, *precision: int*) → float

A high performance version of python's built-in round() function.

Note: This function is not as accurate as the built-in round() function. But is sufficient in some cases.

Example:

```
>>> round(3.5662457892, 1)
3.6
>>> round(3.5662457892, 2)
3.57
>>> round(3.5662457892, 3)
3.566
>>> round(3.5662457892, 4)
3.5662
```

Parameters

- **`value`** – The value to round
- **`precision`** – The number of decimal places to round to

Returns
The rounded value

33.28 OpenGL

This is the low level rendering API in Arcade and is used internally for all drawing/rendering. It's a higher level wrapper over OpenGL 3.3+ core and gives the user easy access to GPU programs (shaders), textures, framebuffers, queries, buffers, vertex arrays/geometry and compute shaders (Note that compute shaders are not supported on MacOS).

This API is also heavily inspired by ModernGL. It's basically a subset of ModernGL except we are using pyglet's OpenGL bindings. However, we don't have the context flexibility and speed of ModernGL, but we are at the very least on par with PyOpenGL or slightly better because pyglet's OpenGL bindings are very light. The higher level abstraction is the main selling point as it saves the user from an enormous amount of work.

Note that all resources are created through the `arcade.gl.Context` / `arcade.ArcadeContext`. An instance of this type should be accessible the window (`arcade.Window.ctx`).

This API can also be used with pyglet by creating an instance of `arcade.gl.Context` after the window creation. The `arcade.ArcadeContext` on the other hand extends the default Context with arcade specific helper methods and should only be used by arcade.

Some prior knowledge of OpenGL might be needed to understand how this API works, but we do have examples in the experimental directory (git).

33.28.1 Context

Context

class `arcade.gl.Context`(*window: BaseWindow*, *gc_mode: str = 'context_gc'*, *gl_api: str = 'gl'*)

Bases:

Represents an OpenGL context. This context belongs to a `pyglet.Window` normally accessed through `window.ctx`.

The Context class contains methods for creating resources, global states and commonly used enums. All enums also exist in the `gl` module. (`ctx.BLEND` or `arcade.gl.BLEND`).

active: *Context* | None = None

The active context

NEAREST = 9728

Nearest pixel

Type
Texture interpolation

LINEAR = 9729

Linear interpolate

Type
Texture interpolation

NEAREST_MIPMAP_NEAREST = 9984

Minification filter for mipmaps

Type
Texture interpolation

LINEAR_MIPMAP_NEAREST = 9985

Minification filter for mipmaps

Type
Texture interpolation

NEAREST_MIPMAP_LINEAR = 9986

Minification filter for mipmaps

Type
Texture interpolation

LINEAR_MIPMAP_LINEAR = 9987

Minification filter for mipmaps

Type
Texture interpolation

REPEAT = 10497

Repeat

Type
Texture wrap mode

CLAMP_TO_EDGE = 33071

`CLAMP_TO_BORDER = 33069`

`MIRRORED_REPEAT = 33648`

`BLEND = 3042`

Blending

Type
Context flag

`DEPTH_TEST = 2929`

Depth testing

Type
Context flag

`CULL_FACE = 2884`

Face culling

Type
Context flag

`PROGRAM_POINT_SIZE = 34370`

Enables `gl_PointSize` in vertex or geometry shaders.

When enabled we can write to `gl_PointSize` in the vertex shader to specify the point size for each individual point.

If this value is not set in the shader the behavior is undefined. This means the points may or may not appear depending if the drivers enforce some default value for `gl_PointSize`.

When disabled *`Context.point_size`* is used.

Type
Context flag

`ZERO = 0`

Blend function

`ONE = 1`

Blend function

`SRC_COLOR = 768`

Blend function

`ONE_MINUS_SRC_COLOR = 769`

Blend function

`SRC_ALPHA = 770`

Blend function

`ONE_MINUS_SRC_ALPHA = 771`

Blend function

`DST_ALPHA = 772`

Blend function

`ONE_MINUS_DST_ALPHA = 773`

Blend function

`DST_COLOR = 774`
: Blend function

`ONE_MINUS_DST_COLOR = 775`
: Blend function

`FUNC_ADD = 32774`
: source + destination

`FUNC_SUBTRACT = 32778`
: source - destination

    Type
    Blend equations

`FUNC_REVERSE_SUBTRACT = 32779`
: destination - source

    Type
    Blend equations

`MIN = 32775`
: Minimum of source and destination

    Type
    Blend equations

`MAX = 32776`
: Maximum of source and destination

    Type
    Blend equations

`BLEND_DEFAULT = (770, 771)`
: `SRC_ALPHA, ONE_MINUS_SRC_ALPHA`

    Type
    Blend mode shortcut for default blend mode

`BLEND_ADDITIVE = (1, 1)`
: `ONE, ONE`

    Type
    Blend mode shortcut for additive blending

`BLEND_PREMULTIPLIED_ALPHA = (770, 1)`
: `SRC_ALPHA, ONE`

    Type
    Blend mode shortcut for pre-multiplied alpha

`POINTS = 0`
: Primitive mode

`LINES = 1`
: Primitive mode

`LINE_LOOP = 2`
: Primitive mode

`LINE_STRIP = 3`

Primitive mode

`TRIANGLES = 4`

Primitive mode

`TRIANGLE_STRIP = 5`

Primitive mode

`TRIANGLE_FAN = 6`

Primitive mode

`LINES_ADJACENCY = 10`

Primitive mode

`LINE_STRIP_ADJACENCY = 11`

Primitive mode

`TRIANGLES_ADJACENCY = 12`

Primitive mode

`TRIANGLE_STRIP_ADJACENCY = 13`

Primitive mode

`PATCHES = 14`

Patch mode (tessellation)

`gl_api: str = 'gl'`

The OpenGL api. Usually "gl" or "gles".

`objects: Deque[Any]`

Collected objects to gc when gc_mode is "context_gc". This can be used during debugging.

`info`

Get the Limits object for this context containing information about hardware/driver limits and other context information.

Example:

```
>> ctx.info.MAX_TEXTURE_SIZE
(16384, 16384)
>> ctx.info.VENDOR
NVIDIA Corporation
>> ctx.info.RENDERER
NVIDIA GeForce RTX 2080 SUPER/PCIe/SSE2
```

`limits`

Get the Limits object for this context containing information about hardware/driver limits and other context information.

Warning: This an old alias for *info* and is only around for backwards compatibility.

Example:

```
>> ctx.limits.MAX_TEXTURE_SIZE
(16384, 16384)
>> ctx.limits.VENDOR
NVIDIA Corporation
>> ctx.limits.RENDERER
NVIDIA GeForce RTX 2080 SUPER/PCIe/SSE2
```

stats

Get the stats instance containing runtime information about creation and destruction of OpenGL objects.

Example:

```
>> ctx.limits.MAX_TEXTURE_SIZE
(16384, 16384)
>> ctx.limits.VENDOR
NVIDIA Corporation
>> ctx.limits.RENDERER
NVIDIA GeForce RTX 2080 SUPER/PCIe/SSE2
```

window

The window this context belongs to.

Type
`pyglet.Window`

screen

The framebuffer for the window.

Type
`Framebuffer`

fbo

Get the currently active framebuffer. This property is read-only

Type
arcade.gl.Framebuffer

gl_version

The OpenGL version as a 2 component tuple. This is the reported OpenGL version from drivers and might be a higher version than you requested.

Type
tuple (major, minor) version

gc() → int

Run garbage collection of OpenGL objects for this context. This is only needed when `gc_mode` is `context_gc`.

Returns
The number of resources destroyed

gc_mode

Set the garbage collection mode for OpenGL resources. Supported modes are:

```
# Default:
# Defer garbage collection until ctx.gc() is called
# This can be useful to enforce the main thread to
```

(continues on next page)

(continued from previous page)

```
# run garbage collection of opengl resources
ctx.gc_mode = "context_gc"

# Auto collect is similar to python garbage collection.
# This is a risky mode. Know what you are doing before using this.
ctx.gc_mode = "auto"
```

error

Check OpenGL error

Returns a string representation of the occurring error or `None` of no errors has occurred.

Example:

```
err = ctx.error
if err:
    raise RuntimeError("OpenGL error: {err}")
```

Type
str

classmethod activate(*ctx: Context*)

Mark a context as the currently active one.

Warning: Never call this unless you know exactly what you are doing.

enable(**flags*)

Enables one or more context flags:

```
# Single flag
ctx.enable(ctx.BLEND)
# Multiple flags
ctx.enable(ctx.DEPTH_TEST, ctx.CULL_FACE)
```

enable_only(**args*)

Enable only some flags. This will disable all other flags. This is a simple way to ensure that context flag states are not lingering from other sections of your code base:

```
# Ensure all flags are disabled (enable no flags)
ctx.enable_only()
# Make sure only blending is enabled
ctx.enable_only(ctx.BLEND)
# Make sure only depth test and culling is enabled
ctx.enable_only(ctx.DEPTH_TEST, ctx.CULL_FACE)
```

enabled(**flags*)

Temporarily change enabled flags.

Flags that was enabled initially will stay enabled. Only new enabled flags will be reversed when exiting the context.

Example:

```
with ctx.enabled(ctx.BLEND, ctx.CULL_FACE):
    # Render something
```

enabled_only(**flags*)

Temporarily change enabled flags.

Only the supplied flags with be enabled in in the context. When exiting the context the old flags will be restored.

Example:

```
with ctx.enabled_only(ctx.BLEND, ctx.CULL_FACE):
    # Render something
```

disable(**args*)

Disable one or more context flags:

```
# Single flag
ctx.disable(ctx.BLEND)
# Multiple flags
ctx.disable(ctx.DEPTH_TEST, ctx.CULL_FACE)
```

is_enabled(*flag*) → bool

Check if a context flag is enabled

Type
bool

viewport

Get or set the viewport for the currently active framebuffer. The viewport simply describes what pixels of the screen OpenGL should render to. Normally it would be the size of the window's framebuffer:

```
# 4:3 screen
ctx.viewport = 0, 0, 800, 600
# 1080p
ctx.viewport = 0, 0, 1920, 1080
# Using the current framebuffer size
ctx.viewport = 0, 0, *ctx.screen.size
```

Type
tuple (x, y, width, height)

scissor

Get or set the scissor box for the active framebuffer. This is a shortcut for `scissor()`.

By default the scissor box is disabled and has no effect and will have an initial value of `None`. The scissor box is enabled when setting a value and disabled when set to `None`.

Example:

```
# Set and enable scissor box only drawing
# in a 100 x 100 pixel lower left area
ctx.scissor = 0, 0, 100, 100
# Disable scissoring
ctx.scissor = None
```

Type
tuple (x, y, width, height)

blend_func

Get or set the blend function. This is tuple specifying how the color and alpha blending factors are computed for the source and destination pixel.

When using a two component tuple you specify the blend function for the source and the destination.

When using a four component tuple you specify the blend function for the source color, source alpha destination color and destination alpha. (separate blend functions for color and alpha)

Supported blend functions are:

```
ZERO
ONE
SRC_COLOR
ONE_MINUS_SRC_COLOR
DST_COLOR
ONE_MINUS_DST_COLOR
SRC_ALPHA
ONE_MINUS_SRC_ALPHA
DST_ALPHA
ONE_MINUS_DST_ALPHA

# Shortcuts
DEFAULT_BLENDING     # (SRC_ALPHA, ONE_MINUS_SRC_ALPHA)
ADDITIVE_BLENDING    # (ONE, ONE)
PREMULTIPLIED_ALPHA  # (SRC_ALPHA, ONE)
```

These enums can be accessed in the `arcade.gl` module or simply as attributes of the context object. The raw enums from `pyglet.gl` can also be used.

Example:

```
# Using constants from the context object
ctx.blend_func = ctx.ONE, ctx.ONE
# from the gl module
from arcade import gl
ctx.blend_func = gl.ONE, gl.ONE
```

Type
tuple (src, dst)

front_face

Configure front face winding order of triangles.

By default the counter-clockwise winding side is the front face. This can be set set to clockwise or counter-clockwise:

```
ctx.front_face = "cw"
ctx.front_face = "ccw"
```

cull_face

The face side to cull when face culling is enabled.

By default the back face is culled. This can be set to front, back or front_and_back:

```
ctx.cull_face = "front"
ctx.cull_face = "back"
ctx.cull_face = "front_and_back"
```

wireframe

Get or set the wireframe mode. When enabled all primitives will be rendered as lines.

Type
bool

patch_vertices

Get or set number of vertices that will be used to make up a single patch primitive. Patch primitives are consumed by the tessellation control shader (if present) and subsequently used for tessellation.

Type
int

point_size

Set or get the point size. Default is *1.0*.

Point size changes the pixel size of rendered points. The min and max values are limited by `POINT_SIZE_RANGE`. This value usually at least `(1, 100)`, but this depends on the drivers/vendors.

If variable point size is needed you can enable *PROGRAM_POINT_SIZE* and write to `gl_PointSize` in the vertex or geometry shader.

Note: Using a geometry shader to create triangle strips from points is often a safer way to render large points since you don't have have any size restrictions.

primitive_restart_index

Get or set the primitive restart index. Default is `-1`. The primitive restart index can be used in index buffers to restart a primitive. This is for example useful when you use triangle strips or line strips and want to start on a new strip in the same buffer / draw call.

finish() → None

Wait until all OpenGL rendering commands are completed.

This function will actually stall until all work is done and may have severe performance implications.

flush() → None

A suggestion to the driver to execute all the queued drawing calls even if the queue is not full yet. This is not a blocking call and only a suggestion. This can potentially be used for speedups when we don't have anything else to render.

copy_framebuffer(*src:* Framebuffer, *dst:* Framebuffer)

Copies/blits a framebuffer to another one.

This operation has many restrictions to ensure it works across different platforms and drivers:

- The source and destination framebuffer must be the same size
- The formats of the attachments must be the same
- Only the source framebuffer can be multisampled
- Framebuffers cannot have integer attachments

Parameters

- **src** – The framebuffer to copy from
- **dst** – The framebuffer we copy to

buffer(**, data: ByteString | memoryview | array | Array | None = None, reserve: int = 0, usage: str = 'static'*) → *Buffer*

Create an OpenGL Buffer object. The buffer will contain all zero-bytes if no data is supplied.

Examples:

```
# Create 1024 byte buffer
ctx.buffer(reserve=1024)
# Create a buffer with 1000 float values using python's array.array
from array import array
ctx.buffer(data=array('f', [i for in in range(1000)])
# Create a buffer with 1000 random 32 bit floats using numpy
self.ctx.buffer(data=np.random.random(1000).astype("f4"))
```

The `data` parameter can be anything that implements the Buffer Protocol.

This includes `bytes`, `bytearray`, `array.array`, and more. You may need to use typing workarounds for non-builtin types. See *Writing Raw Bytes to GL Buffers & Textures* for more information.

The `usage` parameter enables the GL implementation to make more intelligent decisions that may impact buffer object performance. It does not add any restrictions. If in doubt, skip this parameter and revisit when optimizing. The result are likely to be different between vendors/drivers or may not have any effect.

The available values mean the following:

```
stream
    The data contents will be modified once and used at most a few times.
static
    The data contents will be modified once and used many times.
dynamic
    The data contents will be modified repeatedly and used many times.
```

Parameters

- **data** – The buffer data. This can be a `bytes` instance or any any other object supporting the buffer protocol.
- **reserve** – The number of bytes to reserve
- **usage** – Buffer usage. 'static', 'dynamic' or 'stream'

framebuffer(**, color_attachments: Texture2D | List[Texture2D] | None = None, depth_attachment: Texture2D | None = None*) → *Framebuffer*

Create a Framebuffer.

Parameters

- **color_attachments** – List of textures we want to render into
- **depth_attachment** – Depth texture

texture(*size: Tuple[int, int], *, components: int = 4, dtype: str = 'f1', data: ByteString | memoryview | array | Array | None = None, wrap_x: int | None = None, wrap_y: int | None = None, filter: Tuple[int, int] | None = None, samples: int = 0, immutable: bool = False*) → *Texture2D*

Create a 2D Texture.

Wrap modes: `GL_REPEAT`, `GL_MIRRORED_REPEAT`, `GL_CLAMP_TO_EDGE`, `GL_CLAMP_TO_BORDER`

Minifying filters: `GL_NEAREST`, `GL_LINEAR`, `GL_NEAREST_MIPMAP_NEAREST`, `GL_LINEAR_MIPMAP_NEAREST` `GL_NEAREST_MIPMAP_LINEAR`, `GL_LINEAR_MIPMAP_LINEAR`

Magnifying filters: `GL_NEAREST`, `GL_LINEAR`

Parameters

- **size** (*Tuple[int, int]*) – The size of the texture
- **components** – Number of components (1: R, 2: RG, 3: RGB, 4: RGBA)
- **dtype** – The data type of each component: f1, f2, f4 / i1, i2, i4 / u1, u2, u4
- **data** – The texture data (optional). Can be `bytes` or any object supporting the buffer protocol.
- **wrap_x** – How the texture wraps in x direction
- **wrap_y** – How the texture wraps in y direction
- **filter** – Minification and magnification filter
- **samples** – Creates a multisampled texture for values > 0
- **immutable** – Make the storage (not the contents) immutable. This can sometimes be required when using textures with compute shaders.

depth_texture(*size: Tuple[int, int]*, ***, *data: ByteString | memoryview | array | Array | None = None*) → *Texture2D*

Create a 2D depth texture. Can be used as a depth attachment in a `Framebuffer`.

Parameters

- **size** (*Tuple[int, int]*) – The size of the texture
- **data** – The texture data (optional). Can be `bytes` or any object supporting the buffer protocol.

geometry(*content: Sequence[BufferDescription] | None = None*, *index_buffer: Buffer | None = None*, *mode: int | None = None*, *index_element_size: int = 4*)

Create a Geometry instance. This is Arcade's version of a vertex array adding a lot of convenience for the user. Geometry objects are fairly light. They are mainly responsible for automatically map buffer inputs to your shader(s) and provide various methods for rendering or processing this geometry,

The same geometry can be rendered with different programs as long as your shader is using one or more of the input attribute. This means geometry with positions and colors can be rendered with a program only using the positions. We will automatically map what is necessary and cache these mappings internally for performace.

In short, the geometry object is a light object that describes what buffers contains and automatically negotiate with shaders/programs. This is a very complex field in OpenGL so the Geometry object provides substantial time savings and greatly reduces the complexity of your code.

Geometry also provide rendering methods supporting the following:

- Rendering geometry with and without index buffer
- Rendering your geometry using instancing. Per instance buffers can be provided or the current instance can be looked up using `gl_InstanceID` in shaders.

- Running transform feedback shaders that writes to buffers instead the screen. This can write to one or multiple buffer.
- Render your geometry with indirect rendering. This means packing multiple meshes into the same buffer(s) and batch drawing them.

Examples:

```
# Single buffer geometry with a vec2 vertex position attribute
ctx.geometry([BufferDescription(buffer, '2f', ["in_vert"])], mode=ctx.TRIANGLES)

# Single interlaved buffer with two attributes. A vec2 position and vec2␣
↪velocity
ctx.geometry([
        BufferDescription(buffer, '2f 2f', ["in_vert", "in_velocity"])
    ],
    mode=ctx.POINTS,
)

# Geometry with index buffer
ctx.geometry(
    [BufferDescription(buffer, '2f', ["in_vert"])],
    index_buffer=ibo,
    mode=ctx.TRIANGLES,
)

# Separate buffers
ctx.geometry([
        BufferDescription(buffer_pos, '2f', ["in_vert"])
        BufferDescription(buffer_vel, '2f', ["in_velocity"])
    ],
    mode=ctx.POINTS,
)

# Providing per-instance data for instancing
ctx.geometry([
        BufferDescription(buffer_pos, '2f', ["in_vert"])
        BufferDescription(buffer_instance_pos, '2f', ["in_offset"],␣
↪instanced=True)
    ],
    mode=ctx.POINTS,
)
```

Parameters

- **content** – List of *BufferDescription* (optional)
- **index_buffer** – Index/element buffer (optional)
- **mode** – The default draw mode (optional)
- **mode** – The default draw mode (optional)
- **index_element_size** – Byte size of a single index/element in the index buffer. In other words, the index buffer can be 8, 16 or 32 bit integers. Can be 1, 2 or 4 (8, 16 or 32 bit unsigned integer)

program(**, vertex_shader: str, fragment_shader: str | None = None, geometry_shader: str | None = None, tess_control_shader: str | None = None, tess_evaluation_shader: str | None = None, common: List[str] | None = None, defines: Dict[str, str] | None = None, varyings: Sequence[str] | None = None, varyings_capture_mode: str = 'interleaved'*) → *Program*

Create a `Program` given the vertex, fragment and geometry shader.

Parameters

- **vertex_shader** – vertex shader source
- **fragment_shader** – fragment shader source (optional)
- **geometry_shader** – geometry shader source (optional)
- **tess_control_shader** – tessellation control shader source (optional)
- **tess_evaluation_shader** – tessellation evaluation shader source (optional)
- **common** – Common shader sources injected into all shaders
- **defines** – Substitute #defines values in the source (optional)
- **varyings** – The name of the out attributes in a transform shader. This is normally not necessary since we auto detect them, but some more complex out structures we can't detect.
- **varyings_capture_mode** – The capture mode for transforms. `"interleaved"` means all out attribute will be written to a single buffer. `"separate"` means each out attribute will be written separate buffers. Based on these settings the *transform()* method will accept a single buffer or a list of buffer.

query(**, samples=True, time=True, primitives=True*) → *Query*

Create a query object for measuring rendering calls in opengl.

Parameters

- **samples** – Collect written samples
- **time** – Measure rendering duration
- **primitives** – Collect the number of primitives emitted

compute_shader(**, source: str, common: Iterable[str] = ()*) → *ComputeShader*

Create a compute shader.

Parameters

- **source** – The glsl source
- **common** – Common / library source injected into compute shader

ContextStats

class arcade.gl.context.**ContextStats**(*warn_threshold=100*)

Bases:

Runtime allocation statistics of OpenGL objects.

texture

Textures (created, freed)

framebuffer

Framebuffers (created, freed)

buffer
Buffers (created, freed)

program
Programs (created, freed)

vertex_array
Vertex Arrays (created, freed)

geometry
Geometry (created, freed)

compute_shader
Compute Shaders (created, freed)

query
Queries (created, freed)

incr(*key: str*) → None
Increments a counter.

Parameters
key – The attribute name / counter to increment.

decr(*key*)
Decrement a counter.

Parameters
key – The attribute name / counter to decrement.

Limits

class arcade.gl.context.**Limits**(*ctx*)
Bases:

OpenGL Limitations

MINOR_VERSION
Minor version number of the OpenGL API supported by the current context

MAJOR_VERSION
Major version number of the OpenGL API supported by the current context.

VENDOR
The vendor string. For example "NVIDIA Corporation"

RENDERER
The renderer things. For example "NVIDIA GeForce RTX 2080 SUPER/PCIe/SSE2"

SAMPLE_BUFFERS
Value indicating the number of sample buffers associated with the framebuffer

SUBPIXEL_BITS
An estimate of the number of bits of subpixel resolution that are used to position rasterized geometry in window coordinates

UNIFORM_BUFFER_OFFSET_ALIGNMENT
Minimum required alignment for uniform buffer sizes and offset

`MAX_ARRAY_TEXTURE_LAYERS`
: Value indicates the maximum number of layers allowed in an array texture, and must be at least 256

`MAX_3D_TEXTURE_SIZE`
: A rough estimate of the largest 3D texture that the GL can handle. The value must be at least 64

`MAX_COLOR_ATTACHMENTS`
: Maximum number of color attachments in a framebuffer

`MAX_COLOR_TEXTURE_SAMPLES`
: Maximum number of samples in a color multisample texture

`MAX_COMBINED_FRAGMENT_UNIFORM_COMPONENTS`
: the number of words for fragment shader uniform variables in all uniform blocks

`MAX_COMBINED_GEOMETRY_UNIFORM_COMPONENTS`
: Number of words for geometry shader uniform variables in all uniform blocks

`MAX_COMBINED_TEXTURE_IMAGE_UNITS`
: Maximum supported texture image units that can be used to access texture maps from the vertex shader

`MAX_COMBINED_UNIFORM_BLOCKS`
: Maximum number of uniform blocks per program

`MAX_COMBINED_VERTEX_UNIFORM_COMPONENTS`
: Number of words for vertex shader uniform variables in all uniform blocks

`MAX_CUBE_MAP_TEXTURE_SIZE`
: A rough estimate of the largest cube-map texture that the GL can handle

`MAX_DEPTH_TEXTURE_SAMPLES`
: Maximum number of samples in a multisample depth or depth-stencil texture

`MAX_DRAW_BUFFERS`
: Maximum number of simultaneous outputs that may be written in a fragment shader

`MAX_ELEMENTS_INDICES`
: Recommended maximum number of vertex array indices

`MAX_ELEMENTS_VERTICES`
: Recommended maximum number of vertex array vertices

`MAX_FRAGMENT_INPUT_COMPONENTS`
: Maximum number of components of the inputs read by the fragment shader

`MAX_FRAGMENT_UNIFORM_COMPONENTS`
: Maximum number of individual floating-point, integer, or boolean values that can be held in uniform variable storage for a fragment shader

`MAX_FRAGMENT_UNIFORM_VECTORS`
: maximum number of individual 4-vectors of floating-point, integer, or boolean values that can be held in uniform variable storage for a fragment shader

`MAX_FRAGMENT_UNIFORM_BLOCKS`
: Maximum number of uniform blocks per fragment shader.

`MAX_GEOMETRY_INPUT_COMPONENTS`
: Maximum number of components of inputs read by a geometry shader

MAX_GEOMETRY_OUTPUT_COMPONENTS
Maximum number of components of outputs written by a geometry shader

MAX_GEOMETRY_TEXTURE_IMAGE_UNITS
Maximum supported texture image units that can be used to access texture maps from the geometry shader

MAX_GEOMETRY_UNIFORM_BLOCKS
Maximum number of uniform blocks per geometry shader

MAX_GEOMETRY_UNIFORM_COMPONENTS
Maximum number of individual floating-point, integer, or boolean values that can be held in uniform variable storage for a geometry shader

MAX_INTEGER_SAMPLES
Maximum number of samples supported in integer format multisample buffers

MAX_SAMPLES
Maximum samples for a framebuffer

MAX_RENDERBUFFER_SIZE
Maximum supported size for renderbuffers

MAX_SAMPLE_MASK_WORDS
Maximum number of sample mask words

MAX_TEXTURE_SIZE
The value gives a rough estimate of the largest texture that the GL can handle

MAX_UNIFORM_BUFFER_BINDINGS
Maximum number of uniform buffer binding points on the context

MAX_UNIFORM_BLOCK_SIZE
Maximum size in basic machine units of a uniform block

MAX_VARYING_VECTORS
The number 4-vectors for varying variables

MAX_VERTEX_ATTRIBS
Maximum number of 4-component generic vertex attributes accessible to a vertex shader.

MAX_VERTEX_TEXTURE_IMAGE_UNITS
Maximum supported texture image units that can be used to access texture maps from the vertex shader.

MAX_VERTEX_UNIFORM_COMPONENTS
Maximum number of individual floating-point, integer, or boolean values that can be held in uniform variable storage for a vertex shader

MAX_VERTEX_UNIFORM_VECTORS
Maximum number of 4-vectors that may be held in uniform variable storage for the vertex shader

MAX_VERTEX_OUTPUT_COMPONENTS
Maximum number of components of output written by a vertex shader

MAX_VERTEX_UNIFORM_BLOCKS
Maximum number of uniform blocks per vertex shader.

MAX_TEXTURE_MAX_ANISOTROPY
The highest supported anisotropy value. Usually 8.0 or 16.0.

MAX_VIEWPORT_DIMS: Tuple[int, int]

The maximum support window or framebuffer viewport. This is usually the same as the maximum texture size

MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS

How many buffers we can have as output when doing a transform(feedback). This is usually 4

POINT_SIZE_RANGE

The minimum and maximum point size

get_int_tuple(*enum: c_uint* | *int*, *length: Literal[2]*) → Tuple[int, int]

get_int_tuple(*enum: c_uint* | *int*, *length: int*) → Tuple[int, ...]

Get an enum as an int tuple

get(*enum: c_uint* | *int*, *default=0*) → int

Get an integer limit

get_float(*enum: c_uint* | *int*, *default=0.0*) → float

Get a float limit

get_str(*enum: c_uint* | *int*) → str

Get a string limit

33.28.2 Texture

class arcade.gl.**Texture2D**(*ctx: Context*, *size: Tuple[int, int]*, ***, *components: int = 4*, *dtype: str = 'f1'*, *data: BufferProtocol* | *None = None*, *filter: Tuple[PyGLuint, PyGLuint]* | *None = None*, *wrap_x: PyGLuint* | *None = None*, *wrap_y: PyGLuint* | *None = None*, *target=3553*, *depth=False*, *samples: int = 0*, *immutable: bool = False*)

Bases:

An OpenGL 2D texture. We can create an empty black texture or a texture from byte data. A texture can also be created with different datatypes such as float, integer or unsigned integer.

The best way to create a texture instance is through `arcade.gl.Context.texture()`

Supported `dtype` values are:

```
# Float formats
'f1': UNSIGNED_BYTE
'f2': HALF_FLOAT
'f4': FLOAT
# int formats
'i1': BYTE
'i2': SHORT
'i4': INT
# uint formats
'u1': UNSIGNED_BYTE
'u2': UNSIGNED_SHORT
'u4': UNSIGNED_INT
```

Parameters

- **ctx** – The context the object belongs to
- **size** (*Tuple[int, int]*) – The size of the texture

- **components** – The number of components (1: R, 2: RG, 3: RGB, 4: RGBA)
- **dtype** – The data type of each component: f1, f2, f4 / i1, i2, i4 / u1, u2, u4
- **data** – The texture data (optional). Can be bytes or any object supporting the buffer protocol.
- **filter** – The minification/magnification filter of the texture
- **wrap_x** – Wrap mode x
- **wrap_y** – Wrap mode y
- **target** – The texture type (Ignored. Legacy)
- **depth** – creates a depth texture if *True*
- **samples** – Creates a multisampled texture for values > 0. This value will be clamped between 0 and the max sample capability reported by the drivers.
- **immutable** – Make the storage (not the contents) immutable. This can sometimes be required when using textures with compute shaders.

resize(*size: Tuple[int, int]*)

Resize the texture. This will re-allocate the internal memory and all pixel data will be lost.

ctx

The context this texture belongs to

Type
`Context`

glo

The OpenGL texture id

Type
GLuint

width

The width of the texture in pixels

Type
int

height

The height of the texture in pixels

Type
int

dtype

The data type of each component

Type
str

size

The size of the texture as a tuple

Type
tuple (width, height)

`samples`

Number of samples if multisampling is enabled (read only)

Type
int

`byte_size`

The byte size of the texture.

Type
int

`components`

Number of components in the texture

Type
int

`component_size`

Size in bytes of each component

Type
int

`depth`

If this is a depth texture.

Type
bool

`immutable`

Does this texture have immutable storage?

Type
bool

`swizzle`

The swizzle mask of the texture (Default `'RGBA'`).

The swizzle mask change/reorder the `vec4` value returned by the `texture()` function in a GLSL shaders. This is represented by a 4 character string were each character can be:

```
'R' GL_RED
'G' GL_GREEN
'B' GL_BLUE
'A' GL_ALPHA
'0' GL_ZERO
'1' GL_ONE
```

Example:

```
# Alpha channel will always return 1.0
texture.swizzle = 'RGB1'

# Only return the red component. The rest is masked to 0.0
texture.swizzle = 'R000'

# Reverse the components
texture.swizzle = 'ABGR'
```

Type
str

filter

Get or set the `(min, mag)` filter for this texture. These are rules for how a texture interpolates. The filter is specified for minification and magnification.

Default value is `LINEAR, LINEAR`. Can be set to `NEAREST, NEAREST` for pixelated graphics.

When mipmapping is used the min filter needs to be one of the `MIPMAP` variants.

Accepted values:

```
# Enums can be accessed on the context or arcade.gl
NEAREST                # Nearest pixel
LINEAR                 # Linear interpolate
NEAREST_MIPMAP_NEAREST # Minification filter for mipmaps
LINEAR_MIPMAP_NEAREST  # Minification filter for mipmaps
NEAREST_MIPMAP_LINEAR  # Minification filter for mipmaps
LINEAR_MIPMAP_LINEAR   # Minification filter for mipmaps
```

Also see

- https://www.khronos.org/opengl/wiki/Texture#Mip_maps
- https://www.khronos.org/opengl/wiki/Sampler_Object#Filtering

Type
tuple (min filter, mag filter)

wrap_x

Get or set the horizontal wrapping of the texture. This decides how textures are read when texture coordinates are outside the `[0.0, 1.0]` area. Default value is `REPEAT`.

Valid options are:

```
# Note: Enums can also be accessed in arcade.gl
# Repeat pixels on the y axis
texture.wrap_x = ctx.REPEAT
# Repeat pixels on the y axis mirrored
texture.wrap_x = ctx.MIRRORED_REPEAT
# Repeat the edge pixels when reading outside the texture
texture.wrap_x = ctx.CLAMP_TO_EDGE
# Use the border color (black by default) when reading outside the texture
texture.wrap_x = ctx.CLAMP_TO_BORDER
```

Type
int

wrap_y

Get or set the horizontal wrapping of the texture. This decides how textures are read when texture coordinates are outside the `[0.0, 1.0]` area. Default value is `REPEAT`.

Valid options are:

```
# Note: Enums can also be accessed in arcade.gl
# Repeat pixels on the x axis
texture.wrap_x = ctx.REPEAT
# Repeat pixels on the x axis mirrored
texture.wrap_x = ctx.MIRRORED_REPEAT
# Repeat the edge pixels when reading outside the texture
texture.wrap_x = ctx.CLAMP_TO_EDGE
# Use the border color (black by default) when reading outside the texture
texture.wrap_x = ctx.CLAMP_TO_BORDER
```

Type
int

anisotropy
Get or set the anisotropy for this texture.

compare_func
Get or set the compare function for a depth texture:

```
texture.compare_func = None  # Disable depth comparison completely
texture.compare_func = '<='  # GL_LEQUAL
texture.compare_func = '<'   # GL_LESS
texture.compare_func = '>='  # GL_GEQUAL
texture.compare_func = '>'   # GL_GREATER
texture.compare_func = '=='  # GL_EQUAL
texture.compare_func = '!='  # GL_NOTEQUAL
texture.compare_func = '0'   # GL_NEVER
texture.compare_func = '1'   # GL_ALWAYS
```

Type
str

read(*level: int = 0*, *alignment: int = 1*) → bytes
Read the contents of the texture.

Parameters

- **level** – The texture level to read
- **alignment** – Alignment of the start of each row in memory in number of bytes. Possible values: 1,2,4

write(*data: ByteString | memoryview | array | Array | Buffer*, *level: int = 0*, *viewport=None*) → None
Write byte data from the passed source to the texture.

The `data` value can be either an *`arcade.gl.Buffer`* or anything that implements the Buffer Protocol.

The latter category includes `bytes`, `bytearray`, `array.array`, and more. You may need to use typing workarounds for non-builtin types. See *Writing Raw Bytes to GL Buffers & Textures* for more information.

Parameters

- **data** – *`Buffer`* or buffer protocol object with data to write.
- **level** – The texture level to write

- **viewport** (*Union[Tuple[int, int], Tuple[int, int, int, int]]*) – The area of the texture to write. 2 or 4 component tuple

build_mipmaps(*base: int = 0*, *max_level: int = 1000*) → None

Generate mipmaps for this texture.

The default values usually work well.

Mipmaps are successively smaller versions of an original texture with special filtering applied. Using mipmaps allows OpenGL to render scaled versions of original textures with fewer scaling artifacts.

Mipmaps can be made for textures of any size. Each mipmap version halves the width and height of the previous one (e.g. 256 x 256, 128 x 128, 64 x 64, etc) down to a minimum of 1 x 1.

Note: Mipmaps will only be used if a texture's filter is configured with a mipmap-type minification:

```
# Set up linear interpolating minification filter
texture.filter = ctx.LINEAR_MIPMAP_LINEAR, ctx.LINEAR
```

Parameters

- **base** – Level the mipmaps start at (usually 0)
- **max_level** – The maximum number of levels to generate

Also see: https://www.khronos.org/opengl/wiki/Texture#Mip_maps

delete()

Destroy the underlying OpenGL resource. Don't use this unless you know exactly what you are doing.

static delete_glo(*ctx: Context*, *glo: gl.GLuint*)

Destroy the texture. This is called automatically when the object is garbage collected.

Parameters

- **ctx** – OpenGL Context
- **glo** – The OpenGL texture id

use(*unit: int = 0*) → None

Bind the texture to a channel,

Parameters

unit – The texture unit to bind the texture.

bind_to_image(*unit: int*, *read: bool = True*, *write: bool = True*, *level: int = 0*)

Bind textures to image units.

Note that either or both `read` and `write` needs to be `True`. The supported modes are: read only, write only, read-write

Parameters

- **unit** – The image unit
- **read** – The compute shader intends to read from this image
- **write** – The compute shader intends to write to this image
- **level** –

get_handle(*resident: bool = True*) → int

Get a handle for bindless texture access.

Once a handle is created its parameters cannot be changed. Attempting to do so will have no effect. (filter, wrap etc). There is no way to undo this immutability.

Handles cannot be used by shaders until they are resident. This method can be called multiple times to move a texture in and out of residency:

```
>> texture.get_handle(resident=False)
4294969856
>> texture.get_handle(resident=True)
4294969856
```

Ths same handle is returned if the handle already exists.

Note: Limitations from the OpenGL wiki

The amount of storage available for resident images/textures may be less than the total storage for textures that is available. As such, you should attempt to minimize the time a texture spends being resident. Do not attempt to take steps like making textures resident/unresident every frame or something. But if you are finished using a texture for some time, make it unresident.

Keyword Arguments

resident (*bool*) – Make the texture resident.

33.28.3 Buffer

class arcade.gl.**Buffer**(*ctx: Context, data: BufferProtocol | None = None, reserve: int = 0, usage: str = 'static'*)

Bases:

OpenGL buffer object. Buffers store byte data and upload it to graphics memory so shader programs can process the data. They are used for storage of vertex data, element data (vertex indexing), uniform block data etc.

The `data` parameter can be anything that implements the Buffer Protocol.

This includes `bytes`, `bytearray`, `array.array`, and more. You may need to use typing workarounds for non-builtin types. See *Writing Raw Bytes to GL Buffers & Textures* for more information.

Warning: Buffer objects should be created using *arcade.gl.Context.buffer()*

Parameters

- **ctx** – The context this buffer belongs to
- **data** – The data this buffer should contain. It can be a `bytes` instance or any object supporting the buffer protocol.
- **reserve** – Create a buffer of a specific byte size
- **usage** – A hit of this buffer is `static` or `dynamic` (can mostly be ignored)

`size`

The byte size of the buffer.

Type

int

`ctx`

The context this resource belongs to.

Type

`arcade.gl.Context`

`glo`

The OpenGL resource id

Type

gl.GLuint

`delete()`

Destroy the underlying OpenGL resource. Don't use this unless you know exactly what you are doing.

`static delete_glo`(*ctx:* Context, *glo: gl.GLuint*)

Release/delete open gl buffer. This is automatically called when the object is garbage collected.

`read`(*size: int = -1*, *offset: int = 0*) → bytes

Read data from the buffer.

Parameters

- **`size`** – The bytes to read. -1 means the entire buffer (default)
- **`offset`** – Byte read offset

`write`(*data: ByteString | memoryview | array | Array*, *offset: int = 0*)

Write byte data to the buffer from a buffer protocol object.

The `data` value can be anything that implements the Buffer Protocol.

This includes `bytes`, `bytearray`, `array.array`, and more. You may need to use typing workarounds for non-builtin types. See *Writing Raw Bytes to GL Buffers & Textures* for more information.

If the supplied data is larger than the buffer, it will be truncated to fit. If the supplied data is smaller than the buffer, the remaining bytes will be left unchanged.

Parameters

- **`data`** – The byte data to write. This can be bytes or any object supporting the buffer protocol.
- **`offset`** – The byte offset

`copy_from_buffer`(*source:* Buffer, *size=-1*, *offset=0*, *source_offset=0*)

Copy data into this buffer from another buffer

Parameters

- **`source`** – The buffer to copy from
- **`size`** – The amount of bytes to copy
- **`offset`** – The byte offset to write the data in this buffer
- **`source_offset`** – The byte offset to read from the source buffer

orphan(*size: int = -1*, *double: bool = False*)

Re-allocate the entire buffer memory. This can be used to resize a buffer or for re-specification (orphan the buffer to avoid blocking).

If the current buffer is busy in rendering operations it will be deallocated by OpenGL when completed.

Parameters

- **size** – New size of buffer. -1 will retain the current size.
- **double** – Is passed in with *True* the buffer size will be doubled

bind_to_uniform_block(*binding: int = 0*, *offset: int = 0*, *size: int = -1*)

Bind this buffer to a uniform block location. In most cases it will be sufficient to only provide a binding location.

Parameters

- **binding** – The binding location
- **offset** – byte offset
- **size** – size of the buffer to bind.

bind_to_storage_buffer(**, binding=0, offset=0, size=-1*)

Bind this buffer as a shader storage buffer.

Parameters

- **binding** – The binding location
- **offset** – Byte offset in the buffer
- **size** – The size in bytes. The entire buffer will be mapped by default.

33.28.4 BufferDescription

class arcade.gl.**BufferDescription**(*buffer: Buffer*, *formats: str*, *attributes: Sequence[str]*, *normalized: Iterable[str] | None = None*, *instanced: bool = False*)

Bases:

Buffer Object description used with `arcade.gl.Geometry`.

This class provides a Buffer object with a description of its content, allowing the a `Geometry` object to correctly map shader attributes to a program/shader.

The formats is a string providing the number and type of each attribute. Currently we only support f (float), i (integer) and B (unsigned byte).

`normalized` enumerates the attributes which must have their values normalized. This is useful for instance for colors attributes given as unsigned byte and normalized to floats with values between 0.0 and 1.0.

`instanced` allows this buffer to be used as instanced buffer. Each value will be used once for the whole geometry. The geometry will be repeated a number of times equal to the number of items in the Buffer.

Example:

```
# Describe my_buffer
# It contains two floating point numbers being a 2d position
# and two floating point numbers being texture coordinates.
# We expect the shader using this buffer to have an in_pos and in_uv attribute␣
```

(continues on next page)

(continued from previous page)

```
↪(exact name)
BufferDescription(
    my_buffer,
    '2f 2f',
    ['in_pos', 'in_uv'],
)
```

Parameters

- **buffer** – The buffer to describe
- **formats** – The format of each attribute
- **attributes** – List of attributes names (strings)
- **normalized** – list of attribute names that should be normalized
- **instanced** – `True` if this is per instance data

buffer: *Buffer*

The *Buffer* this description object describes

attributes

List of string attributes

normalized: Set[str]

List of normalized attributes

instanced: bool

Instanced flag (bool)

formats: List[AttribFormat]

Formats of each attribute

stride: int

The byte stride of the buffer

num_vertices: int

Number of vertices in the buffer

33.28.5 Geometry

Geometry Methods

arcade.gl.geometry.**quad_2d_fs**() → *Geometry*

Creates a screen aligned quad using normalized device coordinates

arcade.gl.geometry.**quad_2d**(*size: Tuple[float, float] = (1.0, 1.0), pos: Tuple[float, float] = (0.0, 0.0)*) → *Geometry*

Creates 2D quad Geometry using 2 triangle strip with texture coordinates.

Parameters

- **size** – width and height
- **pos** – Center position x and y

`arcade.gl.geometry.`**`screen_rectangle`**(*bottom_left_x: float*, *bottom_left_y: float*, *width: float*, *height: float*) → *Geometry*

Creates screen rectangle using 2 triangle strip with texture coordinates.

Parameters

- **`bottom_left_x`** – Bottom left x position
- **`bottom_left_y`** – Bottom left y position
- **`width`** – Width of the rectangle
- **`height`** – Height of the rectangle

`arcade.gl.geometry.`**`cube`**(*size: Tuple[float, float, float] = (1.0, 1.0, 1.0)*, *center: Tuple[float, float, float] = (0.0, 0.0, 0.0)*) → *Geometry*

Creates a cube with normals and texture coordinates.

Parameters

- **`size`** – size of the cube as a 3-component tuple
- **`center`** – center of the cube as a 3-component tuple

Returns
A cube

Geometry

`class` `arcade.gl.`**`Geometry`**(*ctx: Context*, *content: Sequence[BufferDescription] | None*, *index_buffer: Buffer | None = None*, *mode: int | None = None*, *index_element_size: int = 4*)

Bases:

A higher level abstraction of the VertexArray. It generates VertexArray instances on the fly internally matching the incoming program. This means we can render the same geometry with different programs as long as the *`Program`* and *`BufferDescription`* have compatible attributes.

Geometry objects should be created through *`arcade.gl.Context.geometry()`*

Parameters

- **`ctx`** – The context this object belongs to
- **`content`** – List of BufferDescriptions
- **`index_buffer`** – Index/element buffer
- **`mode`** – The default draw mode

`ctx`

The context this geometry belongs to.

Type
`Geometry`

`index_buffer`

Index/element buffer if supplied at creation.

Type
`Buffer`

num_vertices

Get or set the number of vertices. Be careful when modifying this properly and be absolutely sure what you are doing.

Type
int

append_buffer_description(*descr: BufferDescription*)

Append a new BufferDescription to the existing Geometry. .. Warning:: a Geometry cannot contain two BufferDescriptions which share an attribute name.

instance(*program: Program*) → *VertexArray*

Get the `arcade.gl.VertexArray` compatible with this program

render(*program: Program*, ***, *mode: c_uint | int | None = None*, *first: int = 0*, *vertices: int | None = None*, *instances: int = 1*) → None

Render the geometry with a specific program.

The geometry object will know how many vertices your buffers contains so overriding vertices is not needed unless you have a special case or have resized the buffers after the geometry instance was created.

Parameters

- **program** – The Program to render with
- **mode** – Override what primitive mode should be used
- **first** – Offset start vertex
- **vertices** – Override the number of vertices to render
- **instances** – Number of instances to render

render_indirect(*program: Program*, *buffer: Buffer*, ***, *mode: c_uint | int | None = None*, *count: int = -1*, *first: int = 0*, *stride: int = 0*)

Render the VertexArray to the framebuffer using indirect rendering.

Warning: This requires OpenGL 4.3

The following structs are expected for the buffer:

```
// Array rendering - no index buffer (16 bytes)
typedef  struct {
    uint  count;
    uint  instanceCount;
    uint  first;
    uint  baseInstance;
} DrawArraysIndirectCommand;

// Index rendering - with index buffer 20 bytes
typedef  struct {
    GLuint  count;
    GLuint  instanceCount;
    GLuint  firstIndex;
    GLuint  baseVertex;
    GLuint  baseInstance;
} DrawElementsIndirectCommand;
```

The `stride` is the byte stride between every redering command in the buffer. By default we assume this is 16 for array rendering (no index buffer) and 20 for indexed rendering (with index buffer)

Parameters

- **program** – The program to execute
- **buffer** – The buffer containing one or multiple draw parameters
- **mode** – Primitive type to render. TRIANGLES, LINES etc.
- **count** – The number if indirect draw calls to run. If omitted all draw commands in the buffer will be executed.
- **first** – The first indirect draw call to start on
- **stride** – The byte stride of the draw command buffer. Keep the default (0) if the buffer is tightly packed.

transform(*program: Program*, *buffer: Buffer | List[Buffer]*, ***, *first: int = 0*, *vertices: int | None = None*, *instances: int = 1*, *buffer_offset: int = 0*) → None

Render with transform feedback. Instead of rendering to the screen or a framebuffer the result will instead end up in the `buffer` we supply.

If a geometry shader is used the output primitive mode is automatically detected.

Parameters

- **program** – The Program to render with
- **buffer** (*Union[Buffer, Sequence[Buffer]]*) – The buffer(s) we transform into. This depends on the programs `varyings_capture_mode`. We can transform into one buffer interlaved or transform each attribute into separate buffers.
- **first** – Offset start vertex
- **vertices** – Number of vertices to render
- **instances** – Number of instances to render
- **buffer_offset** – Byte offset for the buffer

flush() → None

Flush all the internally generated VertexArrays.

The Geometry instance will store a VertexArray for every unique set of input attributes it stumbles over when redering and transform calls are issued. This data is usually pretty light weight and usually don't need flushing.

VertexArray

class arcade.gl.**VertexArray**(*ctx: Context*, *program: Program*, *content: Sequence[BufferDescription]*, *index_buffer: Buffer | None = None*, *index_element_size: int = 4*)

Bases:

Wrapper for Vertex Array Objects (VAOs).

This objects should not be instantiated from user code. Use `arcade.gl.Geometry` instead. It will create VAO instances for you automatically. There is a lot of complex interaction between programs and vertex arrays that will be done for you automatically.

`ctx`

The Context this object belongs to

Type
`arcade.gl.Context`

`program`

The assigned program

Type
`arcade.gl.Program`

`ibo`

Element/index buffer

Type
`arcade.gl.Buffer`

`num_vertices`

The number of vertices

Type
int

`delete()`

Destroy the underlying OpenGL resource. Don't use this unless you know exactly what you are doing.

`static delete_glo`(*ctx:* Context, *glo: gl.GLuint*)

Delete this object. This is automatically called when this object is garbage collected.

`render`(*mode: c_uint* | *int*, *first: int = 0*, *vertices: int = 0*, *instances: int = 1*)

Render the VertexArray to the currently active framebuffer.

Parameters

- **`mode`** – Primitive type to render. TRIANGLES, LINES etc.
- **`first`** – The first vertex to render from
- **`vertices`** – Number of vertices to render
- **`instances`** – OpenGL instance, used in using vertices over and over

`render_indirect`(*buffer:* Buffer, *mode: c_uint* | *int*, *count*, *first*, *stride*)

Render the VertexArray to the framebuffer using indirect rendering.

Warning: This requires OpenGL 4.3

Parameters

- **`buffer`** – The buffer containing one or multiple draw parameters
- **`mode`** – Primitive type to render. TRIANGLES, LINES etc.
- **`count`** – The number if indirect draw calls to run
- **`first`** – The first indirect draw call to start on
- **`stride`** – The byte stride of the draw command buffer. Keep the default (0) if the buffer is tightly packed.

transform_interleaved(*buffer: Buffer, mode: c_uint | int, output_mode: c_uint | int, first: int = 0, vertices: int = 0, instances: int = 1, buffer_offset=0*)

Run a transform feedback.

Parameters

- **buffer** – The buffer to write the output
- **mode** – The input primitive mode
- **output_mode** – The output primitive mode
- **first** – Offset start vertex
- **vertices** – Number of vertices to render
- **instances** – Number of instances to render
- **buffer_offset** – Byte offset for the buffer (target)

transform_separate(*buffers: List[Buffer], mode: c_uint | int, output_mode: c_uint | int, first: int = 0, vertices: int = 0, instances: int = 1, buffer_offset=0*)

Run a transform feedback writing to separate buffers.

Parameters

- **buffers** – The buffers to write the output
- **mode** – The input primitive mode
- **output_mode** – The output primitive mode
- **first** – Offset start vertex
- **vertices** – Number of vertices to render
- **instances** – Number of instances to render
- **buffer_offset** – Byte offset for the buffer (target)

`glo`

33.28.6 Framebuffer

FrameBuffer

class arcade.gl.**Framebuffer**(*ctx: Context, *, color_attachments=None, depth_attachment: Texture2D | None = None*)

Bases:

An offscreen render target also called a Framebuffer Object in OpenGL. This implementation is using texture attachments. When creating a Framebuffer we supply it with textures we want our scene rendered into. The advantage of using texture attachments is the ability we get to keep working on the contents of the framebuffer.

The best way to create framebuffer is through `arcade.gl.Context.framebuffer()`:

```python
# Create a 100 x 100 framebuffer with one attachment
ctx.framebuffer(color_attachments=[ctx.texture((100, 100), components=4)])

# Create a 100 x 100 framebuffer with two attachments
# Shaders can be configured writing to the different layers
```

(continues on next page)

(continued from previous page)

```
ctx.framebuffer(
    color_attachments=[
        ctx.texture((100, 100), components=4),
        ctx.texture((100, 100), components=4),
    ]
)
```

Parameters

- **ctx** – The context this framebuffer belongs to
- **color_attachments** – List of color attachments.
- **depth_attachment** – A depth attachment (optional)

is_default = False

Is this the default framebuffer? (window buffer)

glo

The OpenGL id/name of the framebuffer

Type
GLuint

viewport

Get or set the framebuffer's viewport. The viewport parameter are `(x, y, width, height)`. It determines what part of the framebuffer should be rendered to. By default the viewport is `(0, 0, width, height)`.

The viewport value is persistent all will automatically be applies every time the framebuffer is bound.

Example:

```
# 100, x 100 lower left with size 200 x 200px
fb.viewport = 100, 100, 200, 200
```

scissor

Get or set the scissor box for this framebuffer.

By default the scissor box is disabled and has no effect and will have an initial value of `None`. The scissor box is enabled when setting a value and disabled when set to `None`

# Set and enable scissor box only drawing # in a 100 x 100 pixel lower left area ctx.scissor = 0, 0, 100, 100 # Disable scissoring ctx.scissor = None

Type
tuple (x, y, width, height)

ctx

The context this object belongs to.

Type
arcade.gl.Context

width

The width of the framebuffer in pixels

Type
int

`height`
The height of the framebuffer in pixels

Type
int

`size`
Size as a `(w, h)` tuple

Type
tuple (int, int)

`samples`
Number of samples (MSAA)

Type
int

`color_attachments`
A list of color attachments

Type
list of `arcade.gl.Texture`

`depth_attachment`
Depth attachment

Type
`arcade.gl.Texture`

`depth_mask`
`True`). It determines if depth values should be written to the depth texture when depth testing is enabled.

The depth mask value is persistent all will automatically be applies every time the framebuffer is bound.

Type
bool

Type
Get or set the depth mask (default

`activate()`
Context manager for binding the framebuffer.

Unlike the default context manager in this class this support nested framebuffer binding.

`use(*, force: bool = False)`
Bind the framebuffer making it the target of all rendering commands

Parameters
force – Force the framebuffer binding even if the system already believes it's already bound.

`clear(color: Tuple[int, int, int] | Tuple[int, int, int, int] | Tuple[float, float, float] | Tuple[float, float, float, float] = (0.0, 0.0, 0.0, 0.0), *, depth: float = 1.0, normalized: bool = False, viewport: Tuple[int, int, int, int] | None = None)`
Clears the framebuffer:

```
# Clear the framebuffer using arcade's colors (not normalized)
fb.clear(color=arcade.color.WHITE)

# Clear framebuffer using the color red in normalized form
fbo.clear(color=(1.0, 0.0, 0.0, 1.0), normalized=True)
```

If the background color is an `RGB` value instead of `RGBA`` we assume alpha value 255.

Parameters

- **`color`** – A 3 or 4 component tuple containing the color
- **`depth`** – Value to clear the depth buffer (unused)
- **`normalized`** – If the color values are normalized or not
- **`viewport`** (*Tuple[int, int, int, int]*) – The viewport range to clear

`read`(***, *viewport=None*, *components=3*, *attachment=0*, *dtype='f1'*) → bytes

Read framebuffer pixels

Parameters

- **`viewport`** – The x, y, with, height to read
- **`components`** –
- **`attachment`** – The attachment id to read from
- **`dtype`** – The data type to read

Returns

pixel data as a bytearray

`resize`()

Detects size changes in attachments. This will reset the viewport to `0, 0, width, height`.

`delete`()

Destroy the underlying OpenGL resource. Don't use this unless you know exactly what you are doing.

`static delete_glo`(*ctx*, *framebuffer_id*)

Destroys the framebuffer object

Parameters

- **`ctx`** – OpenGL context
- **`framebuffer_id`** – Framebuffer to destroy (glo)

DefaultFrameBuffer

`class` `arcade.gl.framebuffer.`**`DefaultFrameBuffer`**(*ctx: Context*)

Bases: *`Framebuffer`*

Represents the default framebuffer. This is the framebuffer of the window itself and need some special handling.

We are not allowed to destroy this framebuffer since it's owned by pyglet. This framebuffer can also change size and pixel ratio at any point.

We're doing some initial introspection to guess somewhat sane initial values. Since this is a dynamic framebuffer we cannot trust the internal values. We can only trust what the pyglet window itself reports related to window size and framebuffer size. This should be updated in the `on_resize` callback.

is_default = True

Is this the default framebuffer? (window buffer)

viewport

Get or set the framebuffer's viewport. The viewport parameter are `(x, y, width, height)`. It determines what part of the framebuffer should be rendered to. By default the viewport is `(0, 0, width, height)`.

The viewport value is persistent all will automatically be applies every time the framebuffer is bound.

Example:

```
# 100, x 100 lower left with size 200 x 200px
fb.viewport = 100, 100, 200, 200
```

scissor

Get or set the scissor box for this framebuffer.

By default the scissor box is disabled and has no effect and will have an initial value of `None`. The scissor box is enabled when setting a value and disabled when set to `None`

# Set and enable scissor box only drawing # in a 100 x 100 pixel lower left area ctx.scissor = 0, 0, 100, 100 # Disable scissoring ctx.scissor = None

Type
tuple (x, y, width, height)

33.28.7 Query

class `arcade.gl.Query`(*ctx:* Context, *samples=True*, *time=True*, *primitives=True*)

Bases:

A query object to perform low level measurements of OpenGL rendering calls.

The best way to create a program instance is through *`arcade.gl.Context.query()`*

Example usage:

```
query = ctx.query()
with query:
    geometry.render(..)

print('samples_passed:', query.samples_passed)
print('time_elapsed:', query.time_elapsed)
print('primitives_generated:', query.primitives_generated)
```

ctx

The context this query object belongs to

Type
`arcade.gl.Context`

samples_passed

How many samples was written. These are per component (RGBA)

Type
int

`time_elapsed`

The time elapsed in nanoseconds

Type

int

`primitives_generated`

How many primitives a vertex or geometry shader processed. When using a geometry shader this only counts the primitives actually emitted.

Type

int

`delete()`

Destroy the underlying OpenGL resource. Don't use this unless you know exactly what you are doing.

`static delete_glo`(*ctx*, *glos*) → None

Delete this query object. This is automatically called when the object is garbage collected.

33.28.8 Program

Program

`class arcade.gl.`**`Program`**(*ctx: Context*, ***, *vertex_shader: str*, *fragment_shader: str | None = None*, *geometry_shader: str | None = None*, *tess_control_shader: str | None = None*, *tess_evaluation_shader: str | None = None*, *varyings: List[str] | None = None*, *varyings_capture_mode: str = 'interleaved'*)

Bases:

Compiled and linked shader program.

Currently supports vertex, fragment and geometry shaders. Transform feedback also supported when output attributes names are passed in the varyings parameter.

The best way to create a program instance is through *`arcade.gl.Context.program()`*

Access Uniforms via the `[]` operator. Example:

```
program['MyUniform'] = value
```

Parameters

- **`ctx`** – The context this program belongs to
- **`vertex_shader`** – vertex shader source
- **`fragment_shader`** – fragment shader source
- **`geometry_shader`** – geometry shader source
- **`tess_control_shader`** – tessellation control shader source
- **`tess_evaluation_shader`** – tessellation evaluation shader source
- **`varyings`** – List of out attributes used in transform feedback.
- **`varyings_capture_mode`** – The capture mode for transforms. `"interleaved"` means all out attribute will be written to a single buffer. `"separate"` means each out attribute will be written separate buffers. Based on these settings the *transform()* method will accept a single buffer or a list of buffer.

`attribute_key: str`

Internal cache key used with vertex arrays

`ctx`

The context this program belongs to

Type
arcade.gl.Context

`glo`

The OpenGL resource id for this program

Type
int

`attributes`

List of attribute information

`varyings`

Out attributes names used in transform feedback

Type
list of str

`out_attributes`

Out attributes names used in transform feedback.

Warning: Old alias for `varyings`. May be removed in the future.

Type
list of str

`varyings_capture_mode`

Get the capture more for transform feedback (single, multiple).

This is a read only property since capture mode can only be set before the program is linked.

`geometry_input`

The geometry shader's input primitive type. This an be compared with `GL_TRIANGLES`, `GL_POINTS` etc. and is queried when the program is created.

Type
int

`geometry_output`

The geometry shader's output primitive type. This an be compared with `GL_TRIANGLES`, `GL_POINTS` etc. and is queried when the program is created.

Type
int

`geometry_vertices`

The maximum number of vertices that can be emitted. This is queried when the program is created.

Type
int

delete()

Destroy the underlying OpenGL resource. Don't use this unless you know exactly what you are doing.

static **delete_glo**(*ctx*, *prog_id*)

Deletes a program. This is normally called automatically when the program is garbage collected.

Parameters

- **ctx** – The context
- **prog_id** – The OpenGL resource id

self[item] → *Uniform* | *UniformBlock*

Get a uniform or uniform block

self[key] = value

Set a uniform value

set_uniform_safe(*name: str*, *value: Any*)

Safely set a uniform catching KeyError.

Parameters

- **name** – The uniform name
- **value** – The uniform value

set_uniform_array_safe(*name: str*, *value: List[Any]*)

Safely set a uniform array. Arrays can be shortened by the glsl compiler not all elements are determined to be in use. This function checks the length of the actual array and sets a subset of the values if needed. If the uniform don't exist no action will be done.

Parameters

- **name** – Name of uniform
- **value** – List of values

use()

Activates the shader. This is normally done for you automatically.

static **compile_shader**(*source: str*, *shader_type: int*) → c_uint

Compile the shader code of the given type.

shader_type could be GL_VERTEX_SHADER, GL_FRAGMENT_SHADER, . . .

Returns the shader id as a GLuint

static **link**(*glo*)

Link a shader program

Program Members

Uniform

class arcade.gl.uniform.**Uniform**(*ctx*, *program_id*, *location*, *name*, *data_type*, *array_length*)

Bases:

A Program uniform

Parameters

- `ctx` – The context
- `program_id` – The program id to which this uniform belongs
- `location` – The uniform location
- `name` – The uniform name
- `data_type` – The data type of the uniform
- `array_length` – The array length of the uniform

`location`

The location of the uniform in the program

`name`

Name of the uniform

`array_length`

Length of the uniform array. If not an array 1 will be returned

`components`

How many components for the uniform. A vec4 will for example have 4 components.

`getter`

`setter`

UniformBlock

class arcade.gl.uniform.**UniformBlock**(*glo: int*, *index: int*, *size: int*, *name: str*)

Bases:

Wrapper for a uniform block in shaders.

`glo`

The OpenGL object handle

`index`

The index of the uniform block

`size`

The size of the uniform block

`name`

The name of the uniform block

`binding`

Get or set the binding index for this uniform block

`getter()`

The getter function for this uniform block. Returns self.

`setter`(*value: int*)

The setter function for this uniform block.

Parameters
`value` – The binding index to set.

33.28.9 Compute Shader

class arcade.gl.ComputeShader(*ctx: Context*, *glsl_source: str*)

Bases:

A higher level wrapper for an OpenGL compute shader.

glo

The name/id of the OpenGL resource

use()

Use/activate the compute shader.

Note: This is not necessary to call in normal use cases since `run()` already does this for you.

run(*group_x=1*, *group_y=1*, *group_z=1*) → None

Run the compute shader.

When running a compute shader we specify how many work groups should be executed on the `x`, `y` and `z` dimension. The size of the work group is defined in the compute shader.

```
// Work group with one dimension. 16 work groups executed.
layout(local_size_x=16) in;
// Work group with two dimensions. 256 work groups executed.
layout(local_size_x=16, local_size_y=16) in;
// Work group with three dimensions. 4096 work groups executed.
layout(local_size_x=16, local_size_y=16, local_size_z=16) in;
```

Group sizes are `1` by default. If your compute shader doesn't specify a size for a dimension or uses `1` as size you don't have to supply this parameter.

Parameters

- **group_x** – The number of work groups to be launched in the X dimension.
- **group_y** – The number of work groups to be launched in the y dimension.
- **group_z** – The number of work groups to be launched in the z dimension.

self**[**item**]** → *Uniform* | *UniformBlock*

Get a uniform or uniform block

self**[**key**]** = value

Set a uniform value

delete()

Destroy the internal compute shader object. This is normally not necessary, but depends on the garbage collection more configured in the context.

static delete_glo(*ctx*, *prog_id*)

Low level method for destroying a compute shader by id

33.28.10 Exceptions

`class arcade.gl.ShaderException`

Bases: `Exception`

Exception class for shader-specific problems.

33.29 GUI

`class arcade.gui.UIDraggableMixin`(**, x: float = 0, y: float = 0, width: float = 100, height: float = 100, children: Iterable[UIWidget] = (), size_hint=None, size_hint_min=None, size_hint_max=None, **kwargs*)

Bases: `UILayout`

UIDraggableMixin can be used to make any `UIWidget` draggable.

Example, create a draggable Frame, with a background, useful for window like constructs:

class DraggablePane(UITexturePane, UIDraggableMixin):
…

This does overwrite `UILayout` behaviour which position themselves, like `UIAnchorWidget`

do_layout()

on_event(*event*) → bool | None

`class arcade.gui.UIMouseFilterMixin`(**, x: float = 0, y: float = 0, width: float = 100, height: float = 100, children: Iterable[UIWidget] = (), size_hint=None, size_hint_min=None, size_hint_max=None, **kwargs*)

Bases: `UIWidget`

`UIMouseFilterMixin` can be used to catch all mouse events which occur inside this widget.

Useful for window like widgets, `UIMouseEvents` should not trigger effects which are under the widget.

on_event(*event*) → bool | None

`class arcade.gui.UIWindowLikeMixin`(**, x: float = 0, y: float = 0, width: float = 100, height: float = 100, children: Iterable[UIWidget] = (), size_hint=None, size_hint_min=None, size_hint_max=None, **kwargs*)

Bases: `UIMouseFilterMixin`, `UIDraggableMixin`, `UIWidget`

Makes a widget window like:

- handles all mouse events that occur within the widgets boundaries
- can be dragged

`class arcade.gui.Surface`(**, size: Tuple[int, int], position: Tuple[int, int] = (0, 0), pixel_ratio: float = 1.0*)

Bases:

Holds a `arcade.gl.Framebuffer` and abstracts the drawing on it. Used internally for rendering widgets.

activate()

Save and restore projection and activate Surface buffer to draw on. Also resets the limit of the surface (viewport).

clear(*color: Tuple[int, int, int, int] = (0, 0, 0, 0)*)

Clear the surface

draw(*area: Tuple[float, float, float, float] | List[float] | None = None*) → None

Draws the contents of the surface.

The surface will be rendered at the configured `position` and limited by the given `area`. The area can be out of bounds.

Parameters

area – Limit the area in the surface we're drawing (x, y, w, h)

draw_sprite(*x*, *y*, *width*, *height*, *sprite*)

Draw a sprite to the surface

draw_texture(*x: float*, *y: float*, *width: float*, *height: float*, *tex:* Texture | NinePatchTexture, *angle: float = 0.0*, *alpha: int = 255*)

limit(*x*, *y*, *width*, *height*)

Reduces the draw area to the given rect

resize(*, *size: Tuple[int, int]*, *pixel_ratio: float*) → None

Resize the internal texture by re-allocating a new one

Parameters

- **size** – The new size in pixels (xy)
- **pixel_ratio** – The pixel scale of the window

blend_func_render

Blend mode for when we're drawing the surface

blend_func_render_into

Blend modes for when we're drawing into the surface

height

pixel_ratio

position

Get or set the surface position

size

Size of the surface in window coordinates

size_scaled

The physical size of the buffer

width

class arcade.gui.**UIButtonRow**(*, *vertical: bool = False*, *align: str = 'center'*, *size_hint: ~typing.Any = (0, 0)*, *size_hint_min: ~typing.Any | None = None*, *size_hint_max: ~typing.Any | None = None*, *space_between: int = 10*, *style: ~typing.Any | None = None*, *button_factory: type = <class 'arcade.gui.widgets.buttons.UIFlatButton'>*)

Bases: `UIBoxLayout`

Places buttons in a row. :param vertical: Whether the button row is vertical or not. :param align: Where to align the button row. :param size_hint: Tuple of floats (0.0 - 1.0) of how much space of the parent should be requested. :param size_hint_min: Min width and height in pixel. :param size_hint_max: Max width and height in pixel. :param space_between: The space between the children. :param style: Not used. :param Tuple[str, . . .] button_labels: The labels for the buttons. :param callback: The callback function which will receive the text of the clicked button.

add_button(*label: str*, ***, *style=None*, *multiline=False*)

on_action(*event:* UIOnActionEvent)

class arcade.gui.**UIMessageBox**(***, *width: float*, *height: float*, *message_text: str*, *buttons=('Ok',)*)

Bases: `UIMouseFilterMixin`, `UIAnchorLayout`

A simple dialog box that pops up a message with buttons to close. Subclass this class or overwrite the 'on_action' event handler with

```
box = UIMessageBox(...)
@box.event("on_action")
def on_action(event: UIOnActionEvent):
    pass
```

Parameters

- **width** – Width of the message box
- **height** – Height of the message box
- **message_text** – Text to show as message to the user
- **buttons** – List of strings, which are shown as buttons

on_action(*event:* UIOnActionEvent)

Called when button was pressed

class arcade.gui.**UIManager**(*window:* Window | *None = None*)

Bases: `EventDispatcher`

UIManager is the central component within Arcade's GUI system. Handles window events, layout process and rendering.

To process window events, *`UIManager.enable()`* has to be called, which will inject event callbacks for all window events and redirects them through the widget tree.

If used within a view *`UIManager.enable()`* should be called from `View.on_show_view()` and *`UIManager.disable()`* should be called from `View.on_hide_view()`

Supports *size_hint* to grow/shrink direct children dependent on window size. Supports *size_hint_min* to ensure size of direct children (e.g. UIBoxLayout). Supports *size_hint_max* to ensure size of direct children (e.g. UIBoxLayout).

```
class MyView(arcade.View):
    def __init__():
        super().__init__()
        manager = UIManager()

        manager.add(Dummy())

    def on_show_view(self):
        # Set background color
        self.window.background_color = arcade.color.DARK_BLUE_GRAY
```

(continues on next page)

(continued from previous page)

```
        # Enable UIManager when view is shown to catch window events
        self.ui.enable()

    def on_hide_view(self):
        # Disable UIManager when view gets inactive
        self.ui.disable()

    def on_draw():
        self.clear()

        ...

        manager.draw() # draws the UI on screen
```

add(*widget: W*, ***, *index=None*, *layer=0*) → W

Add a widget to the *UIManager*. Added widgets will receive ui events and be rendered.

By default the latest added widget will receive ui events first and will be rendered on top of others.

The UIManager supports layered setups, widgets added to a higher layer are drawn above lower layers and receive events first. The layer 10 is reserved for overlaying components like dropdowns or tooltips.

Parameters

- **widget** – widget to add
- **index** – position a widget is added, None has the highest priority
- **layer** – layer which the widget should be added to, higher layer are above

Returns

the widget

adjust_mouse_coordinates(*x*, *y*)

This method is used, to translate mouse coordinates to coordinates respecting the viewport and projection of cameras. The implementation should work in most common cases.

If you use scrolling in the *arcade.Camera* you have to reset scrolling or overwrite this method using the camera conversion:

```
ui_manager.adjust_mouse_coordinates = camera.mouse_coordinates_to_world
```

clear()

Remove all widgets from UIManager

debug()

Walks through all widgets of a UIManager and prints out the rect

disable() → None

Remove handler functions (*on_…*) from *arcade.Window*

If every *arcade.View* uses its own *arcade.gui.UIManager*, this method should be called in *arcade.View.on_hide_view()*.

dispatch_ui_event(*event*)

draw() → None

`enable`() → None

Registers handler functions (*on_...*) to `arcade.gui.UIElement`

on_draw is not registered, to provide full control about draw order, so it has to be called by the devs themselves.

Within a view, this method should be called from *`arcade.View.on_show_view()`*.

`get_widgets_at`(*pos: ~typing.Tuple[float, float], cls: ~typing.Type[~arcade.gui.ui_manager.W] = <class 'arcade.gui.widgets.UIWidget'>, layer=0*) → Iterable[W]

Yields all widgets containing a position, returns first top laying widgets which is instance of cls.

Parameters

- **`pos`** – Pos within the widget bounds
- **`cls`** – class which the widget should be an instance of
- **`layer`** – layer to search, None will search through all layers

Returns

iterator of widgets of given type at position

`on_event`(*event*) → bool | None

`on_key_press`(*symbol: int, modifiers: int*)

`on_key_release`(*symbol: int, modifiers: int*)

`on_mouse_drag`(*x: float, y: float, dx: float, dy: float, buttons: int, modifiers: int*)

`on_mouse_motion`(*x: float, y: float, dx: float, dy: float*)

`on_mouse_press`(*x: float, y: float, button: int, modifiers: int*)

`on_mouse_release`(*x: float, y: float, button: int, modifiers: int*)

`on_mouse_scroll`(*x, y, scroll_x, scroll_y*)

`on_resize`(*width, height*)

`on_text`(*text*)

`on_text_motion`(*motion*)

`on_text_motion_select`(*motion*)

`on_update`(*time_delta*)

`remove`(*child: UIWidget*)

Removes the given widget from UIManager.

Parameters

`child` – widget to remove

`trigger_render`()

Request rendering of all widgets

walk_widgets(***, *root: UIWidget | None = None*, *layer=0*) → Iterable[*UIWidget*]

walks through widget tree, in reverse draw order (most top drawn widget first)

Parameters

- **root** – root widget to start from, if None, the layer is used
- **layer** – layer to search, None will search through all layers

OVERLAY_LAYER = 10

camera

Camera used when drawing the UI

rect

class arcade.gui.NinePatchTexture(***, *left: int*, *right: int*, *bottom: int*, *top: int*, *texture: Texture*, *atlas: TextureAtlas | None = None*)

Bases:

Keeps borders & corners at constant widths while stretching the middle.

It can be used with new or existing `UIWidget` subclasses wherever an ordinary `arcade.Texture` is supported. This is useful for GUI elements which must grow or shrink while keeping their border decorations constant, such as dialog boxes or text boxes.

The diagram below explains the stretching behavior of this class:

- Numbered regions with arrows (<--->) stretch along the direction(s) of any arrows present
- Bars (`|---|`) mark the distances specified by the border parameters (`left`, etc)

Listing 1: Stretch Axes & Border Parameters

```
left                          right
|------|                 |------|
                                       top
+------+-----------------+------+      ---
| (1)  | (2)             | (3)  |       |
|      | <-------------> |      |       |
+------+-----------------+------+      ---
| (4)  | (5)    ^        | (6)  |
|  ^   |        |        |   ^  |
|  |   |        |        |   |  |
|  |   | <------+------> |   |  |
|  |   |        |        |   |  |
|  |   |        |        |   |  |
|  v   |        v        |   v  |
+------+-----------------+------+      ---
| (7)  | (8)             | (9)  |       |
|      | <-------------> |      |       |
+------+-----------------+------+      ---
                                     bottom
```

As the texture is stretched, the numbered slices of the texture behave as follows:

- Areas `(1)`, `(3)`, `(7)` and `(9)` never stretch.
- Area `(5)` stretches both horizontally and vertically.
- Areas `(2)` and `(8)` only stretch horizontally.

- Areas `(4)` and `(6)` only stretch vertically.

Parameters

- **`left`** – The width of the left border of the 9-patch (in pixels)
- **`right`** – The width of the right border of the 9-patch (in pixels)
- **`bottom`** – The height of the bottom border of the 9-patch (in pixels)
- **`top`** – The height of the top border of the 9-patch (in pixels)
- **`texture`** – The raw texture to use for the 9-patch
- **`atlas`** – Specify an atlas other than arcade's default texture atlas

`draw_sized`(***, *position: Tuple[float, float] = (0.0, 0.0)*, *size: Tuple[float, float]*, *pixelated: bool = False*, ***kwargs*)

Draw the 9-patch texture with a specific size.

> **Warning:** This method assumes the passed dimensions are proper!
>
> Unexpected behavior may occur if you specify a size smaller than the total size of the border areas.

Parameters

- **`position`** – Bottom left offset of the texture in pixels
- **`size`** – Size of the 9-patch as width, height in pixels
- **`pixelated`** – Whether to draw with nearest neighbor interpolation

`bottom`

Get or set the bottom border of the 9-patch.

`ctx`

The OpenGL context.

`height`

The height of the texture in pixels.

`left`

Get or set the left border of the 9-patch.

`program`

Get or set the shader program.

Returns the default shader if no other shader is assigned.

`right`

Get or set the right border of the 9-patch.

`size`

The size of texture as a width, height tuple in pixels.

`texture`

Get or set the texture.

`top`

Get or set the top border of the 9-patch.

`width`

The width of the texture in pixels.

33.30 GUI Widgets

class `arcade.gui.UIImage`(*, *texture: Texture | NinePatchTexture*, ***kwargs*)

Bases: *UIWidget*

UIWidget showing a texture.

`do_render`(*surface: Surface*)

`texture`: *Texture* | *NinePatchTexture*

An observable property which triggers observers when changed.

Parameters

- `default` – Default value which is returned, if no value set before
- `default_factory` – A callable which returns the default value. Will be called with the property and the instance

class `arcade.gui.UISlider`(*, *value: float = 0*, *min_value: float = 0*, *max_value: float = 100*, *x: float = 0*, *y: float = 0*, *width: float = 300*, *height: float = 20*, *size_hint=None*, *size_hint_min=None*, *size_hint_max=None*, *style: Mapping[str, UISliderStyle] | None = None*, ***kwargs*)

Bases: *UIStyledWidget*[*UISliderStyle*]

A simple horizontal slider. The value of the slider can be set by moving the cursor(indicator).

There are four states of the UISlider i.e normal, hovered, pressed and disabled.

Parameters

- `value` – Current value of the curosr of the slider.
- `min_value` – Minimum value of the slider.
- `max_value` – Maximum value of the slider.
- `x` – x coordinate of bottom left.
- `y` – y coordinate of bottom left.
- `width` – Width of the slider.
- `height` – Height of the slider.
- `style` (*Mapping[str, "UISlider.UIStyle"] | None*) – Used to style the slider for different states.

`UIStyle`

alias of *UISliderStyle*

`do_render`(*surface: Surface*)

`get_current_state()` → str

Returns the current state of the slider i.e disabled, press, hover or normal.

`on_change(`*event:* UIOnChangeEvent`)`

To be implemented by the user, triggered when the cursor's value is changed.

`on_event(`*event:* UIEvent`)` → bool | None

```
DEFAULT_STYLE = {'disabled': UISliderStyle(bg=Color(r=94, g=104, b=117, a=255),
border=Color(r=77, g=81, b=87, a=255), border_width=1, filled_bar=Color(r=50, g=50,
b=50, a=255), unfilled_bar=Color(r=116, g=125, b=123, a=255)), 'hover':
UISliderStyle(bg=Color(r=96, g=103, b=112, a=255), border=Color(r=77, g=81, b=87,
a=255), border_width=2, filled_bar=Color(r=50, g=50, b=50, a=255),
unfilled_bar=Color(r=116, g=125, b=123, a=255)), 'normal':
UISliderStyle(bg=Color(r=94, g=104, b=117, a=255), border=Color(r=77, g=81, b=87,
a=255), border_width=1, filled_bar=Color(r=50, g=50, b=50, a=255),
unfilled_bar=Color(r=116, g=125, b=123, a=255)), 'press':
UISliderStyle(bg=Color(r=96, g=103, b=112, a=255), border=Color(r=77, g=81, b=87,
a=255), border_width=3, filled_bar=Color(r=50, g=50, b=50, a=255),
unfilled_bar=Color(r=116, g=125, b=123, a=255))}
```

`disabled`

An observable property which triggers observers when changed.

Parameters

- **`default`** – Default value which is returned, if no value set before
- **`default_factory`** – A callable which returns the default value. Will be called with the property and the instance

`hovered`

An observable property which triggers observers when changed.

Parameters

- **`default`** – Default value which is returned, if no value set before
- **`default_factory`** – A callable which returns the default value. Will be called with the property and the instance

`norm_value`

Normalized value between 0.0 and 1.0

`parent:` *`UIManager`* `|` *`UIWidget`* `|` `None`

`pressed`

An observable property which triggers observers when changed.

Parameters

- **`default`** – Default value which is returned, if no value set before
- **`default_factory`** – A callable which returns the default value. Will be called with the property and the instance

`value`

An observable property which triggers observers when changed.

Parameters

- **`default`** – Default value which is returned, if no value set before

- **default_factory** – A callable which returns the default value. Will be called with the property and the instance

value_x

Returns the current value of the cursor of the slider.

class arcade.gui.**UISliderStyle**(*bg: Tuple[int, int, int, int] = (94, 104, 117, 255), border: Tuple[int, int, int, int] = (77, 81, 87, 255), border_width: int = 1, filled_bar: Tuple[int, int, int, int] = (50, 50, 50, 255), unfilled_bar: Tuple[int, int, int, int] = (116, 125, 123, 255)*)

Bases: *UIStyleBase*

Used to style the slider for different states. Below is its use case.

```
button = UITextureButton(style={"normal": UITextureButton.UIStyle(...),})
```

bg: Tuple[int, int, int, int] = (94, 104, 117, 255)

border: Tuple[int, int, int, int] = (77, 81, 87, 255)

border_width: int = 1

filled_bar: Tuple[int, int, int, int] = (50, 50, 50, 255)

unfilled_bar: Tuple[int, int, int, int] = (116, 125, 123, 255)

class arcade.gui.**UIAnchorLayout**(**, x: float = 0, y: float = 0, width: float = 1, height: float = 1, children: Iterable[UIWidget] = (), size_hint=(1, 1), size_hint_min=None, size_hint_max=None, **kwargs*)

Bases: *UILayout*

Places children based on anchor values.

Defaults to `size_hint = (1, 1)`.

Supports the options `size_hint`, `size_hint_min`, and `size_hint_max`. Children are allowed to overlap.

Child are resized based on `size_hint`. `size_hint_min/max` only take effect if a `size_hint` is set.

Allowed keyword options for *add()*:

- `anchor_x`: `str` = None

  Horizontal anchor position for the layout. The class constant *default_anchor_x* is used as default.

- `anchor_y`: `str` = None

  Vertical anchor position for the layout. The class constant *default_anchor_y* is used as default.

- `align_x`: `float` = 0

  Horizontal alignment for the layout.

- `align_y`: `float` = 0

  Vertical alignement for the layout.

add(*child: W, *, anchor_x: str | None = None, align_x: float = 0, anchor_y: str | None = None, align_y: float = 0, **kwargs*) → W

Add a widget to the layout as a child. Added widgets will receive all user-interface events and be rendered.

By default, the latest added widget will receive events first and will be rendered on top of others. The widgets will be automatically placed within this widget.

Parameters

- **`child`** – Specified child widget to add.
- **`anchor_x`** – Horizontal anchor. Valid options are `left`, `right`, and `center`.
- **`align_x`** – Offset or padding for the horizontal anchor.
- **`anchor_y`** – Vertical anchor. Valid options are `top`, `center`, and `bottom`.
- **`align_y`** – Offset or padding for the vertical anchor.

Returns

Given child that was just added to the layout.

`do_layout()`

`default_anchor_x = 'center'`

`default_anchor_y = 'center'`

`class arcade.gui.`**`UIBoxLayout`**`(`**, x=0, y=0, width=1, height=1, vertical=True, align='center', children: Iterable[UIWidget] = (), size_hint=(0, 0), size_hint_max=None, space_between=0, style=None, **kwargs*`)`

Bases: *`UILayout`*

Place widgets next to each other. Depending on the `vertical` attribute, the widgets are placed top to bottom or left to right.

Hint: *`UIBoxLayout`* does not adjust its own size if children are added. This requires a *`UIManager`* or a *`UIAnchorLayout`* as a parent.

Or use *`arcade.gui.UIBoxLayout.fit_content()`* to resize the layout. The bottom-left corner is used as the default anchor point.

Supports the options: `size_hint`, `size_hint_min`, `size_hint_max`. `size_hint_min` is automatically updated based on the minimal required space by children.

If a child widget provides a `size_hint` for a dimension, the child will be resized within the given range of `size_hint_min` and `size_hint_max` (unrestricted if not given). If the parameter `vertical` is True, any available space (`layout size - min_size` of children) will be distributed to the child widgets based on their `size_hint`.

Parameters

- **`x`** – x coordinate of the bottom left corner.
- **`y`** – y coordinate of the bottom left corner.
- **`vertical`** – Layout children vertical (True) or horizontal (False).
- **`align`** – Align children in orthogonal direction:: - `x`: `left`, `center`, and `right` - `y`: `top`, `center`, and `bottom`
- **`children`** – Initial list of children. More can be added later.
- **`size_hint`** – Size hint for the *`UILayout`* if the widget would like to grow. Defaults to `0, 0` -> minimal size to contain children.
- **`size_hint_max`** – Maximum width and height in pixels.
- **`space_between`** – Space in pixels between the children.

do_layout()

fit_content()

Resize the layout to fit the content. This will take the minimal required size into account.

class arcade.gui.**UIGridLayout**(**, x=0, y=0, align_horizontal='center', align_vertical='center', children: Iterable[UIWidget] = (), size_hint=(0, 0), size_hint_max=None, horizontal_spacing: int = 0, vertical_spacing: int = 0, column_count: int = 1, row_count: int = 1, style=None, **kwargs*)

Bases: `UILayout`

Place widgets in a grid layout. This is similar to tkinter's `grid` layout geometry manager.

Defaults to `size_hint = (0, 0)`.

Supports the options `size_hint`, `size_hint_min`, and `size_hint_max`.

Children are resized based on `size_hint`. Maximum and minimum `size_hint``s only take effect if a ``size_hint` is given. `size_hint_min` is automatically updated based on the minimal required space by children.

Parameters

- **x** – x coordinate of bottom left corner.
- **y** – y coordinate of bottom left corner.
- **align_horizontal** – Align children in orthogonal direction. Options include `left`, `center`, and `right`.
- **align_vertical** – Align children in orthogonal direction. Options include `top`, `center`, and `bottom`.
- **children** – Initial list of children. More can be added later.
- **size_hint** – A size hint for `UILayout`, if the `UIWidget` would like to grow.
- **size_hint_max** – Maximum width and height in pixels.
- **horizontal_spacing** – Space between columns.
- **vertical_spacing** – Space between rows.
- **column_count** – Number of columns in the grid. This can be changed later.
- **row_count** – Number of rows in the grid. This can be changed later.

add(*child: W, col_num: int = 0, row_num: int = 0, col_span: int = 1, row_span: int = 1, **kwargs*) → W

Add a widget to the grid layout.

Parameters

- **child** – Specified child widget to add.
- **col_num** – Column index in which the widget is to be added. The first column is numbered 0; which is the top left corner.
- **row_num** – The row number in which the widget is to be added. The first row is numbered 0; which is the
- **col_span** – Number of columns the widget will stretch for.
- **row_span** – Number of rows the widget will stretch for.

do_layout()

class arcade.gui.**UIDropdown**(**, x: float = 0, y: float = 0, width: float = 100, height: float = 100, default: str | None = None, options: List[str | None] | None = None, style=None, **kwargs*)

Bases: `UILayout`

A dropdown layout. When clicked displays a list of options provided.

Triggers an event when an option is clicked, the event can be read by

```
dropdown = Dropdown()

@dropdown.event()
def on_change(event: UIOnChangeEvent):
    print(event.old_value, event.new_value)
```

Parameters

- **x** – x coordinate of bottom left
- **y** – y coordinate of bottom left
- **width** – Width of each of the option.
- **height** – Height of each of the option.
- **default** – The default value shown.
- **options** – The options displayed when the layout is clicked.
- **style** – Used to style the dropdown.

do_layout()

on_change(*event: UIOnChangeEvent*)

To be implemented by the user, triggered when the current selected value is changed to a different option.

DIVIDER = None

value

Current selected option.

class arcade.gui.**UIInputText**(**, x: float = 0, y: float = 0, width: float = 100, height: float = 24, text: str = '', font_name=('Arial',), font_size: float = 12, text_color: Tuple[int, int, int] | Tuple[int, int, int, int] = (0, 0, 0, 255), multiline=False, caret_color: Tuple[ChannelType, ChannelType, ChannelType] = (0, 0, 0), size_hint=None, size_hint_min=None, size_hint_max=None, **kwargs*)

Bases: `UIWidget`

An input field the user can type text into. This is useful in returning string input from the user. A caret is displayed, which the user can move around with a mouse or keyboard.

A mouse drag selects text, a mouse press moves the caret, and keys can move around the caret. Arcade confirms that the field is active before allowing users to type, so it is okay to have multiple of these.

Parameters

- **x** – x position (default anchor is bottom-left).
- **y** – y position (default anchor is bottom-left).
- **width** – Width of the text field.
- **height** – Height of the text field.

- **text** – Initial text displayed. This can be modified later programmatically or by the user's interaction with the caret.
- **font_name** – A list of fonts to use. Arcade will start at the beginning of the tuple and keep trying to load fonts until success.
- **font_size** – Font size of font.
- **text_color** – Color of the text.
- **multiline** – If enabled, a \n will start a new line. A `UITextWidget` `multiline` of True is the same thing as a `UITextArea`.
- **caret_color** – RGB color of the caret.
- **size_hint** – A tuple of floats between 0 and 1 defining the amount of space of the parent should be requested.
- **size_hint_min** – Minimum size hint width and height in pixel.
- **size_hint_max** – Maximum size hint width and height in pixel.
- **style** – Style has not been implemented for this widget, however it will be added in the near future.

do_render(*surface: Surface*)

on_event(*event: UIEvent*) → bool | None

on_update(*dt*)

LAYOUT_OFFSET = 1

text

class arcade.gui.**UILabel**(*text: str = '', *, x: float = 0, y: float = 0, width: float | None = None, height: float | None = None, font_name=('Arial',), font_size: float = 12, text_color: Tuple[int, int, int] | Tuple[int, int, int, int] = (255, 255, 255, 255), bold=False, italic=False, align='left', multiline: bool = False, size_hint=None, size_hint_min=None, size_hint_max=None, **kwargs*)

Bases: `UIWidget`

A simple text label. This widget is meant to display user instructions or information. This label supports multiline text.

If you want to make a scrollable viewing text box, use a `UITextArea`.

By default, a label will fit its initial content. If the text is changed use `fit_content()` to adjust the size.

Parameters

- **text** – Text displayed on the label.
- **x** – x position (default anchor is bottom-left).
- **y** – y position (default anchor is bottom-left).
- **width** – Width of the label. Defaults to text width if not specified. See `content_width()`.
- **height** – Height of the label. Defaults to text height if not specified. See `content_height()`.
- **font_name** – A list of fonts to use. Arcade will start at the beginning of the tuple and keep trying to load fonts until success.

- **font_size** – Font size of font.
- **text_color** – Color of the text.
- **bold** – If enabled, the label's text will be in a **bold** style.
- **italic** – If enabled, the label's text will be in an *italic* style.
- **stretch** – Stretch font style.
- **align** – Horizontal alignment of text on a line. This only applies if a width is supplied. Valid options include `"left"`, `"center"` or `"right"`.
- **dpi** – Resolution of the fonts in the layout. Defaults to 96.
- **multiline** – If enabled, a `\n` will start a new line. A *UITextWidget* with `multiline` of True is the same thing as a *UITextArea*.
- **size_hint** – A tuple of floats between 0 and 1 defining the amount of space of the parent should be requested.
- **size_hint_min** – Minimum size hint width and height in pixel.
- **size_hint_max** – Maximum size hint width and height in pixel.
- **style** – Not used. Labels will have no need for a style; they are too simple (just a text display).

do_render(*surface: Surface*)

fit_content()

Set the width and height of the label to contain the whole text.

text

class arcade.gui.**UITextArea**(**, x: float = 0, y: float = 0, width: float = 400, height: float = 40, text: str = '', font_name=('Arial',), font_size: float = 12, text_color: Tuple[int, int, int, int] = (255, 255, 255, 255), multiline: bool = True, scroll_speed: float | None = None, size_hint=None, size_hint_min=None, size_hint_max=None, **kwargs*)

Bases: *UIWidget*

A text area that allows users to view large documents of text by scrolling the mouse.

Parameters

- **x** – x position (default anchor is bottom-left).
- **y** – y position (default anchor is bottom-left).
- **width** – Width of the text area.
- **height** – Height of the text area.
- **text** – Initial text displayed.
- **font_name** – A list of fonts to use. Arcade will start at the beginning of the tuple and keep trying to load fonts until success.
- **font_size** – Font size of font.
- **text_color** – Color of the text.
- **multiline** – If enabled, a `\n` will start a new line.
- **scroll_speed** – Speed of mouse scrolling.

- **size_hint** – A tuple of floats between 0 and 1 defining the amount of space of the parent should be requested.
- **size_hint_min** – Minimum size hint width and height in pixel.
- **size_hint_max** – Maximum size hint width and height in pixel.
- **style** – Style has not been implemented for this widget, however it will be added in the near future.

do_render(*surface:* Surface)

fit_content()

Set the width and height of the text area to contain the whole text.

on_event(*event:* UIEvent) → bool | None

text

class arcade.gui.**UITextWidget**(*, *text:* *str*, *multiline:* *bool* = *False*, ***kwargs*)

Bases: `UIAnchorLayout`

Adds the ability to add text to a widget. Use this to create subclass widgets, which have text.

The text can be placed within the widget using `UIAnchorLayout` parameters with `place_text()`.

place_text(*anchor_x:* *str* | *None* = *None*, *align_x:* *float* = *0*, *anchor_y:* *str* | *None* = *None*, *align_y:* *float* = *0*, ***kwargs*)

Place widget's text within the widget using `UIAnchorLayout` parameters.

label

multiline

Get or set the multiline mode.

Newline characters ("\n") will only be honored when this is set to `True`. If you want a scrollable text widget, please use `UITextArea` instead.

parent: `UIManager` | `UIWidget` | `None`

text

Text of the widget. Modifying this repeatedly will cause significant lag; calculating glyph position is very expensive.

ui_label

Internal py:class:~*arcade.gui.UILabel* used for rendering the text.

class arcade.gui.**Rect**(*x:* *float*, *y:* *float*, *width:* *float*, *height:* *float*)

Bases: `NamedTuple`

Representing a rectangle for GUI module. Rect is idempotent.

Bottom left corner is used as fix point (x, y)

repr(self)

Return a nicely formatted representation string

align_bottom(*value:* *float*) → *Rect*

Returns new Rect, which is aligned to the bottom

align_center(*center_x*, *center_y*)

Returns new Rect, which is aligned to the center x and y

align_center_x(*value: float*) → *Rect*

Returns new Rect, which is aligned to the center_x

align_center_y(*value: float*) → *Rect*

Returns new Rect, which is aligned to the center_y

align_left(*value: float*) → *Rect*

Returns new Rect, which is aligned to the left

align_right(*value: float*) → *Rect*

Returns new Rect, which is aligned to the right

align_top(*value: float*) → *Rect*

Returns new Rect, which is aligned to the top

collide_with_point(*x*, *y*)

max_size(*width: float | None = None*, *height: float | None = None*)

Limits the size to the given max values.

min_size(*width=None*, *height=None*)

Sets the size to at least the given min values.

move(*dx: float = 0*, *dy: float = 0*)

Returns new Rect which is moved by dx and dy

resize(*width=None*, *height=None*)

Returns a rect with changed width and height. Fix x and y coordinate.

scale(*scale: float*) → *Rect*

Returns a new rect with scale applied

union(*rect:* Rect)

Returns a new Rect that is the union of this rect and another. The union is the smallest rectangle that contains theses two rectangles.

bottom

center

center_x

center_y

height: float

Alias for field number 3

left

position

Bottom left coordinates

right

size

top

width: float

Alias for field number 2

x: float

Alias for field number 0

y: float

Alias for field number 1

class arcade.gui.**UIDummy**(**, x=0, y=0, width=100, height=100, size_hint=None, size_hint_min=None, size_hint_max=None, **kwargs*)

Bases: `UIInteractiveWidget`

Solid color widget used for testing & examples

It should not be subclassed for real-world usage.

When clicked, it does the following:

- Outputs its *rect* attribute to the console
- Changes its color to a random fully opaque color

Parameters

- **x** – x coordinate of bottom left
- **y** – y coordinate of bottom left
- **color** – fill color for the widget
- **width** – width of widget
- **height** – height of widget
- **size_hint** – Tuple of floats (0.0-1.0), how much space of the parent should be requested
- **size_hint_min** – min width and height in pixel
- **size_hint_max** – max width and height in pixel
- **style** – not used

do_render(*surface: Surface*)

on_click(*event: UIOnClickEvent*)

on_update(*dt*)

class arcade.gui.**UIInteractiveWidget**(**, x: float = 0, y: float = 0, width: float, height: float, size_hint=None, size_hint_min=None, size_hint_max=None, **kwargs*)

Bases: `UIWidget`

Base class for widgets which use mouse interaction (hover, pressed, clicked)

Parameters

- **x** – x coordinate of bottom left
- **y** – y coordinate of bottom left
- **width** – width of widget

- **height** – height of widget
- **size_hint** – Tuple of floats (0.0-1.0), how much space of the parent should be requested
- **size_hint_min** – min width and height in pixel
- **size_hint_max** – max width and height in pixel:param x: center x of widget
- **style** – not used

on_click(*event: UIOnClickEvent*)

on_event(*event: UIEvent*) → bool | None

disabled

An observable property which triggers observers when changed.

Parameters

- **default** – Default value which is returned, if no value set before
- **default_factory** – A callable which returns the default value. Will be called with the property and the instance

hovered

An observable property which triggers observers when changed.

Parameters

- **default** – Default value which is returned, if no value set before
- **default_factory** – A callable which returns the default value. Will be called with the property and the instance

pressed

An observable property which triggers observers when changed.

Parameters

- **default** – Default value which is returned, if no value set before
- **default_factory** – A callable which returns the default value. Will be called with the property and the instance

class arcade.gui.**UILayout**(**, x: float = 0, y: float = 0, width: float = 100, height: float = 100, children: Iterable[UIWidget] = (), size_hint=None, size_hint_min=None, size_hint_max=None, **kwargs*)

Bases: `UIWidget`

Base class for widgets, which position themselves or their children.

Parameters

- **x** – x coordinate of bottom left
- **y** – y coordinate of bottom left
- **width** – width of widget
- **height** – height of widget
- **children** – Child widgets of this group
- **size_hint** – A hint for `UILayout`, if this `UIWidget` would like to grow
- **size_hint** – Tuple of floats (0.0-1.0), how much space of the parent should be requested

- **size_hint_min** – min width and height in pixel
- **size_hint_max** – max width and height in pixel
- **style** – not used

do_layout()

Triggered by the UIManager before rendering, *UILayout* s should place themselves and/or children. Do layout will be triggered on children afterwards.

Use *UIWidget.trigger_render()* to trigger a rendering before the next frame, this will happen automatically if the position or size of this widget changed.

class arcade.gui.**UISpace**(***, *x=0*, *y=0*, *width=100*, *height=100*, *color=(0, 0, 0, 0)*, *size_hint=None*, *size_hint_min=None*, *size_hint_max=None*, ***kwargs*)

Bases: *UIWidget*

Widget reserving space, can also have a background color.

Parameters

- **x** – x coordinate of bottom left
- **y** – y coordinate of bottom left
- **width** – width of widget
- **height** – height of widget
- **color** – Color for widget area
- **size_hint** – Tuple of floats (0.0-1.0), how much space of the parent should be requested
- **size_hint_min** – min width and height in pixel
- **size_hint_max** – max width and height in pixel
- **style** – not used

do_render(*surface:* Surface)

color

class arcade.gui.**UISpriteWidget**(***, *x=0*, *y=0*, *width=100*, *height=100*, *sprite:* Sprite | *None* = *None*, *size_hint=None*, *size_hint_min=None*, *size_hint_max=None*, ***kwargs*)

Bases: *UIWidget*

Create a UI element with a sprite that controls what is displayed.

Parameters

- **x** – x coordinate of bottom left
- **y** – y coordinate of bottom left
- **width** – width of widget
- **height** – height of widget
- **sprite** – Sprite to embed in gui
- **size_hint** – Tuple of floats (0.0-1.0), how much space of the parent should be requested
- **size_hint_min** – min width and height in pixel
- **size_hint_max** – max width and height in pixel
- **style** – not used

do_render(*surface: Surface*)

on_update(*dt*)

class arcade.gui.**UIWidget**(**, x: float = 0, y: float = 0, width: float = 100, height: float = 100, children: Iterable[UIWidget] = (), size_hint=None, size_hint_min=None, size_hint_max=None, **kwargs*)

Bases: `EventDispatcher`, `ABC`

The `UIWidget` class is the base class required for creating widgets.

We also have some default values and behaviors that you should be aware of:

- A `UIWidget` is not a `UILayout`: it will not change the position or the size of its children. If you want control over positioning or sizing, use a `UILayout`.

  Parameters

  - **x** – x coordinate of bottom left
  - **y** – y coordinate of bottom left
  - **width** – width of widget
  - **height** – height of widget
  - **size_hint** – Tuple of floats (0.0-1.0), how much space of the parent should be requested
  - **size_hint_min** – min width and height in pixel
  - **size_hint_max** – max width and height in pixel
  - **style** – not used

add(*child: W, **kwargs*) → W

Add a widget to this `UIWidget` as a child. Added widgets will receive ui events and be rendered.

By default, the latest added widget will receive ui events first and will be rendered on top of others.

Parameters

- **child** – widget to add
- **index** – position a widget is added, None has the highest priority

Returns

given child

center_on_screen() → W

Places this widget in the center of the current window.

clear()

dispatch_ui_event(*event: UIEvent*)

Dispatch a `UIEvent` using pyglet event dispatch mechanism

do_render(*surface: Surface*)

Render the widgets graphical representation, use `UIWidget.prepare_render()` to limit the drawing area to the widgets rect and draw relative to 0,0.

do_render_base(*surface: Surface*)

Renders background, border and “padding”

move(*dx=0*, *dy=0*)

Move the widget by dx and dy.

Parameters

- **dx** – x axis difference
- **dy** – y axis difference

on_event(*event:* UIEvent) → bool | None

Passes *UIEvent* s through the widget tree.

on_update(*dt*)

Custom logic which will be triggered.

prepare_render(*surface*)

Helper for rendering, the drawing area will be adjusted to the own position and size. Draw calls have to be relative to 0,0. This will also prevent any overdraw outside of the widgets area

Parameters
surface – Surface used for rendering

remove(*child:* UIWidget)

Removes a child from the UIManager which was directly added to it. This will not remove widgets which are added to a child of UIManager.

resize(**, *width=None*, *height=None*)

scale(*factor*)

Scales the size of the widget (x,y,width, height) by factor. :param factor: scale factor

trigger_full_render() → None

In case a widget uses transparent areas or was moved, it might be important to request a full rendering of parents

trigger_render()

This will delay a render right before the next frame is rendered, so that *UIWidget.do_render()* is not called multiple times.

with_background(**, *color: 'builtins.ellipsis'* | Color = *Ellipsis*, *texture:* *None* | Texture | NinePatchTexture = *Ellipsis*) → *UIWidget*

Set widgets background.

A color or texture can be used for background, if a texture is given, start and end point can be added to use the texture as ninepatch.

Parameters

- **color** – A color used as background
- **texture** – A texture or ninepatch texture used as background

Returns
self

with_border(**, *width=2*, *color=(0, 0, 0)*) → Self

Sets border properties :param width: border width :param color: border color :return: self

with_padding(**, top: 'builtins.ellipsis' | int = Ellipsis, right: 'builtins.ellipsis' | int = Ellipsis, bottom: 'builtins.ellipsis' | int = Ellipsis, left: 'builtins.ellipsis' | int = Ellipsis, all: 'builtins.ellipsis' | int = Ellipsis*) → *UIWidget*

Changes the padding to the given values if set. Returns itself :return: self

bottom

center

center_x

center_y

children

content_height

content_rect

content_size

content_width

height

left

padding

position

Returns bottom left coordinates

rect: *Rect*

An observable property which triggers observers when changed.

Parameters

- **default** – Default value which is returned, if no value set before
- **default_factory** – A callable which returns the default value. Will be called with the property and the instance

right

size

top

visible: bool

An observable property which triggers observers when changed.

Parameters

- **default** – Default value which is returned, if no value set before
- **default_factory** – A callable which returns the default value. Will be called with the property and the instance

width

x

y

class arcade.gui.**UITextureToggle**(**, x: float = 0, y: float = 0, width: float = 100, height: float = 50, on_texture: Texture | None = None, off_texture: Texture | None = None, value=False, size_hint=None, size_hint_min=None, size_hint_max=None, **kwargs*)

Bases: `UIInteractiveWidget`

A toggel button switching between on (True) and off (False) state.

on_texture and off_texture are required.

do_render(*surface: Surface*)

on_change(*event: UIOnChangeEvent*)

on_click(*event: UIOnClickEvent*)

value: bool

An observable property which triggers observers when changed.

Parameters

- **default** – Default value which is returned, if no value set before
- **default_factory** – A callable which returns the default value. Will be called with the property and the instance

class arcade.gui.**UIFlatButton**(**, x: float = 0, y: float = 0, width: float = 100, height: float = 50, text='', multiline=False, size_hint=None, size_hint_min=None, size_hint_max=None, style=None, **kwargs*)

Bases: `UIInteractiveWidget`, `UIStyledWidget`, `UITextWidget`

A text button, with support for background color and a border.

There are four states of the UITextureButton i.e normal, hovered, pressed and disabled.

Parameters

- **x** – x coordinate of bottom left
- **y** – y coordinate of bottom left
- **width** – width of widget. Defaults to texture width if not specified.
- **height** – height of widget. Defaults to texture height if not specified.
- **text** – text to add to the button.
- **multiline** – allows to wrap text, if not enough width available
- **style** – Used to style the button

class UIStyle(*font_size: int = 12, font_name: str | Tuple[str, ...] = ('calibri', 'arial'), font_color: Tuple[int, int, int, int] = (255, 255, 255, 255), bg: Tuple[int, int, int, int] = (21, 19, 21, 255), border: Tuple[int, int, int, int] | None = None, border_width: int = 0*)

Bases: `UIStyleBase`

Used to style the button. Below is its use case.

```
button = UIFlatButton(style={"normal": UIFlatButton.UIStyle(...),})
```

bg: Tuple[int, int, int, int] = (21, 19, 21, 255)

border: Tuple[int, int, int, int] | None = None

border_width: int = 0

font_color: Tuple[int, int, int, int] = (255, 255, 255, 255)

font_name: str | Tuple[str, ...] = ('calibri', 'arial')

font_size: int = 12

do_render(*surface:* Surface)

get_current_state() → str

Returns the current state of the button i.e disabled, press, hover or normal.

DEFAULT_STYLE = {'disabled': UIFlatButton.UIStyle(font_size=12, font_name=('calibri', 'arial'), font_color=Color(r=255, g=255, b=255, a=255), bg=Color(r=128, g=128, b=128, a=255), border=None, border_width=2), 'hover': UIFlatButton.UIStyle(font_size=12, font_name=('calibri', 'arial'), font_color=Color(r=255, g=255, b=255, a=255), bg=(21, 19, 21, 255), border=(77, 81, 87, 255), border_width=2), 'normal': UIFlatButton.UIStyle(font_size=12, font_name=('calibri', 'arial'), font_color=Color(r=255, g=255, b=255, a=255), bg=(21, 19, 21, 255), border=None, border_width=0), 'press': UIFlatButton.UIStyle(font_size=12, font_name=('calibri', 'arial'), font_color=Color(r=0, g=0, b=0, a=255), bg=Color(r=255, g=255, b=255, a=255), border=Color(r=255, g=255, b=255, a=255), border_width=2)}

parent: *UIManager* | *UIWidget* | None

class arcade.gui.**UITextureButton**(**, x: float = 0, y: float = 0, width: float | None = None, height: float | None = None, texture: None | Texture | NinePatchTexture = None, texture_hovered: None | Texture | NinePatchTexture = None, texture_pressed: None | Texture | NinePatchTexture = None, texture_disabled: None | Texture | NinePatchTexture = None, text: str = '', multiline: bool = False, scale: float | None = None, style: Dict[str, UIStyleBase] | None = None, size_hint=None, size_hint_min=None, size_hint_max=None, **kwargs*)

Bases: *UIInteractiveWidget*, *UIStyledWidget*[*UITextureButtonStyle*], *UITextWidget*

A button with an image for the face of the button.

There are four states of the UITextureButton i.e normal, hovered, pressed and disabled.

Parameters

- **x** – x coordinate of bottom left
- **y** – y coordinate of bottom left
- **width** – width of widget. Defaults to texture width if not specified.
- **height** – height of widget. Defaults to texture height if not specified.
- **texture** – texture to display for the widget.
- **texture_hovered** – different texture to display if mouse is hovering over button.
- **texture_pressed** – different texture to display if mouse button is pressed while hovering over button.
- **text** – text to add to the button.

- **multiline** – allows to wrap text, if not enough width available
- **style** – Used to style the button for different states.
- **scale** – scale the button, based on the base texture size.
- **size_hint** – Tuple of floats (0.0-1.0), how much space of the parent should be requested
- **size_hint_min** – min width and height in pixel
- **size_hint_max** – max width and height in pixel

UIStyle

alias of *UITextureButtonStyle*

do_render(*surface: Surface*)

get_current_state() → str

Returns the current state of the button i.e disabled, press, hover or normal.

DEFAULT_STYLE = {'disabled': UITextureButtonStyle(font_size=12, font_name=('calibri', 'arial'), font_color=Color(r=255, g=255, b=255, a=255), border_width=2), 'hover': UITextureButtonStyle(font_size=12, font_name=('calibri', 'arial'), font_color=Color(r=255, g=255, b=255, a=255), border_width=2), 'normal': UITextureButtonStyle(font_size=12, font_name=('calibri', 'arial'), font_color=Color(r=255, g=255, b=255, a=255), border_width=2), 'press': UITextureButtonStyle(font_size=12, font_name=('calibri', 'arial'), font_color=Color(r=0, g=0, b=0, a=255), border_width=2)}

texture

Returns the normal texture for the face of the button.

texture_hovered

Returns the hover texture for the face of the button.

texture_pressed

Returns the pressed texture for the face of the button.

class arcade.gui.**UITextureButtonStyle**(*font_size: int = 12, font_name: str | Tuple[str, ...] = ('calibri', 'arial'), font_color: Tuple[int, int, int, int] = (255, 255, 255, 255), border_width: int = 2*)

Bases: *UIStyleBase*

Used to style the texture button. Below is its use case.

```
button = UITextureButton(style={"normal": UITextureButton.UIStyle(...),})
```

border_width: int = 2

font_color: Tuple[int, int, int, int] = (255, 255, 255, 255)

font_name: str | Tuple[str, ...] = ('calibri', 'arial')

font_size: int = 12

33.31 GUI Events

class arcade.gui.**UIEvent**(*source: Any*)

Bases:

An event created by the GUI system. Can be passed using widget.dispatch("on_event", event). An event always has a source, which is the UIManager for general input events, but will be the specific widget in case of events like on_click events.

source: Any

class arcade.gui.**UIKeyEvent**(*source: Any*, *symbol: int*, *modifiers: int*)

Bases: `UIEvent`

Covers all keyboard event.

modifiers: int

symbol: int

class arcade.gui.**UIKeyPressEvent**(*source: Any*, *symbol: int*, *modifiers: int*)

Bases: `UIKeyEvent`

Triggered when a key is pressed.

modifiers: int

source: Any

symbol: int

class arcade.gui.**UIKeyReleaseEvent**(*source: Any*, *symbol: int*, *modifiers: int*)

Bases: `UIKeyEvent`

Triggered when a key is released.

modifiers: int

source: Any

symbol: int

class arcade.gui.**UIMouseDragEvent**(*source: Any*, *x: float*, *y: float*, *dx: float*, *dy: float*, *buttons: int*, *modifiers: int*)

Bases: `UIMouseEvent`

Triggered when the mouse moves while one of its buttons being pressed.

buttons: int

dx: float

dy: float

modifiers: int

class arcade.gui.**UIMouseEvent**(*source: Any*, *x: float*, *y: float*)

Bases: `UIEvent`

Covers all mouse event

pos

x: float

y: float

class arcade.gui.**UIMouseMovementEvent**(*source: Any*, *x: float*, *y: float*, *dx: float*, *dy: float*)

Bases: *UIMouseEvent*

Triggered when the mouse is moved.

dx: float

dy: float

class arcade.gui.**UIMousePressEvent**(*source: Any*, *x: float*, *y: float*, *button: int*, *modifiers: int*)

Bases: *UIMouseEvent*

Triggered when a mouse button(left, right, middle) is pressed.

button: int

modifiers: int

class arcade.gui.**UIMouseReleaseEvent**(*source: Any*, *x: float*, *y: float*, *button: int*, *modifiers: int*)

Bases: *UIMouseEvent*

Triggered when a mouse button is released.

button: int

modifiers: int

class arcade.gui.**UIMouseScrollEvent**(*source: Any*, *x: float*, *y: float*, *scroll_x: int*, *scroll_y: int*)

Bases: *UIMouseEvent*

Triggered by rotating the scroll wheel on the mouse.

scroll_x: int

scroll_y: int

class arcade.gui.**UIOnActionEvent**(*source: Any*, *action: Any*)

Bases: *UIEvent*

Notification about an action

Parameters

action – Value describing the action, mostly a string

action: Any

class arcade.gui.**UIOnChangeEvent**(*source: Any*, *old_value: Any*, *new_value: Any*)

Bases: *UIEvent*

Value of a widget changed

new_value: Any

old_value: Any

class arcade.gui.**UIOnClickEvent**(*source: Any*, *x: float*, *y: float*)

Bases: *UIMouseEvent*

Triggered when a button is clicked.

source: Any

x: float

y: float

class arcade.gui.**UIOnUpdateEvent**(*source: Any*, *dt: int*)

Bases: *UIEvent*

Arcade on_update callback passed as *UIEvent*

dt: int

class arcade.gui.**UITextEvent**(*source: Any*, *text: str*)

Bases: *UIEvent*

Covers all the text cursor event.

text: str

class arcade.gui.**UITextMotionEvent**(*source: Any*, *motion: Any*)

Bases: *UIEvent*

Triggered when text cursor moves.

motion: Any

class arcade.gui.**UITextMotionSelectEvent**(*source: Any*, *selection: Any*)

Bases: *UIEvent*

Triggered when the text cursor moves selecting the text with it.

selection: Any

33.32 GUI Properties

class arcade.gui.**DictProperty**

Bases: *Property*

Property that represents a dict. Only dict are allowed. Any other classes are forbidden.

set(*instance*, *value: dict*)

default_factory

name: str

obs: WeakKeyDictionary[Any, _Obs]

class arcade.gui.**ListProperty**

Bases: *Property*

Property that represents a list. Only list are allowed. Any other classes are forbidden.

set(*instance*, *value: dict*)

default_factory

name: str

obs: WeakKeyDictionary[Any, _Obs]

class arcade.gui.**Property**(*default: P | None = None*, *default_factory: Callable[[Any, Any], P] | None = None*)

Bases: `Generic`[P]

An observable property which triggers observers when changed.

Parameters

- **default** – Default value which is returned, if no value set before
- **default_factory** – A callable which returns the default value. Will be called with the property and the instance

bind(*instance*, *callback*)

dispatch(*instance*, *value*)

get(*instance*) → P

set(*instance*, *value*)

default_factory

name: str

obs: WeakKeyDictionary[Any, _Obs]

arcade.gui.**bind**(*instance*, *property: str*, *callback*)

Binds a function to the change event of the property. A reference to the function will be kept, so that it will be still invoked, even if it would normally have been garbage collected.

def log_change():
print("Something changed")

class MyObject:
name = Property()

my_obj = MyObject() bind(my_obj, "name", log_change)

my_obj.name = "Hans" # > Something changed

Parameters

- **instance** – Instance owning the property
- **property** – Name of the property
- **callback** – Function to call

Returns
None

33.33 GUI Style

class arcade.gui.**UIStyleBase**

Bases:

Base class for styles to ensure a general interface and implement additional magic.

Support dict like access syntax.

A styled widget should own a dataclass, which subclasses this class

get(*key*, *default: str*) → str

get(*key*, *default: Any*) → Any

class arcade.gui.**UIStyledWidget**(**, *style: Mapping[str, StyleRef]*, ***kwargs*)

Bases: *UIWidget*, Generic[StyleRef]

abstract get_current_state() → str

Return the current state of the widget. These should be contained in the style dict.

Well known states: - normal - hover - press - disabled

get_current_style() → StyleRef

Return style based on any state of the widget

style: Mapping

Property that represents a dict. Only dict are allowed. Any other classes are forbidden.

33.34 arcade.key package

Mapping of keyboard keys to values.

```
# flake8: noqa
"""
Constants used to signify what keys on the keyboard were pressed.
"""

from __future__ import annotations
from sys import platform

# Key modifiers
# Done in powers of two, so you can do a bit-wise 'and' to detect
# multiple modifiers.
MOD_SHIFT = 1
MOD_CTRL = 2
MOD_ALT = 4
MOD_CAPSLOCK = 8
MOD_NUMLOCK = 16
MOD_WINDOWS = 32
MOD_COMMAND = 64
MOD_OPTION = 128
MOD_SCROLLLOCK = 256

# Platform-specific base hotkey modifier
```

(continues on next page)

(continued from previous page)

```
MOD_ACCEL = MOD_CTRL
if platform == 'darwin':
    MOD_ACCEL = MOD_COMMAND

# Keys
BACKSPACE = 65288
TAB = 65289
LINEFEED = 65290
CLEAR = 65291
RETURN = 65293
ENTER = 65293
PAUSE = 65299
SCROLLLOCK = 65300
SYSREQ = 65301
ESCAPE = 65307
HOME = 65360
LEFT = 65361
UP = 65362
RIGHT = 65363
DOWN = 65364
PAGEUP = 65365
PAGEDOWN = 65366
END = 65367
BEGIN = 65368
DELETE = 65535
SELECT = 65376
PRINT = 65377
EXECUTE = 65378
INSERT = 65379
UNDO = 65381
REDO = 65382
MENU = 65383
FIND = 65384
CANCEL = 65385
HELP = 65386
BREAK = 65387
MODESWITCH = 65406
SCRIPTSWITCH = 65406
MOTION_UP = 65362
MOTION_RIGHT = 65363
MOTION_DOWN = 65364
MOTION_LEFT = 65361
MOTION_NEXT_WORD = 1
MOTION_PREVIOUS_WORD = 2
MOTION_BEGINNING_OF_LINE = 3
MOTION_END_OF_LINE = 4
MOTION_NEXT_PAGE = 65366
MOTION_PREVIOUS_PAGE = 65365
MOTION_BEGINNING_OF_FILE = 5
MOTION_END_OF_FILE = 6
MOTION_BACKSPACE = 65288
MOTION_DELETE = 65535
```

(continues on next page)

(continued from previous page)

```
NUMLOCK = 65407
NUM_SPACE = 65408
NUM_TAB = 65417
NUM_ENTER = 65421
NUM_F1 = 65425
NUM_F2 = 65426
NUM_F3 = 65427
NUM_F4 = 65428
NUM_HOME = 65429
NUM_LEFT = 65430
NUM_UP = 65431
NUM_RIGHT = 65432
NUM_DOWN = 65433
NUM_PRIOR = 65434
NUM_PAGE_UP = 65434
NUM_NEXT = 65435
NUM_PAGE_DOWN = 65435
NUM_END = 65436
NUM_BEGIN = 65437
NUM_INSERT = 65438
NUM_DELETE = 65439
NUM_EQUAL = 65469
NUM_MULTIPLY = 65450
NUM_ADD = 65451
NUM_SEPARATOR = 65452
NUM_SUBTRACT = 65453
NUM_DECIMAL = 65454
NUM_DIVIDE = 65455

# Numbers on the numberpad
NUM_0 = 65456
NUM_1 = 65457
NUM_2 = 65458
NUM_3 = 65459
NUM_4 = 65460
NUM_5 = 65461
NUM_6 = 65462
NUM_7 = 65463
NUM_8 = 65464
NUM_9 = 65465

F1 = 65470
F2 = 65471
F3 = 65472
F4 = 65473
F5 = 65474
F6 = 65475
F7 = 65476
F8 = 65477
F9 = 65478
F10 = 65479
F11 = 65480
```

(continues on next page)

(continued from previous page)

```
F12 = 65481
F13 = 65482
F14 = 65483
F15 = 65484
F16 = 65485
F17 = 65486
F18 = 65487
F19 = 65488
F20 = 65489
F21 = 65490
F22 = 65491
F23 = 65492
F24 = 65493
LSHIFT = 65505
RSHIFT = 65506
LCTRL = 65507
RCTRL = 65508
CAPSLOCK = 65509
LMETA = 65511
RMETA = 65512
LALT = 65513
RALT = 65514
LWINDOWS = 65515
RWINDOWS = 65516
LCOMMAND = 65517
RCOMMAND = 65518
LOPTION = 65488
ROPTION = 65489
SPACE = 32
EXCLAMATION = 33
DOUBLEQUOTE = 34
HASH = 35
POUND = 35
DOLLAR = 36
PERCENT = 37
AMPERSAND = 38
APOSTROPHE = 39
PARENLEFT = 40
PARENRIGHT = 41
ASTERISK = 42
PLUS = 43
COMMA = 44
MINUS = 45
PERIOD = 46
SLASH = 47

# Numbers on the main keyboard
KEY_0 = 48
KEY_1 = 49
KEY_2 = 50
KEY_3 = 51
KEY_4 = 52
```

(continues on next page)

(continued from previous page)

```
KEY_5 = 53
KEY_6 = 54
KEY_7 = 55
KEY_8 = 56
KEY_9 = 57
COLON = 58
SEMICOLON = 59
LESS = 60
EQUAL = 61
GREATER = 62
QUESTION = 63
AT = 64
BRACKETLEFT = 91
BACKSLASH = 92
BRACKETRIGHT = 93
ASCIICIRCUM = 94
UNDERSCORE = 95
GRAVE = 96
QUOTELEFT = 96
A = 97
B = 98
C = 99
D = 100
E = 101
F = 102
G = 103
H = 104
# noinspection PyPep8
I = 105
J = 106
K = 107
L = 108
M = 109
N = 110
# noinspection PyPep8
O = 111
P = 112
Q = 113
R = 114
S = 115
T = 116
U = 117
V = 118
W = 119
X = 120
Y = 121
Z = 122
BRACELEFT = 123
BAR = 124
BRACERIGHT = 125
ASCIITILDE = 126
```

33.35 arcade.csscolor package

These are standard CSS named colors you can use when drawing.

You can specify colors four ways:

- Standard CSS color names (this package): `arcade.csscolor.RED`
- Nonstandard color names *arcade.color package*: `arcade.color.RED`
- Three-byte numbers: `(255, 0, 0)`
- Four-byte numbers (fourth byte is transparency. 0 transparent, 255 opaque): `(255, 0, 0, 255)`

33.36 arcade.color package

These are named colors you can use when drawing.

You can specify colors four ways:

- Standard CSS color names *arcade.csscolor package*: `arcade.csscolor.RED`
- Nonstandard color names (this package): `arcade.color.RED`
- Three-byte numbers: `(255, 0, 0)`
- Four-byte numbers (fourth byte is transparency. 0 transparent, 255 opaque): `(255, 0, 0, 255)`

CHAPTER

THIRTYFOUR

BUILT-IN RESOURCES

Resource files are images and sounds built into Arcade that can be used to quickly build and test simple code without having to worry about copying files into the project.

Any file loaded that starts with `:resources:` will attempt to load that file from the library resources instead of the project directory.

Many of the resources come from Kenney.nl and are licensed under CC0 (Creative Commons Zero). Be sure to check out his web page for a much wider selection of assets.

Table 1: :resources:gui_basic_assets/

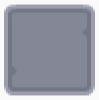
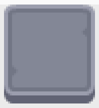
red_button_normal.png	slider_thumb.png	slider_bar.png
red_button_press.png	red_button_hover.png	 button_square_blue_pressed.png
 button_square_blue.png		

Table 2: :resources:gui_basic_assets/icons/

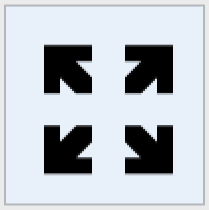
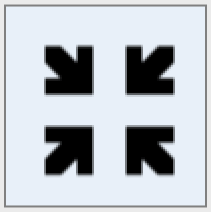
 larger.png	 smaller.png

Table 3: :resources:gui_basic_assets/window/

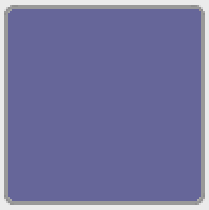
 dark_blue_gray_panel.png	grey_panel.png

Table 4: :resources:gui_basic_assets/items/

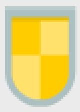
sword_gold.png	 shield_gold.png

Table 5: :resources:gui_basic_assets/toggle/

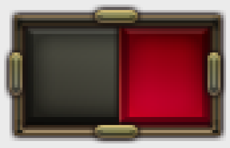
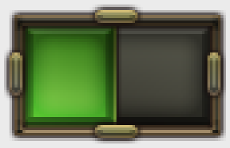
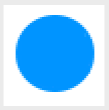
 switch_red.png	 switch_green.png	 circle_switch_on.png
circle_switch_off.png		

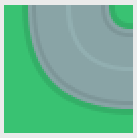
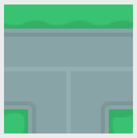
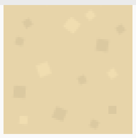
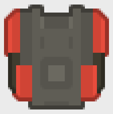
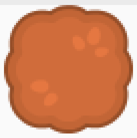
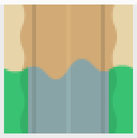
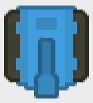
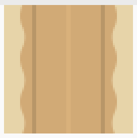
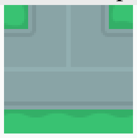
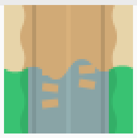
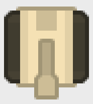
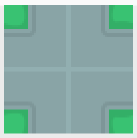
Table 6: :resources:images/miami_synth_parallax/layers/

sun.png	palms.png	buildings.png
back.png	highway.png	

Table 7: :resources:images/miami_synth_parallax/car/

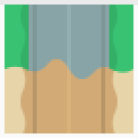
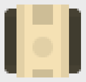
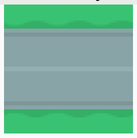
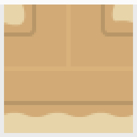
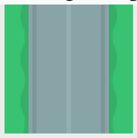
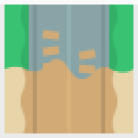
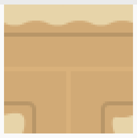
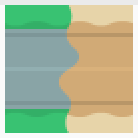
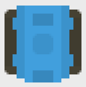
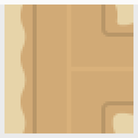
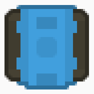
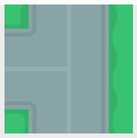
car-idle.png	car-running0.png	car-running1.png
car-running3.png	car-running2.png	

Table 8: :resources:images/topdown_tanks/

 tileGrass_roadCornerUR.png	tileSand_roadCornerUL.png	 tileGrass_roadSplitS.png
tankGreen_barrel3_outline.png	tankBlue_barrel2.png	 tileSand2.png
tankBody_darkLarge_outline.png	 tankBody_bigRed_outline.png	tankSand_barrel2.png
tankGreen_barrel3.png	tankBody_sand_outline.png	 treeBrown_large.png
 tileGrass_roadTransitionN.png	 tank_blue.png	tank_red.png
 tileSand_roadNorth.png	 tileGrass_roadSplitN.png	treeBrown_small.png
tankRed_barrel2_outline.png	tileSand_roadSplitW.png	tankDark_barrel2_outline.png
 tileGrass_roadTransitionN_dirt.png	tankRed_barrel1.png	tankDark_barrel3.png
tankBody_red_outline.png	 tank_sand.png	 tileGrass_roadCrossing.png

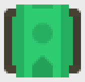
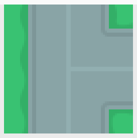
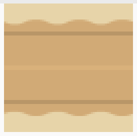
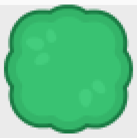
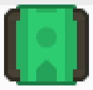
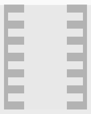
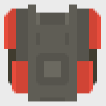
continues on next page

Table 8 – continued from previous page

tankBody_darkLarge.png	tankBlue_barrel3_outline.png	tankBody_huge_outline.png
tankRed_barrel2.png	tankBlue_barrel1.png	tankSand_barrel1_outline.png
tankGreen_barrel2.png	tankGreen_barrel1.png	 tileGrass_roadTransitionS.png
tankBody_dark_outline.png	tank_dark.png	 tankBody_sand.png
 tileGrass_roadEast.png	tank_green.png	tileGrass_transitionN.png
 tileSand_roadSplitN.png	 tileGrass_roadNorth.png	 tileGrass_roadTransitionS_dirt.png
 tileSand_roadSplitS.png	tankDark_barrel1_outline.png	 tileGrass_roadTransitionE.png
tileGrass_transitionS.png	 tankBody_blue.png	 tileSand_roadSplitE.png
tankBody_red.png	tileGrass_roadCornerLR.png	 tankBody_blue_outline.png
tankDark_barrel2.png	 tileGrass_roadSplitW.png	tileSand_roadCrossingRound.png

continues on next page

Table 8 – continued from previous page

tankBody_dark.png	tileGrass_roadCrossingRound.png	tileGrass_transitionE.png
 tankBody_green.png	tileSand_roadCornerUR.png	tileGrass2.png
tankRed_barrel3_outline.png	tankGreen_barrel1_outline.png	tankRed_barrel1_outline.png
tileGrass_roadTransitionW.png	tankBlue_barrel3.png	tileGrass_roadCornerLL.png
tankDark_barrel3_outline.png	tileGrass_transitionW.png	tankBlue_barrel1_outline.png
tileGrass1.png	tracksSmall.png	 tileGrass_roadSplitE.png
tileSand_roadCornerLR.png	tile- Grass_roadTransitionW_dirt.png	 tileSand_roadEast.png
tankBlue_barrel2_outline.png	tileGrass_roadCornerUL.png	 treeGreen_large.png
tankSand_barrel3.png	 tankBody_green_outline.png	tankRed_barrel3.png
 tracksDouble.png	 tankBody_bigRed.png	tankDark_barrel1.png

continues on next page

Table 8 – continued from previous page

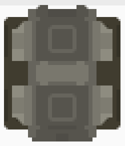
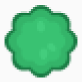
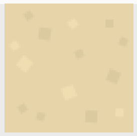
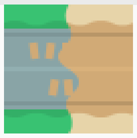
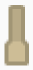
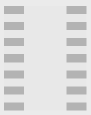
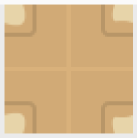
 tankBody_huge.png	tankSand_barrel1.png	 treeGreen_small.png
tankGreen_barrel2_outline.png	 tileSand1.png	tileSand_roadCornerLL.png
 tileGrass_roadTransitionE_dirt.png	 tankSand_barrel2_outline.png	tankSand_barrel3_outline.png
 tracksLarge.png	 tileSand_roadCrossing.png	

Table 9: :resources:images/pinball/

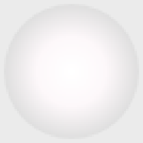
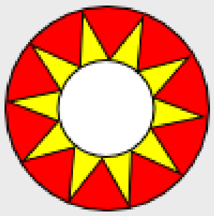
 pool_cue_ball.png	 bumper.png	

Table 10: :resources:images/backgrounds/

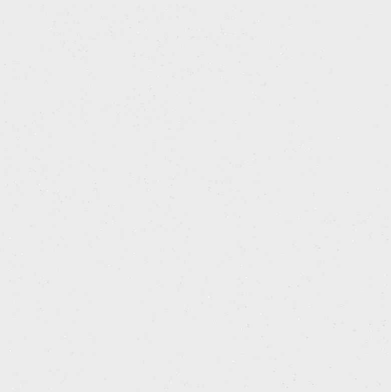
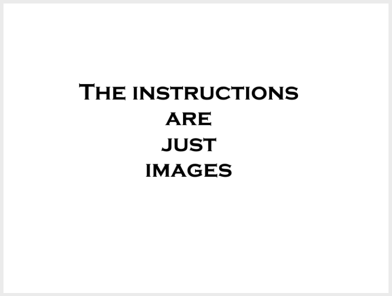
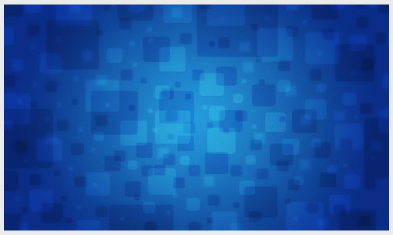
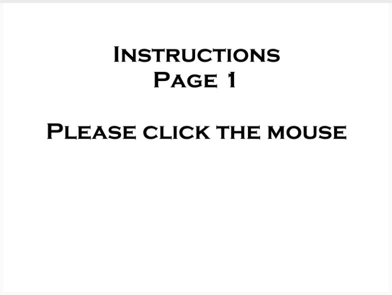
 stars.png	 THE INSTRUCTIONS ARE JUST IMAGES instructions_1.png	 abstract_1.jpg
abstract_2.jpg	 INSTRUCTIONS PAGE 1 PLEASE CLICK THE MOUSE instructions_0.png	

Table 11: :resources:images/cards/

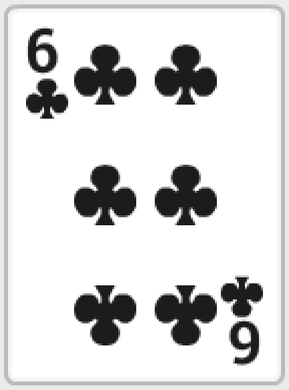
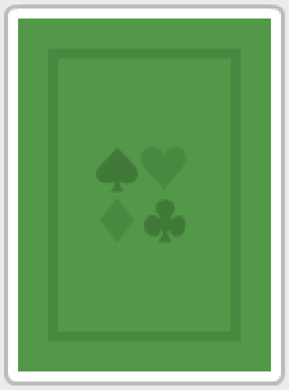
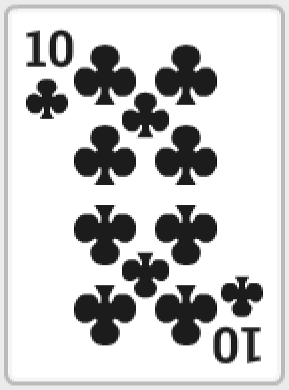
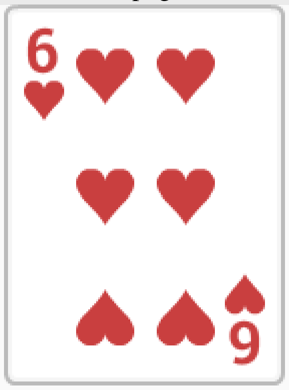
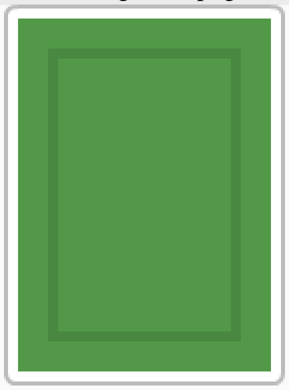
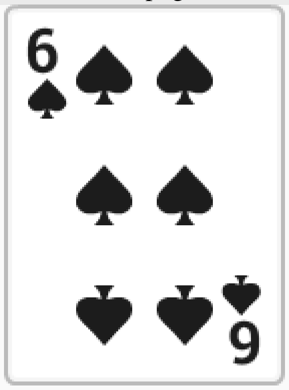
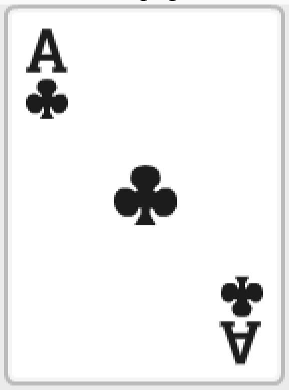
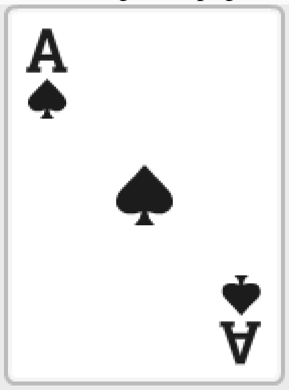
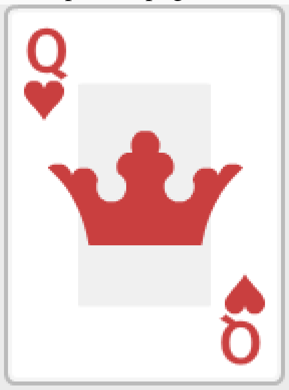
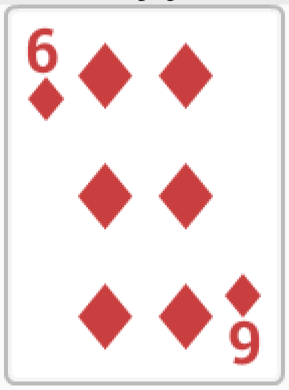
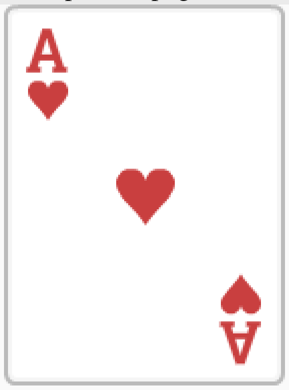
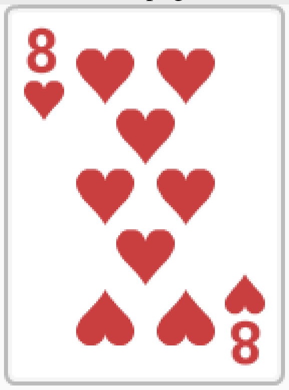
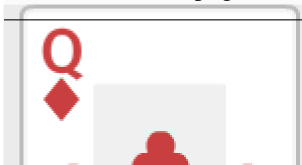
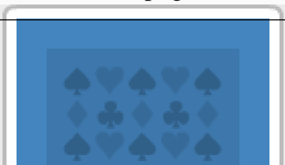
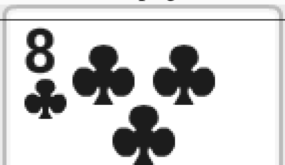
 cardClubs6.png	 cardBack_green5.png	 cardClubs10.png
 cardHearts6.png	 cardBack_green3.png	 cardSpades6.png
 cardClubsA.png	 cardSpadesA.png	 cardHeartsQ.png
 cardDiamonds6.png	 cardHeartsA.png	 cardHearts8.png
		

Table 12: :resources:images/spritesheets/

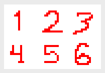
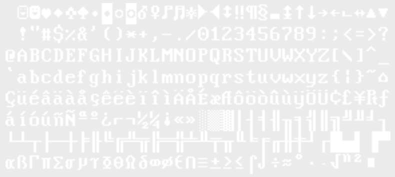
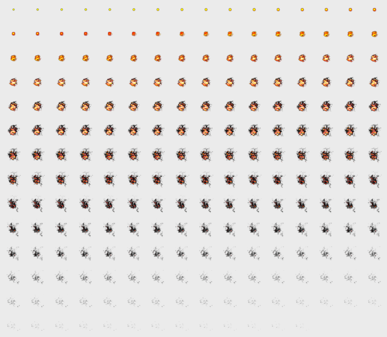
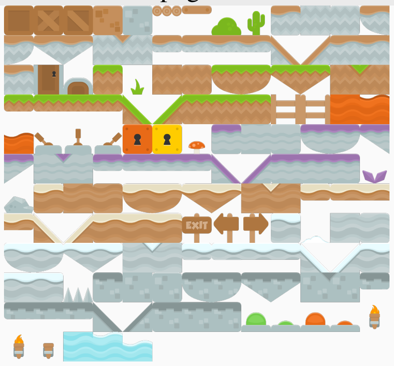
 number_sheet.png	 codepage_437.png	 explosion.png
 tiles.png		

Table 13: :resources:images/cybercity_background/

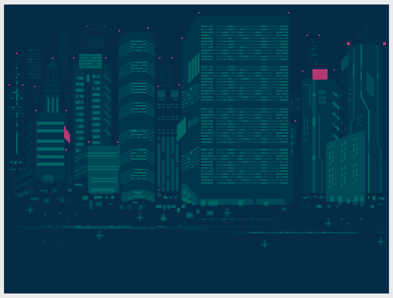
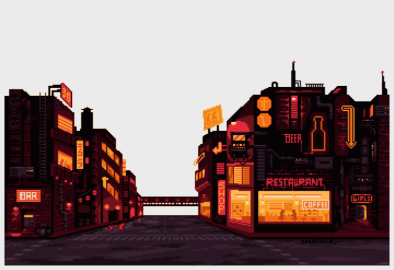
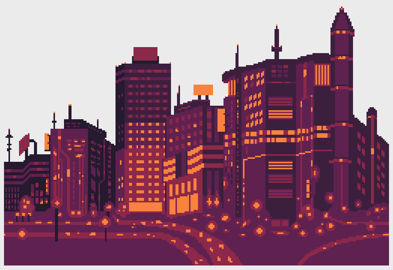
 far-buildings.png	 foreground.png	 back-buildings.png

Table 14: :resources:images/items/

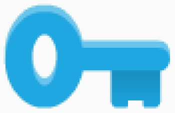
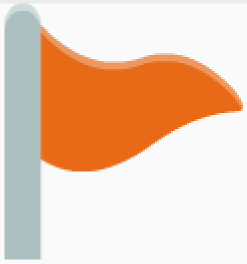
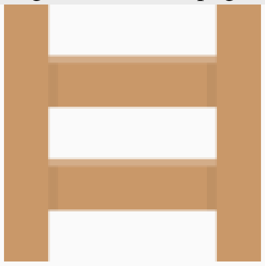
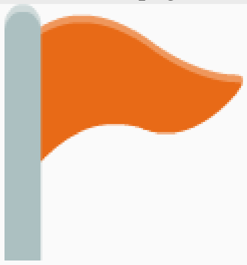
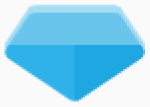
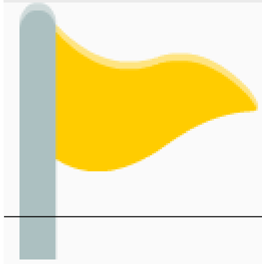
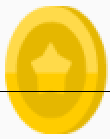
coinGold_ur.png	coinGold_ul.png	coinSilver.png
 keyBlue.png	coinGold.png	 flagRed1.png
flagYellow_down.png	coinGold_lr.png	keyRed.png
 ladderMid.png	 flagRed2.png	 gemBlue.png
flagGreen_down.png	gold_1.png	star.png
 flagYellow1.png	 gold_2.png	coinBronze.png

Table 15: :resources:images/space_shooter/

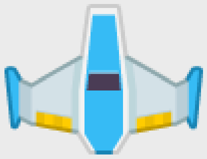
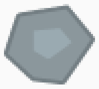
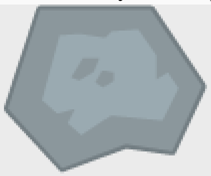
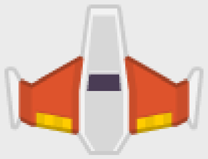
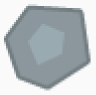
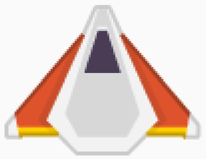
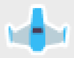
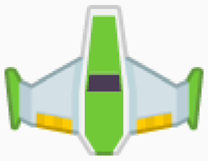
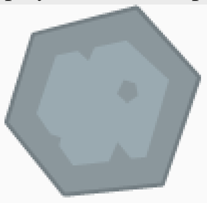
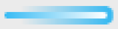
playerShip2_orange.png	meteorGrey_small2.png	 playerShip1_blue.png
 meteorGrey_med2.png	meteorGrey_tiny1.png	laserRed01.png
 meteorGrey_big1.png	 playerShip1_orange.png	playerLife1_orange.png
 meteorGrey_med1.png	 playerShip3_orange.png	meteorGrey_big3.png
meteorGrey_big2.png	playerLife1_green.png	 playerLife1_blue.png
meteorGrey_tiny2.png	 playerShip1_green.png	 meteorGrey_big4.png
meteorGrey_small1.png	 laserBlue01.png	

Table 16: :resources:images/alien/

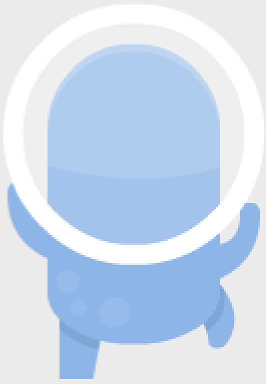
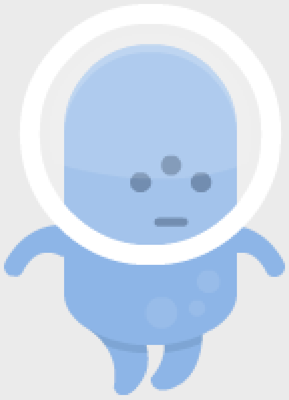
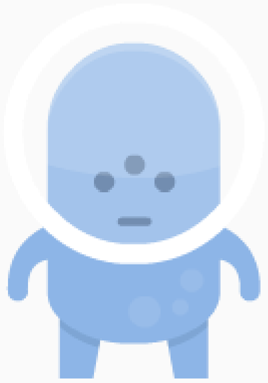
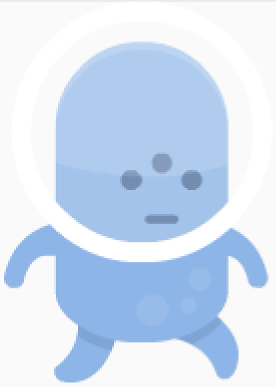
alienBlue_jump.png	 alienBlue_climb1.png	 alienBlue_walk2.png
 alienBlue_front.png	alienBlue_climb2.png	 alienBlue_walk1.png

Table 17: :resources:images/test_textures/

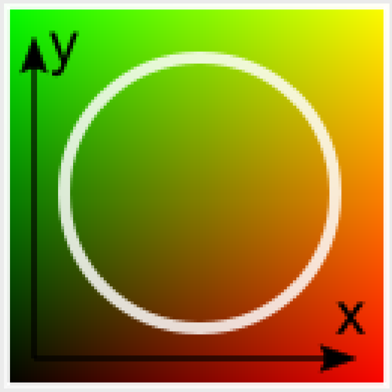
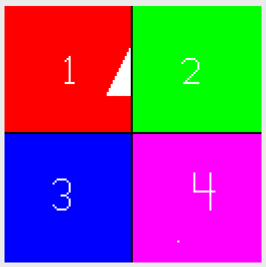
 xy_square.png	 test_texture.png	anim.gif

Table 18: :resources:images/test_textures/normal_mapping/

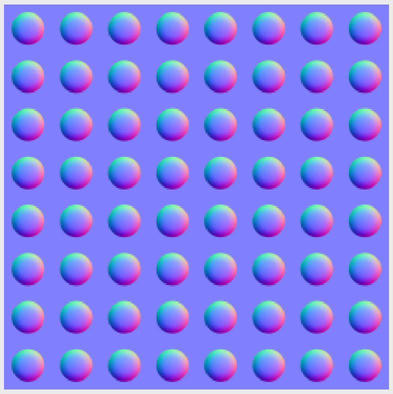
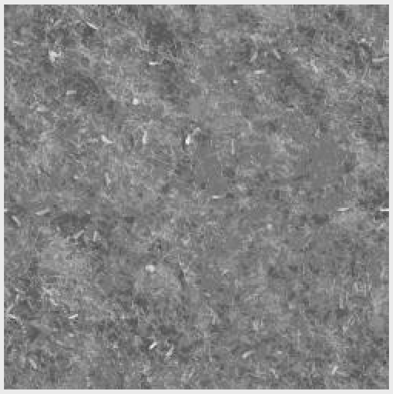
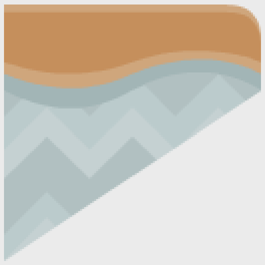
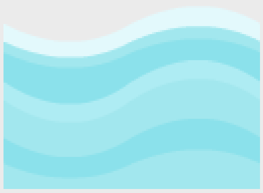
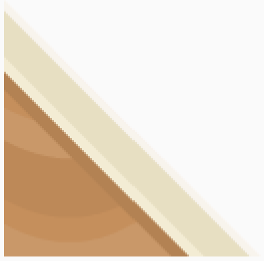
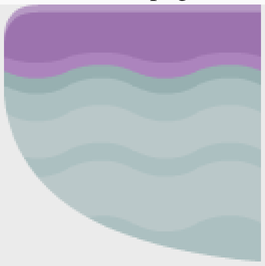
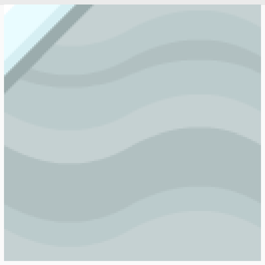
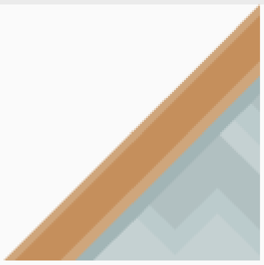
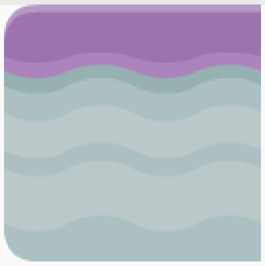
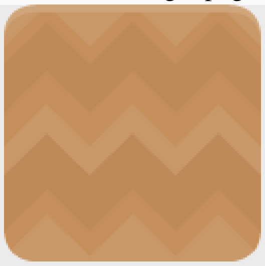
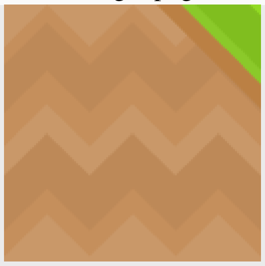
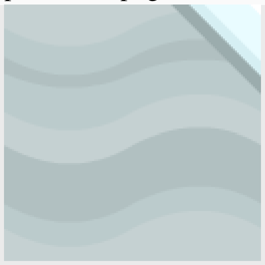
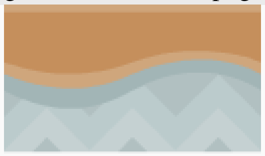
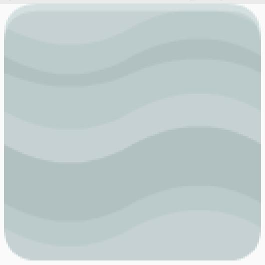
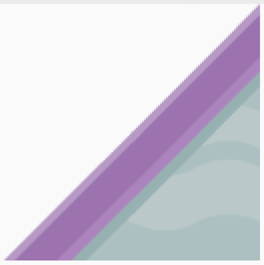
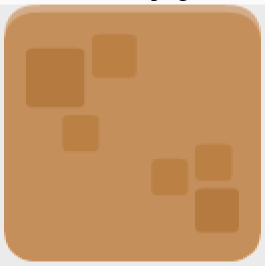
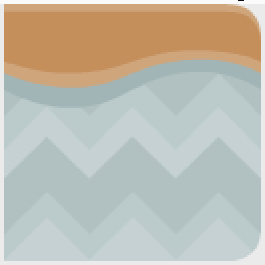
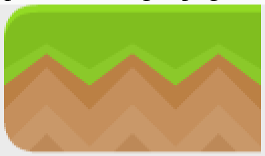
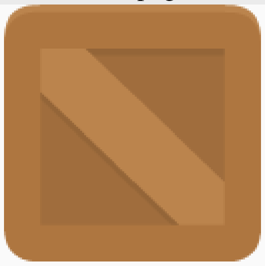
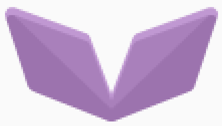
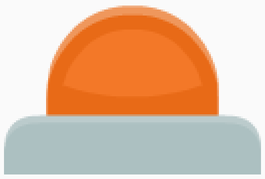
 normal.jpg	 diffuse.jpg	

Table 19: :resources:images/tiles/

sandCorner_right.png	 dirtCliffAlt_right.png	 waterTop_low.png
stoneCenter.png	planetHalf_left.png	 sandHill_left.png
switchGreen_pressed.png	dirtCenter_rounded.png	 planetCliff_left.png

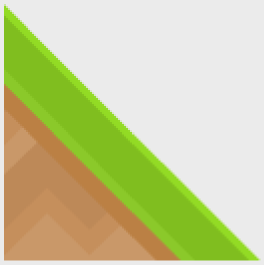
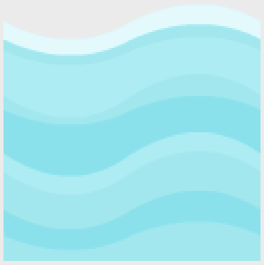
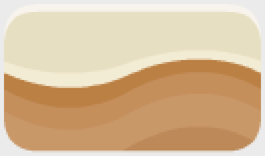
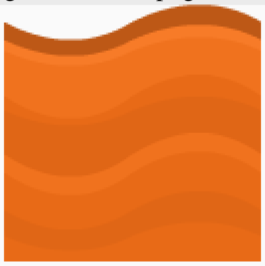
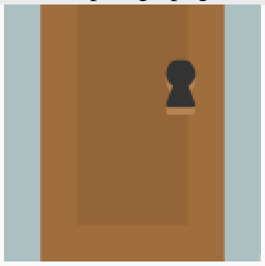
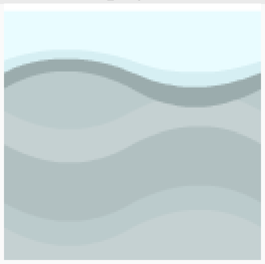
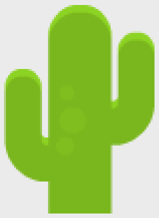
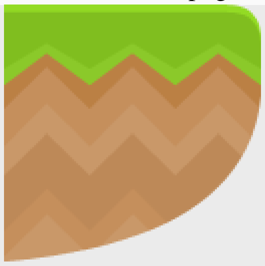
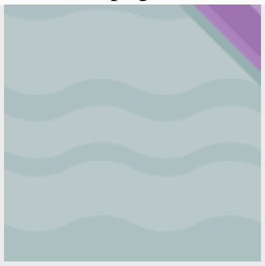
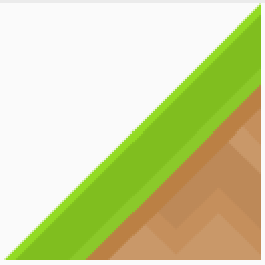
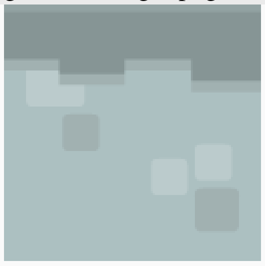
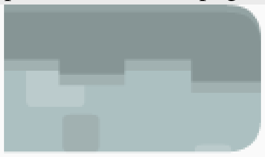
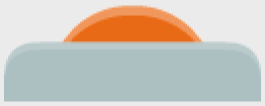
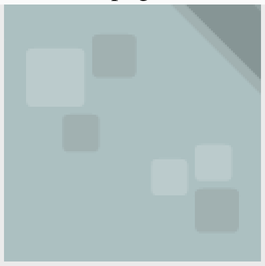
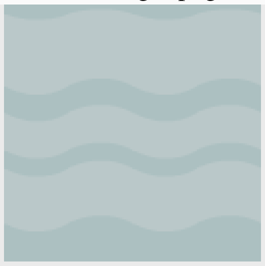
continues on next page

Table 19 – continued from previous page

 snowCorner_right.png	 dirtHill_right.png	 planetLeft.png
 grassCenter_round.png	 grassCorner_left.png	 snowCorner_left.png
 dirtHalf_mid.png	 snowCenter_rounded.png	 planetHill_right.png
 brickBrown.png	 dirtRight.png	 grassHalf_left.png
 boxCrate_single.png	 plantPurple.png	 switchRed.png

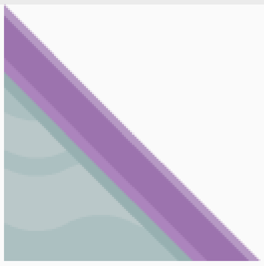
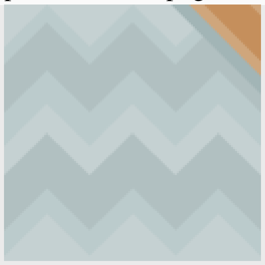
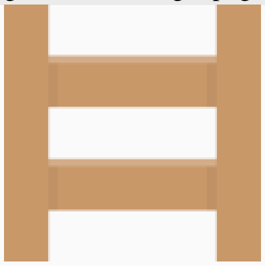
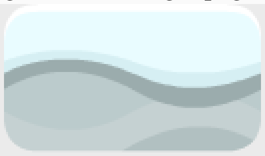
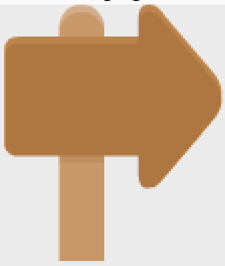
continues on next page

Table 19 – continued from previous page

 grassHill_left.png	 waterTop_high.png	 sandHalf.png
 lavaTop_high.png	 doorClosed_mid.png	 snowMid.png
 cactus.png	 grassCliff_right.png	 planetCorner_left.png
 grassHill_right.png	 stoneMid.png	 stoneHalf_right.png
 switchRed_pressed.png	 stoneCorner_left.png	 planetCenter.png

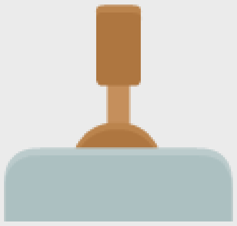
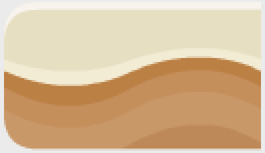
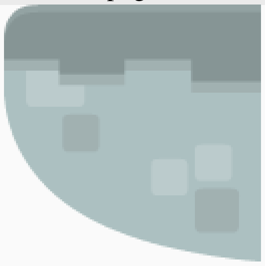
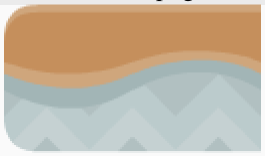
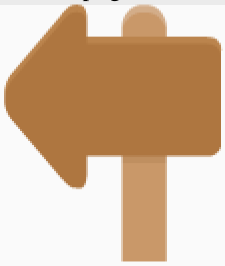
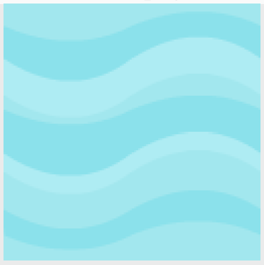
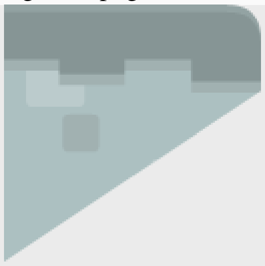
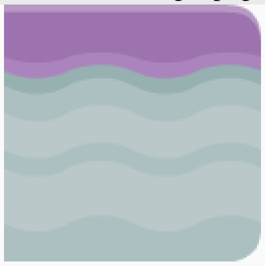
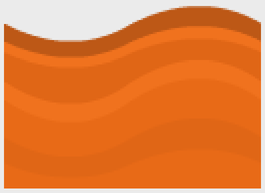
continues on next page

Table 19 – continued from previous page

 planetHill_left.png	sandCenter.png	dirtLeft.png
 dirtCorner_left.png	grassCliffAlt_right.png	planetCorner_right.png
grassCorner_right.png	 ladderMid.png	sandCorner_left.png
 snowHalf.png	 signRight.png	sand.png
snow_pile.png	bomb.png	bush.png

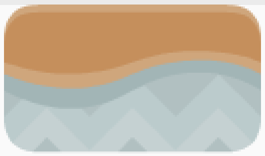
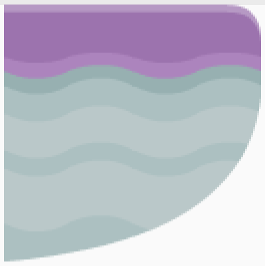
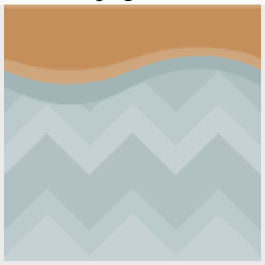
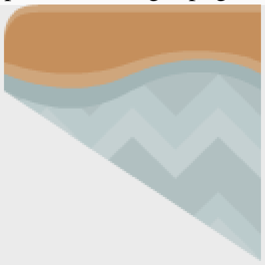
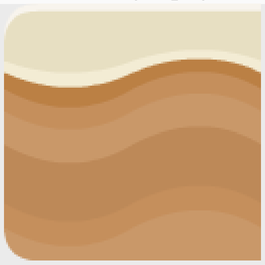
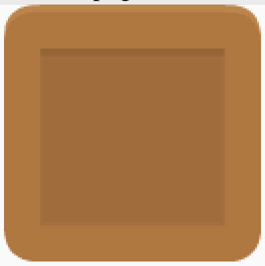
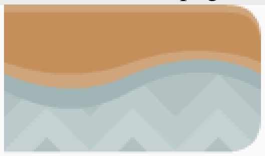
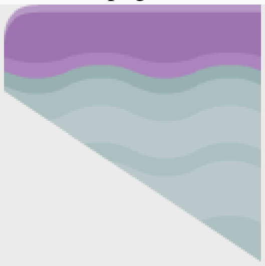
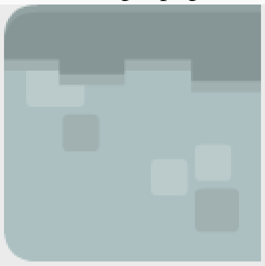
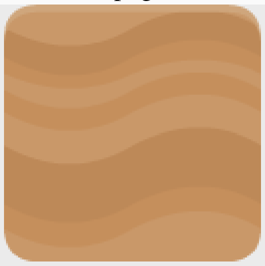
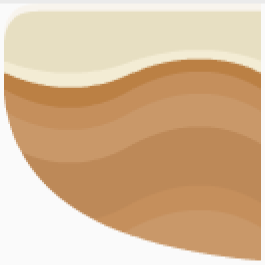
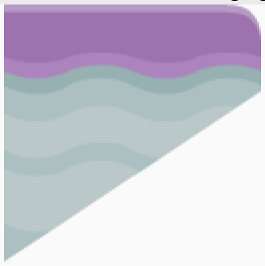
continues on next page

Table 19 – continued from previous page

 leverMid.png	 sandHalf_left.png	torchOff.png
 stoneCliff_left.png	 dirtHalf_left.png	 signLeft.png
grass.png	 water.png	 stoneCliffAlt_right.png
spikes.png	signExit.png	 planetRight.png
snowHill_right.png	snowHalf_mid.png	 lavaTop_low.png

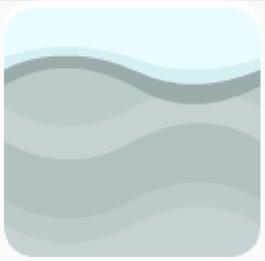
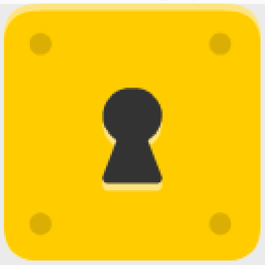
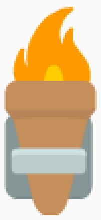
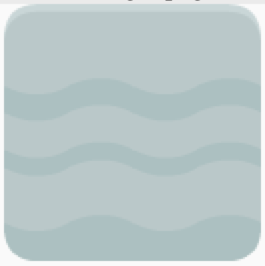
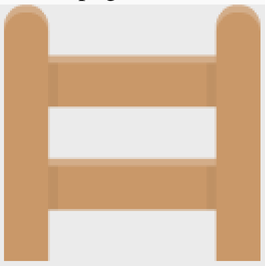
continues on next page

Table 19 – continued from previous page

 dirtHalf.png	 planetCliff_right.png	sandHalf_right.png
 dirtMid.png	 dirtCliffAlt_left.png	 sandLeft.png
 boxCrate.png	 dirtHalf_right.png	stoneHalf.png
 planetCliffAlt_left.png	 stoneLeft.png	 sandCenter_rounded.png
 sandCliff_left.png	dirtCenter.png	 planetCliffAlt_right.png

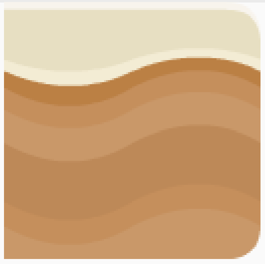
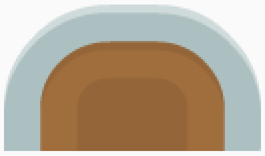
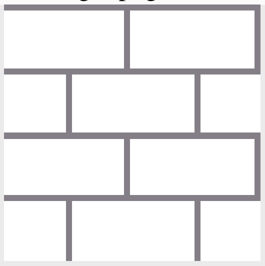
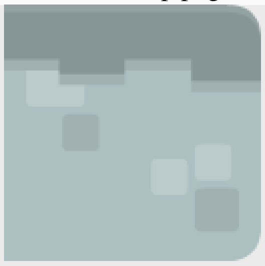
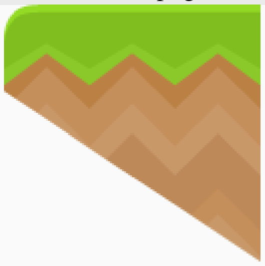
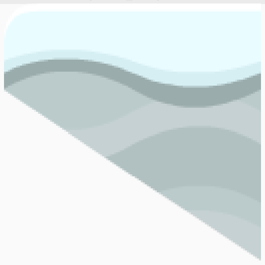
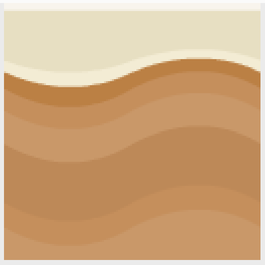
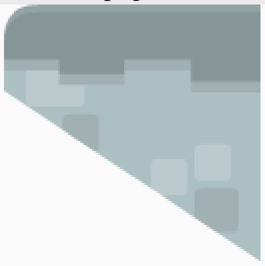
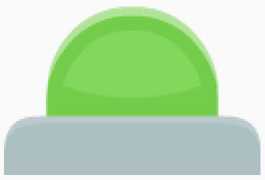
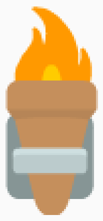
continues on next page

Table 19 – continued from previous page

grass_sprout.png	grassRight.png	brickGrey.png
sandHalf_mid.png	 snow.png	rock.png
 lockYellow.png	sandCliff_right.png	sandHill_right.png
grassMid.png	 torch2.png	 planetCenter_rounded.png
snowRight.png	 ladderTop.png	snowHill_left.png

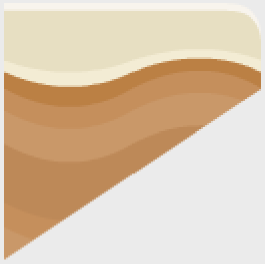
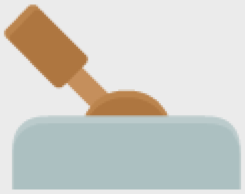
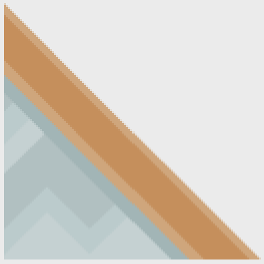
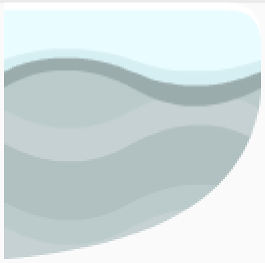
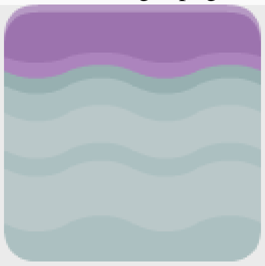
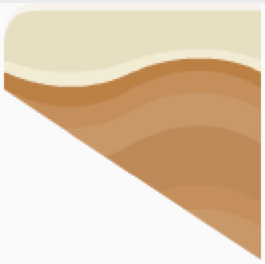
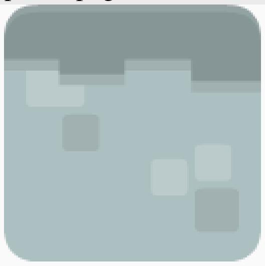
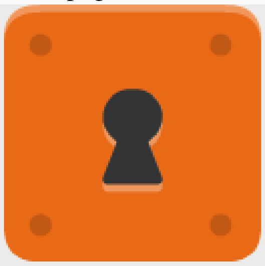
continues on next page

Table 19 – continued from previous page

 sandRight.png	dirtCorner_right.png	 doorClosed_top.png
 brickTextureWhite.png	stoneHill_left.png	 stoneRight.png
grassHalf_mid.png	 grassCliffAlt_left.png	 snowCliffAlt_left.png
 sandMid.png	lava.png	snowLeft.png
 stoneCliffAlt_left.png	 switchGreen.png	 torch1.png

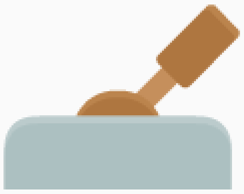
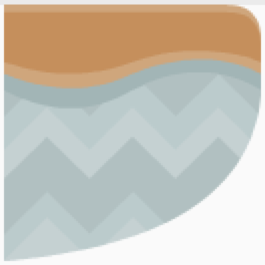
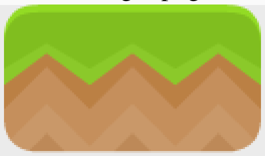
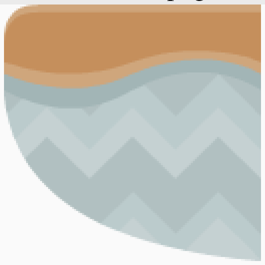
continues on next page

Table 19 – continued from previous page

 sandCliffAlt_right.png	 leverLeft.png	 dirtHill_left.png
planetHalf_mid.png	snowHalf_left.png	 snowCliff_right.png
grassCliff_left.png	bridgeA.png	 planet.png
 sandCliffAlt_left.png	snowCliff_left.png	 stone.png
grassHalf_right.png	grassLeft.png	 lockRed.png

continues on next page

Table 19 – continued from previous page

 leverRight.png	snowHalf_right.png	 dirtCliff_right.png
stoneCenter_rounded.png	planetHalf.png	 grassHalf.png
stoneHalf_left.png	grassCenter.png	stoneHalf_mid.png
mushroomRed.png	stoneHill_right.png	stoneCorner_right.png
 dirtCliff_left.png	planetHalf_right.png	snowCenter.png

continues on next page

Table 19 – continued from previous page

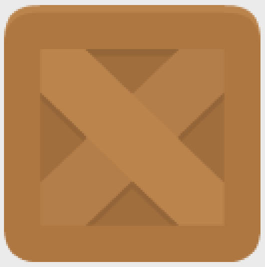
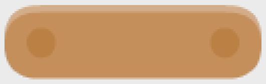
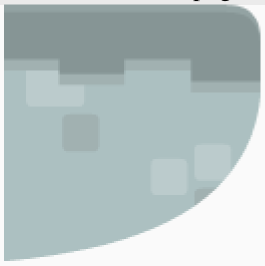
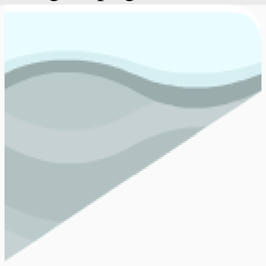
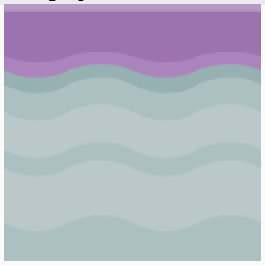
 boxCrate_double.png	 bridgeB.png	dirt.png
 stoneCliff_right.png	 snowCliffAlt_right.png	 planetMid.png

Table 20: :resources:images/enemies/

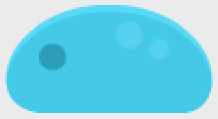
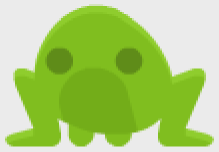
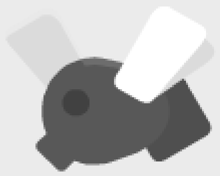
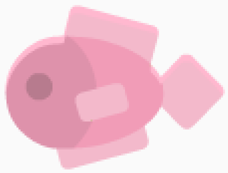
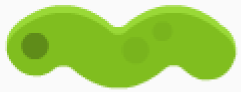
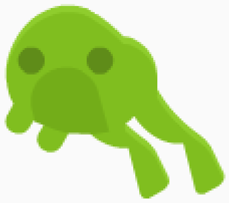
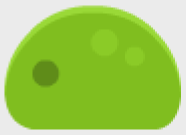
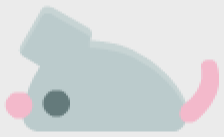
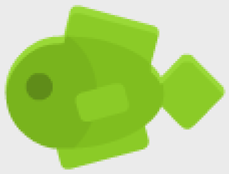
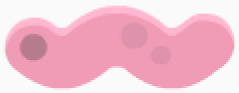
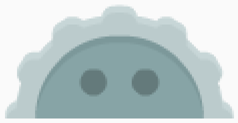
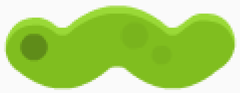
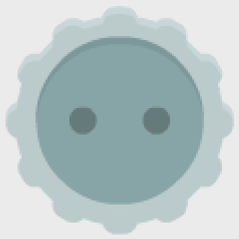
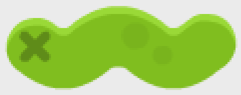
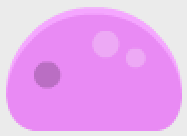
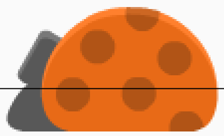
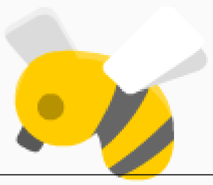
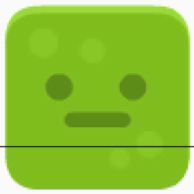
 slimeBlue_move.png	 frog.png	 fly.png
 fishPink.png	 wormGreen_move.png	 frog_move.png
 slimeGreen.png	 mouse.png	 fishGreen.png
 wormPink.png	 sawHalf.png	 wormGreen.png
 saw.png	 wormGreen_dead.png	 slimePurple.png
 ladybug.png	 bee.png	 slimeBlock.png

Table 21: :resources:images/animated_characters/male_person/

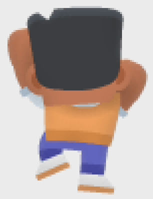
malePerson_walk2.png	malePerson_walk0.png	malePerson_jump.png
malePerson_walk6.png	malePerson_climb0.png	malePerson_walk4.png
malePerson_idle.png	malePerson_walk7.png	malePerson_walk1.png
malePerson_walk5.png	malePerson_fall.png	malePerson_walk3.png
 malePerson_climb1.png		

Table 22: :resources:images/animated_characters/robot/

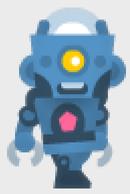
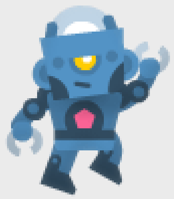
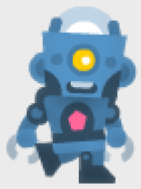
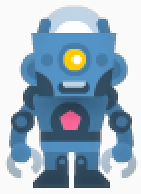
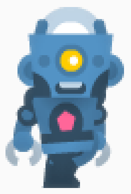
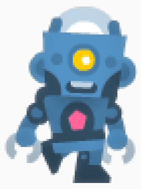
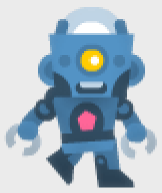
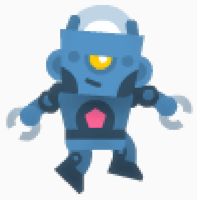
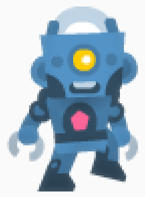
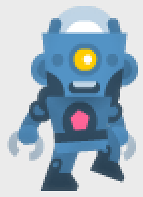
 robot_walk6.png	 robot_jump.png	 robot_walk1.png
 robot_idle.png	 robot_walk2.png	 robot_walk5.png
robot_walk0.png	robot_climb0.png	 robot_walk4.png
 robot_fall.png	 robot_walk3.png	robot_climb1.png
 robot_walk7.png		

Table 23: :resources:images/animated_characters/zombie/

zombie_fall.png	zombie_walk3.png	zombie_jump.png
zombie_walk4.png	zombie_idle.png	zombie_walk2.png
zombie_walk7.png	zombie_climb1.png	zombie_walk5.png
zombie_climb0.png	zombie_walk0.png	zombie_walk6.png
zombie_walk1.png		

Table 24: :resources:images/animated_characters/male_adventurer/

maleAdventurer_jump.png	maleAdventurer_walk3.png	maleAdventurer_climb1.png
maleAdventurer_climb0.png	maleAdventurer_walk1.png	maleAdventurer_walk2.png
maleAdventurer_walk0.png	maleAdventurer_walk5.png	maleAdventurer_idle.png
maleAdventurer_walk6.png	maleAdventurer_fall.png	maleAdventurer_walk4.png
maleAdventurer_walk7.png		

Table 25: :resources:images/animated_characters/female_adventurer/

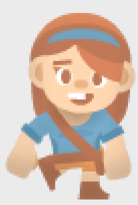
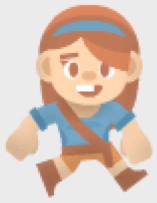
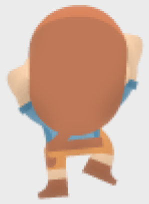
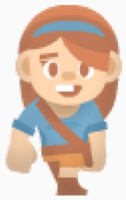
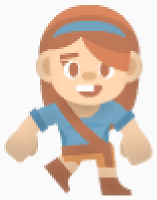
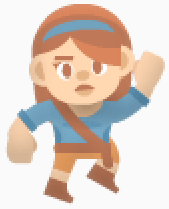
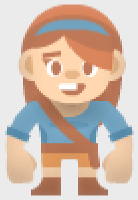
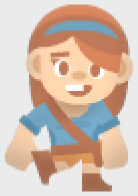
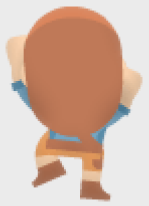
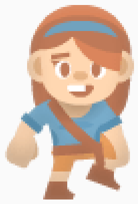
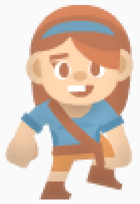
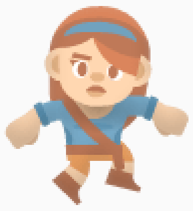
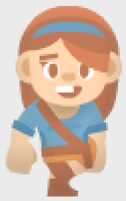
 femaleAdventurer_walk5.png	 femaleAdventurer_walk0.png	 femaleAdventurer_climb0.png
 femaleAdventurer_walk6.png	 femaleAdventurer_walk4.png	 femaleAdventurer_jump.png
 femaleAdventurer_idle.png	 femaleAdventurer_walk1.png	 femaleAdventurer_climb1.png
 femaleAdventurer_walk7.png	 femaleAdventurer_walk3.png	 femaleAdventurer_fall.png
 femaleAdventurer_walk2.png		

Table 26: :resources:images/animated_characters/female_person/

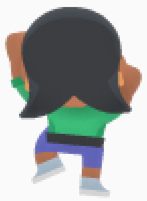
femalePerson_walk6.png	femalePerson_walk5.png	femalePerson_walk3.png
femalePerson_idle.png	femalePerson_walk4.png	femalePerson_walk1.png
femalePerson_fall.png	femalePerson_walk0.png	femalePerson_jump.png
 femalePerson_climb1.png	femalePerson_climb0.png	femalePerson_walk7.png
femalePerson_walk2.png		

Table 27: :resources:cache/

hit_box_cache.json

Table 28: :resources:music/

Table 29: :resources:video/

earth.mp4

Table 30: :resources:onscreen_controls/flat_dark/

cancel.png	key_square.png	pause.png
right.png	start.png	expand.png
checked.png	sound_off.png	wrench.png
pause_square.png	save.png	play.png
hamburger.png	a.png	x.png
search.png	music_off.png	left.png
star.png	gear.png	y.png
star_square.png	r.png	select.png
key_round.png	flatDark20.png	music_on.png

Table 31: :resources:onscreen_controls/shaded_dark/

cancel.png	key_square.png	pause.png
right.png	start.png	expand.png
checked.png	sound_off.png	wrench.png
pause_square.png	save.png	play.png
hamburger.png	a.png	x.png
search.png	music_off.png	left.png
gear.png	y.png	star_round.png
star_square.png	r.png	select.png
key_round.png	music_on.png	down.png

Table 32: :resources:onscreen_controls/flat_light/

cancel.png	key_square.png	pause.png
right.png	start.png	expand.png
checked.png	sound_off.png	wrench.png
pause_square.png	save.png	play.png
hamburger.png	a.png	x.png
search.png	music_off.png	left.png
gear.png	y.png	star_round.png
star_square.png	r.png	select.png
key_round.png	music_on.png	down.png

Table 33: :resources:onscreen_controls/shaded_light/

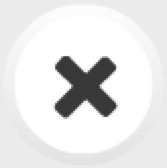
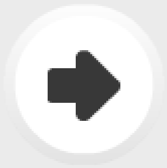
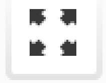
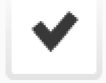
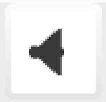
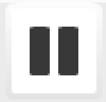
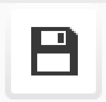
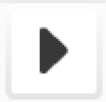
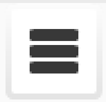
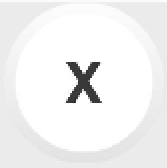
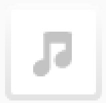
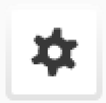
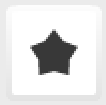
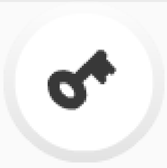
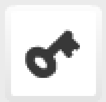
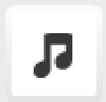
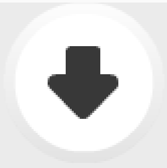
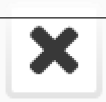
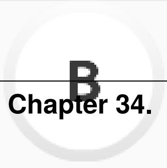
 cancel.png	pause.png	 right.png
start.png	 expand.png	 checked.png
 sound_off.png	wrench.png	 pause_square.png
 save.png	 play.png	 hamburger.png
a.png	 x.png	search.png
 music_off.png	left.png	 gear.png
y.png	star_round.png	 star_square.png
r.png	select.png	 key_round.png
 key.png	 music_on.png	 down.png
back.png	 close.png	 b.png

Table 34: :resources:tiled_maps/

map2_level_2.json	spritesheet.json	test_map_7.json
dirt.json	test_map_2.json	grass.json
items.json	test_objects.json	more_tiles.json
level_1.json	pymunk_test_map.json	test_map_6.json
standard_tileset.json	map2_level_1.json	level_2.json
map.json	test_map_3.json	test_map_5.json
map_with_ladders.json	test_map_1.json	

Table 35: :resources:sounds/

CHAPTER THIRTYFIVE

RELEASE NOTES

Keep up-to-date with the latest changes to the Arcade library by the release notes.

35.1 Version 3.0.0

Unreleased

You can grab pre-release versions from PyPi. See the available versions from the Arcade PyPi Release History.

Version 3.0.0 is a major update to Arcade. It is not 100% compatible with the 2.6 API.

35.1.1 Breaking Changes

These are the API changes which could require updates to existing code based on the 2.6 API. Some of these things may be repeated in the "Updates" section of these release notes, however we have compiled the breaking changes here for an easy reference. There may be other behavior changes that could break specific scenarios, but this section is limited to changes which directly changed the API in a way that is not compatible with how it was used in 2.6.

- *`arcade.Sprite.angle()`* has changed to clockwise. So everything rotates different now.
- Signature for Sprite creation has changed.
- The deprecated `update()` function has been removed from the *`Window`*, *`View`*, *`Section`*, and *`SectionManager`* classes. Instead, please use the *`on_update()`* function. It works the same as the `update` function, but has a `delta_time` parameter which holds the time in seconds since the last update.
- The `update_rate` parameter of *`Window`* can no longer be set to `None`. Previously it defaulted to `1 / 60` however could be set to `None`. The default is still the same, but setting it to None will not do anything.
- Sprites created from the *`TileMap`* class would previously set a key in the `Sprite.properties` dictionary named `type`. This key has been renamed to `class`. This is in keeping with Tiled's renaming of the key and following the Tiled format/API as closely as possible.
- The `arcade.text_pillow` and `arcade.text_pyglet` modules have been completely removed. The Pillow implementation is gone, and the Pyglet one has been renamed to just `arcade.text`. These modules were largely internal, but it is possible to have referenced them directly.
- Due to the above change and removal of the Pillow text implementation, the *`create_text_sprite()`* previously referred to the Pillow text implementation, and there was no easy way to create a sprite from Text with the pyglet implementation. This function has been re-worked to use the pyglet based text system. It has no API breaking changes, but the underlying functionality has changed a lot, so if you are using this function it may be worth checking the docs for it again. The main concern for a difference here would be if you are also using any custom `TextureAtlas`.

- The GUI package has been changed significantly.
- Buffered shapes (shape list items) have been moved to their own sub-module.
- *use_spatial_hash* parameter for *SpriteList* and *TileMap* is now a *bool* instead of *Optional[bool]*
- GUI
  - Removed `UIWrapper` this is now general available in *UILayout*
  - Removed `UIBorder` this is now general available in *UIWidget*
  - Removed `UIPadding` this is now general available in *UIWidget*
  - Removed `UITexturePane` this is now general available in *UIWidget*
  - Removed `UIAnchorWidget` replaced by `UIAnchorLayout`

35.1.2 Featured Updates

- Arcade now supports mixing Pyglet and Arcade drawing. This means you can, for example, use Pyglet batches. Pyglet batches can draw thousands of Pyglet objects with the cost and performance time of only a few.
- The code behind the texture atlas Arcade creates for each SpriteList has been reworked to be faster and more efficient. Reversed/flipped sprites are no longer duplicated.
- Added a new system for handling background textures (ADD MORE INFO)
- Arcade now supports OpenGL ES 3.1/3.2 and have been tested on the Raspberry Pi 4. Any model using the Cortex-A72 CPU should work. Note that you need fairly new Mesa drivers to get the new V3D drivers.

35.1.3 Changes

- *Window*
  - Removal of the `update` function in favor of *on_update()*
  - `update_rate` parameter in the constructor can no longer be set to `None`. Must be a float.
  - Added `draw_rate` parameter to constructor `__init__()`, this will control the interval that the *on_draw()* function is called at. This can be used with the pre-existing `update_rate` parameter which controls *on_update()* to achieve separate draw and update rates.
- *View*
  - Removal of the `update` function in favor of *on_update()*
- *Section* and *SectionManager*
  - Removal of the `update` function in favor of *arcade.Section.on_update()*
- GUI
  - *UIWidget*
    * Supports padding, border and background (color and texture)
    * Visibility: visible=False will prevent rendering of the widget. It will also not receive any UI events
    * Dropped `with_space_around()`
    * `UIWidget.with_` methods do not wrap the widget anymore, they only change the attributes
    * Fixed an blending issue when rendering the gui surface to the screen

        - Support nine patch information to draw background texture
        - Performance improvements
        - **Removed some attributes from public interface, use `UIWidget.with_` methods**
            - `UIWidget.border_width`
            - `UIWidget.border_color`
            - `UIWidget.bg_color`
            - `UIWidget.bg_texture`
            - `UIWidget.padding_top`
            - `UIWidget.padding_right`
            - `UIWidget.padding_bottom`
            - `UIWidget.padding_left`
        - Update and add example code.
        - Iterable (providing direct children)
    - New widgets:
        - *`UIDropdown`*
        - *`UIImage`*
        - *`UISlider`*
        - `UIButtonRow` (PR1580 and PR1253)
    - Arcade *`Property`*:
        - Properties are observable attributes (supported: primitive, list and dict). Listener can be bound with `bind()`
    - All UILayout`s support ``size_hint`, size_hint_min, size_hint_max.
        - *`UIBoxLayout`*
        - *`UIAnchorLayout`*
        - *`UIGridLayout`* (PR1478)
    - Replaces deprecated usage of *`draw_text()`*
    - Misc Changes
        - `arcade.color_from_hex_string()` changed to follow the CSS hex string standard
        - Windows Text glyph are now created with DirectWrite instead of GDI
        - Removal of various deprecated functions and parameters
        - OpenGL examples moved to examples/gl from `experiments/examples`
- Sprites
    - The method signature for `arcade.Sprite.__init__()` has been changed. (May break old code.)
    - The sprite code has been cleaned up and broken into parts.
    - *`arcade.Sprite.angle()`* now rotates clockwise. Why it ever rotated the other way, and why it lasted so long, we do not know.
- Controller Input

- Previously controllers were usable via the `arcade.joysticks` module. This module is still available in 3.0. However, it should largely be seen as deprecated for most people who want basic controller support. This module existed basically just as an alias to the Pyglet joysticks module. We now have a new `arcade.controller` module, which is similarly just an alias to Pyglet's newer Controller API. This change should make a much wider selection of controllers able to work with Arcade, and provide newer functionality and be easier to use for most cases than the joystick module. The joystick module may still be useful if you need specialty controllers such as racing wheels or flight sticks. All existing example code has been updated to use the new controller API.

- Text
  - Complete removal of the old PIL based text system. In Arcade 2.6 we had largely switched to the newer Pyglet based system, however there were still remnants of the PIL implementation around. Namely the `create_text_sprite()` function which has been updated to use the Pyglet system. There's no API breaking change here but if you are using the function it would be worth reading the new docs for it, as there are some different considerations surrounding use of a custom `TextureAtlas` if you are also doing that. This function should now be much much faster than the old PIL implementation. The texture generation happens almost entirely on the GPU now.
  - As part of this move, the `arcade.text_pillow` module has been removed completely, and the `arcade.text_pyglet` module has been re-named just be `arcade.text`.
  - `draw_text()` and `Text` both now accept a `start_z` parameter. This will allow advanced usage to set the Z position of the underlying Label. This parameter defaults to 0 and does not change any existing usage.
- OpenGL
  - Support for OpenGL ES 3.1 and 3.2. 3.2 is fully supported, 3.1 is only supported if the `EXT_geometry_shader` extension is provided by the driver. This is part of the minimum spec in 3.2 so it is guaranteed to be there. This is the only optional extension that Arcade needs to function with 3.1.

    As an example, the Raspberry Pi 4b only supports OpenGL ES 3.1, however does provide this extension, so is fully compatible with Arcade.
  - Textures now support immutable storage for OpenGL ES compatability.
  - Arcade is now using Pyglet's projection and view matrix. All functions setting matrices will update the Pyglet window's `view` and `projection` attributes. Arcade shaders is also using Pyglet's `WindowBlock` UBO.
  - Uniforms are now set using `glProgramUniform` instead of `glUniform` when the extension is available.
  - Fixed many implicit type conversions in the shader code for wider support.
  - Added `front_face` property on the context for configuring front face winding order of triangles
  - Added `cull_face` property on the context for configuring what triangle face to cull
  - Added support for bindless textures
  - Added support for 64 bit integer uniforms
  - Added support for 64 float uniforms
- *TileMap*
  - Added support Tiles defined as a sub-rectangle of an image. See Tiled 1.9 Release Notes for more information on this feature.
  - Changed the `Sprite.properties` key "type" to "class" to stay in line with Tiled's re-naming of this key in their API.
  - You can now define a custom texture atlas for SpriteLists created in a TileMap. You can provide a map default to the `texture_atlas` parameter of the `Tilemap` class or the `load_tilemap()` function. This

will be used by default on all layers, however it can be overridden on a per-layer basis as defined by the new `texture_atlas` key in the `layer_options` dictionary. If no custom atlas is provided, then the global default atlas will be used (This is how it works pre-Arcade 3.0).

  - Fix for animated tiles from sprite sheets
  - TextureAtlas: Added `sync_texture_image` method to sync the texture in the atlas back into the internal pillow image in the `arcade.Texture`.
  - TextureAtlas: Added `get_texture_image` method to get pixel data of a texture in the atlas as a pillow image.

- Collision Detection
  - Collision detection is now even faster.
  - Remove Shapely for collision detection as 3.11 is faster without it.
- Shape list
  - Add in `arcade.create_triangles_strip_filled_with_colors()`
  - Moved all buffered items that can be added to a shape list to *arcade.shape_list*
- Documentation
  - *How-To Example Code* code page has been reorganized
  - CONTRIBUTING.md page has been updated
  - Improve background_parallax example

Special thanks to Einar Forselv Darren Eberly, pushfoo, Maic Siemering, Cleptomania, Aspect1103, Alejandro Casanovas, Ibrahim, Andrew, Alexander, kosvitko, and pvcraven for their contributions to this release. Also, thanks to everyone on the Pyglet team! We depend heavily on Pyglet's continued development.

35.2 Version 2.6.16

Released 2022-Sept-24

- Support Tiled 1.9 via PyTiled Parser 2.2.0 (#1324)
- Headless rendering with EGL should now work again
- Fix code highlights in two examples
- Fix data tables in quick index. (#1312)
- Fix issues running in headless mode
- Update pymunk physics engine to return pre handler (#1322)
- Bump Pyglet version to 2.0dev23
- Few PEP-8 fixes
- Fix perspective example

Note: Development continues on version 2.7, which will be another leap forward in Arcade development. Feel free to check out the 'development' branch for the 2.7 changes.

35.3 Version 2.6.15

Released 2022-Jun-03

- Pin Pygments version to get around a Pygments/Furo incompatibility. (#1224).
- Fix Google analytics ID
- Bump Pyglet version to 2.0.dev18. (Thanks Pyglet!)
- Fix API colors for Furo theme

35.4 Version 2.6.14

Released 2022-May-18

- Various Improvements
  - Allow specifying hit box parameters in `load_textures()` and `load_spritesheet()`
  - `Camera` should no longer apply zoom on the z axis
  - Promote using `arcade.View.on_show_view()` in examples and tutorials
  - The arcade window and views now expose `arcade.Window.on_enter()` `arcade.Window.on_leave()`. These events are triggered when the mouse enters and leaves the window area.
  - Sections should now also support mouse enter/leave events
  - Hit box calculation methods should raise a more useful error message when the texture is not RGBA.
  - Slight optimization in updating sprite location in SpriteList
  - Removed all remaining references to texture transforms
  - Removed the broken `Sprite.__lt__` method
  - Added `get_angle_radians()`
  - Removed `Texture.draw_transformed`
  - Add support for changing the pitch while playing a sound. See the *speed* parameter in `arcade.play_sound()`.
  - Set better blending defaults for arcade GUI
  - Can now create a texture filled with a single color. See `Texture.create_filled()`. The Sprite class will use this when creating a solid colored sprite.
  - Bump version numbers of Sphinx, Pillow to current release as of 17-May.
  - Bump Pyglet version to 2.0.dev16. (Thanks Pyglet!)
- Shadertoy
  - Added `Shadertoy.delta_time` alias for `time_delta` (`iTimeDelta`)
  - Support the `iFrame` uniform. Set frame using the `arcade.experimental.ShadertoyBase.frame` attribute
  - Support the `iChannelTime` uniform. Set time for each individual channel using the `arcade.experimental.ShadertoyBase.channel_time` attribute.
  - Support the `iFrameRate` uniform. Set frame rate using the `arcade.experimental.ShadertoyBase.frame_rate` attribute

  - Support the `iDate` uniform. This uniform will be automatically set. See `arcade.experimental.ShadertoyBase._get_date()`
  - Support the `iChannelResolution` uniform. This uniform will be automatically set
  - Added example using video with shadertoy
  - Improve Shadertoy docstrings + unit tests
- Docs / Tutorials / Examples
  - Updated install docs
  - Added tutorial for compiling an arcade game with Nuika
  - Improved/extended shadertoy tutorials
  - Added example using textures with shadertoy
  - Added sprite rotation examples
  - Clarified the difference between *`arcade.View.on_show_view()`* and *`arcade.View.on_show()`*
  - Improved UIManager docstrings
  - Various annotation and docstring improvements
  - Fixed several broken links in docs
  - We're now building PDF/EPUB docs
- OpenGL
  - Added new method for safely setting shader program uniforms: *`arcade.gl.Program.set_uniform_safe()`*. This method will ignore `KeyError` if the uniform doesn't exist. This is often practical during development because most GLSL compilers/linkers will remove uniforms that is determined to not affect the outcome of a shader.
  - Added new method for safely setting a uniform array: *`arcade.gl.Program.set_uniform_array_safe()`*. This is practical during development because uniform arrays are in most cases shortened by GLSL compiler if not all array indices are used by the shader.
  - Added `arcade.gl.Texture.swizzle`. This can be used to reorder how components are read from the texture by a shader making it easy to crate simple effects or automatically convert BGR pixel formats to RGB when needed.
  - Added ray marching example with fragment shader
  - Allow reading framebuffer data with 2 and 4 byte component sizes
  - Simplified texture atlas texture coordinates to make them easier to use in custom shaders.
  - Support dumping the atlas texture as RGB
  - Support dumping the atlas texture with debug lines showing texture borders
  - We no longer check `GL_CONTEXT_PROFILE_MASK` due to missing support in older drivers. Especially GL 3.1 drivers that can in theory run arcade
  - Various shader cleanups
- Experimental
  - Added a simple profiler class

Special thanks to Vincent Poulailleau Ian Currie Mohammad Ibrahim, pushfoo, Alejandro Casanovas, Darren Eberly, pvcraven and Einar Forselv for their contributions to this release. Also, thanks to everyone on the Pyglet team! We depend heavily on Pyglet's continued development.

35.5 Version 2.6.13

Released 2022-Mar-25

- New Features
  - Arcade can now run in headless mode on linux servers opening more possibilities for users in for example the data science community (#1107). See *Headless Arcade* for more information.
- Bugfixes
  - The random text glitching issue especially affecting users with iGPUs is finally resolved in pyglet. For that reason we have upgraded to the pyglet 2.0a2 release.
  - Fixed an issue causing `arcade.draw_circle_filled()` and `arcade.draw_circle_outline()` to always render with 3 segments on some iGPUs.
  - Fixed an issue causing interactive widgets to unnecessarily re-draw when hovering or pressing them. This could cause performance issues.
  - SectionManager's `on_show_view` was never called when showing a view
- Various Improvements
  - `arcade.load_font()` now supports resource handles
  - `PhysicsEngineSimple` can now take an iterable of wall spritelists
  - Sprite creation is now ~6-8% faster.
  - Removed warning about missing shapely on startup
  - Window titles are now optional. If no window title is specified the title will be the absolute path to the python file it was created in. This was changed because of the new headless mode.
  - Removed `arcade.quick_run`. This function had no useful purpose.
  - Added clear method to UIManager (#1116)
  - Updated from Pillow 9.0.0 to 9.0.1
- Tilemap
  - Rectangle objects which are empty(have no width or height) will now be automatically converted into single points.
  - The Tile ID of a sprite can be access with `sprite.properties["tile_id"]`. This refers to the local ID of the tile within the Tileset. This value can be used to get the tile info for a given Sprite created from loading a tilemap.
- Docs
  - Added python version support info to install instructions (#1122)
  - Fixed typo in `append_texture()` docstring(#1126)
  - Improved the raycasting tutorial (#1124)
  - Replace mentions of 3.6 on Linux install page (#1129)
  - Fix broken links in the homepage (#1139)
  - Lots of other improvements to docstrings throughout the code base
  - General documentation improvements
- OpenGL

- `arcade.gl.Geometry` now supports transforming to multiple buffers.
- Added and improved examples in `experimental/examples`.
- Major improvements to API docs

Special thanks to Mohammad Ibrahim, pushfoo, Alejandro Casanovas, Maic Siemering, Cleptomania, pvcraven and einarf for their contributions to this release. Also, thanks to everyone on the Pyglet team! We depend heavily on Pyglet's continued development.

35.6 Version 2.6.12

Released 2022-Mar-20

- General:
  - Bugfix: `check_for_collision_with_list()` selected the wrong collision algorithm. This could affect performance.
  - Bugfix: GPU collision detection show now work on older MacBooks
  - Added `arcade.Text.draw_debug()` that will visualize the content area of the text and the anchor point. This can be useful to understand the text anchoring.
  - `arcade.Text` now has a `left`, `right` `top` and `bottom` attribute for getting the pixel locations of the content borders.
  - Added performance warning for `arcade.draw_text()`. Using `arcade.Text` is a lot faster. We have also promoted the use of text objects in examples.
  - Removed the deprecated `arcade.create_text` function
  - `UITextureButton.texture_pressed` now returns the pressed texture, not the texture
- Documentation
  - Work on *Shader Toy - Glow*.
  - Docstring improvements throughout the code base
  - Many examples are cleaned up
- OpenGL
  - `arcade.gl.Buffer` is guaranteed to contain zero byte values on creation.
  - Expose `Limits` in `arcade.gl.Context.info` and document all limit values
  - Added limit: `MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS`
  - `arcade.gl.Buffer.read()` now reads the correct number of bytes when only `offset` parameter is passed.
  - Improved compute shader examples
  - Support uniform blocks in compute shaders
  - Bug: `arcade.gl.Context.enabled` now properly reverts to the original context flags
  - Many docstring improvements in the `arcade.gl` module
  - Bugfix: Query objects ignored creation parameters
  - `arcade.gl.ComputeShader` is now part of the gl module
  - `arcade.gl.ComputeShader` was added to docs

  - Expose and document `arcade.gl.context.ContextStats`

Special thanks to MrWardKKHS, pvcraven and einarf for their contributions to this release. Also, thanks to everyone on the Pyglet team! We depend heavily on Pyglet's continued development.

Also thanks to:

- DragonMoffon for arcade.gl testing and feedback
- bunny-therapist discovering collision bug
- Robert Morris for making us aware of the MacBook issue

35.7 Version 2.6.11

Released 2022-Mar-17

- Sections - Add support to divide window into sections. (Thanks janscas for the contribution.)
  - Add `arcade.Section` to the API.
  - Add `arcade.SectionManager` to the API.
  - Add examples on how to use: *Sectioning a View*
- New Example Code:
  - Add parallax example: background_parallax.
  - Add GUI flat button styling example: gui_flat_button_styled.
  - Add perspective example.
- New functionality:
  - Add `arcade.get_angle_degrees()` function.
  - Add easing functions and example. See easing_example_1 and easing_example_2.
  - Add `arcade.Sprite.facePoint()` to face sprite towards a point.
- Fixes:
  - Fixed issue #1074 to prevent a crash when opening a window.
  - Fixed issue #978, copy button in examples moved to the left to prevent it disappearing.
  - Fixed issue #967, CRT example now pulls from resources so people don't have to download image to try it out.
  - PyMunk sample map now in resources so people don't have to download it.
  - `arcade.draw_points()` no longer draws the points twice, improving performance.
- Documentation:
  - Update *Pygame Comparison*.
  - Improve `Sprite.texture` docs.
  - When building Arcade docs, script now lets us know what classes don't have docstrings.
  - Spelling/typo fixes in docs.
- Misc:
  - Update `arcade.Sprite` to use decorators to declare properties instead of the older method.

  - #1095, Improvements to *arcade.Text* and its documentation. We can now also get the pixel size of a Text contents though `content_width`, `content_height` and `content_size`.
  - Force GDI text on windows until direct write is more mature.
  - Optimized text rendering and text rotation
  - *arcade.draw_text()* and *arcade.Text* objects now accepts any python object as text and converts it into a string internally if needed.
  - *SpriteList* now exposes several new members that used to be private. These are lower level members related to the underlying geometry of the spritelist and can be used by custom shaders to do interesting things blazingly fast. SpriteList interaction example with shaders can be found in the experimental directory. Members include *write_sprite_buffers_to_gpu()*, *geometry*, *buffer_positions*, *buffer_sizes*, *buffer_textures*, *buffer_colors*, *buffer_angles* and *buffer_indices*

- OpenGL:
  - Added support for indirect rendering. This is an OpenGL 4.3 feature. It makes us able to render multiple meshes in the the same draw call providing significant speed increases in some use cases. See *arcade.gl.Geometry.render_indirect()* and examples in the experimental directory.
  - Added support for unsigned integer uniform types
  - `arcade.gl.Geometry.transform` no longer takes a mode parameter.

Special thanks to einarf, eruvanos, janscas, MrWardKKHS, DragonMoffon, pvcraven, for their contributions to this release. Also, thanks to everyone on the Pyglet team! We depend heavily on Pyglet's continued development.

35.8 Version 2.6.10

Released 2022-Jan-29

- Sprites
  - Collision checking against one or more sprite lists can use the GPU via a 'transform' for much better performance. The *arcade.check_for_collision_with_list()* and *arcade.check_for_collision_with_lists()* methods now support selection between spatial, GPU, and CPU methods of detection.
  - Added *clear()* for resetting/clearing a spritelist. This will iterate and remove all sprites by default, or do a faster *O(1)* clear. Please read the api docs to find out what version fits your use case.
  - *SpriteList* now supports setting a global color and alpha value. The new *color*, *color_normalized*, *alpha* and *alpha_normalized* will affect every sprite in the list. This global color value is multiplied by the individual sprite colors.
  - The *Sprite* initializer now also accepts `None` value for `hit_box_algorithm` in line with the underlying texture method.
  - Fixed a bug causing sprites to have incorrect scale when passing a texture during creation.
  - Removed the texture transform feature in sprites. This feature no longer makes sense since arcade 2.6.0 due to the new texture atlas feature.
- Tiled Maps
  - Fixed issue #1068 (#1069) where loaded rectangular hit box was wrong.
  - Add better error for infinite tile maps

   - Added `SpriteList.properties` and properties from Image and Tile layers will automatically be loaded into that when loading a Tiled map
- General
   - `Window.current_camera` will now hold a reference to the currently active camera. This will be set when calling *`arcade.Camera.use()`*, if no camera is active then it will be `None`.
   - `Window.clear` can now clear a sub-section of the screen through the new optional `viewport` parameter.
   - *`arcade.Window.clear()`* can now take normalized/float color values
   - The new *`arcade.View.clear()`* method now clears the current window. This can be used as a shortcut *`arcade.Window.clear()`* when inside of a View class.
   - Add support for custom resource handles
   - Add support for anisotropic filtering with textures.
   - Clearing the window should always clear the entire window regardless of camera / viewport setup (unless a scissor box is set)
- Documentation
   - Change examples so instead of `arcade.start_render()` we use `self.clear()`. The start render function was confusing people. #1071
   - Fix a bunch of links that were incorrectly pointing to old pvcraven instead of pythonarcade. #1063
   - Update pyinstaller instructions
   - Various documentation improvements and updates
- `arcade.gl`
   - Fixed a bug were out attributes in transforms was not properly detected with geometry shaders
   - Fixed a bug were specifying vertex count wasn't possible with transforms when the vertex array has an index buffer bound.
   - The *`Query`* object now allows for selecting what specific queries should be performed
   - Fixed a issue causing the wrong garbage collection mode to activate during context creation
   - Viewport values for the default framebuffer now applies pixel ratio by default
   - Scissor values for the default framebuffer now applies pixel ratio by default
- `arcade.gui`
   - *`UIBoxLayout`* supports now align in constructor (changing later requires a *UIBoxLayout.trigger_full_render()*).
   - *`UIBoxLayout`* supports now space_between in constructor.
   - *`UIManager`* fix #1067, consume press and release mouse events
   - UIManager *`add()`* returns added child
   - UILayout `add()` returns added child
   - UIWidget *`add()`* returns added child
   - New method in UIManager: *`walk_widgets()`*
   - New method in UIManager: *`get_widgets_at()`*
   - New method in UIWidget: *`move()`*

Special thanks to Cleptomania, einarf, eruvanos, nrukin, Jayman2000, pvcraven, for their contributions to this release. Also, thanks to everyone on the Pyglet team! We depend heavily on Pyglet's continued development.

35.9 Version 2.6.9

Released on 2022-Jan-13

- Bump version of Pillow from 8.4 to 9.0.0 due to security vulnerability in Pillow.

35.10 Version 2.6.8

Released on 2021-Dec-25

- The Shapely library is now optional. The shapely library uses native code to make operations such as collision detection and some other geometry operations faster. However they have not updated their binaries to support Python 3.10 on macOS and Windows. If Shapely is installed, Arcade will use that library. Otherwise it will fall back to slower, but Python-only code. See: https://github.com/shapely/shapely/issues/1215
- `TileMap` changes:

  There are no API changes to the TileMap class, however full support for TMX maps, TSX tilesets, and TX object templates has been added thanks to pytiled-parser 2.0. You should be able to load these formats with 0 change to your code, and use all the same features that were available with JSON maps.

  This update also includes the ability to cross-load JSON and TMX maps/tilesets. Meaning you can have a JSON map load a TSX tileset, or have a TMX map load a JSON tileset.

  You don't ever need to explicitly set or configure a format to use, it will be automatically determined based on the file you pass in. It is determined based on the actual content of the file, and not the filetype, so if you give it a `.json` file that actually contains TMX, or vice versa, it will still work without problem.
- Update Pyglet to 2.0.dev13 which fixes a bug where `on_resize` wasn't getting called.
- Added a compute shader tutorial.

Special thanks to Cleptomania, einarf, pvcraven, for their contributions to this release. Also, thanks to everyone on the Pyglet team! We depend heavily on Pyglet's continued development.

35.11 Version 2.6.7

Released on 2021-Dec-15

- This version updates Pyglet to 2.0dev12. Programs WILL NOT RUN with prior versions of Pyglet.
- *`Window`* changes:
  - Added `enable_polling` option to constructor. If enabled then `window.keyboard` and `window.mouse` will be activated and able to be used to poll input by accessing them as if they were a dictionary. This option is enabled by default. See #1038

    `window.keyboard` can be polled using the values from `arcade.key`.

    `window.mouse` can be polled using the following values:

    * 1: Left click
    * 2: Right click

    * 3: Middle click
    * "x": X position
    * "y": Y position

- *Camera* changes:
  - Defaults the viewport width and height to the window size if they are set to 0 now, since you cannot have a size of 0 in any direction due to projection calculation. This means that if you do not provide those arguments to the constructor it will default to the window size. See #1041
- *TileMap* changes:
  - Added support for layer position offsets. This allows passing a tuple containing an X and Y offset that will be applied to each Sprite/Object within the layer. You can set this via an `offset` parameter in the `layer_options` dict, or you can supply a global offset to the map which will be applied to all layers via the `offset` parameter of either `arcade.load_tilemap` or to the TileMap constructor directly. Layer specific offsets will override the global default if both are set. See #1048
  - Added a new error message for JSONDecodeError exceptions, a common problem when tilesets are TSX but maps are JSON. This change simply provides a more clear error of the most likely cause of the problem so users don't have to dig as much.
- Text
  - Reverted the extra guards around text rendering that was implemented in 2.6.6. This turned out to cause slowdowns where text was being used heavily. Work is still ongoing to fix the remaining issues with text.
- Docs Fixes:
  - See #1033 and #1046
  - #1043 Update moving platforms example.

Special thanks to Cleptomania, einarf, pvcraven, mlr07, pushfoo, for their contributions to this release. Also, thanks to everyone on the Pyglet team! We depend heavily on Pyglet's continued development.

35.12 Version 2.6.6

Released on 2021-Dec-04

- `TileMap` changes:
  - Added `tiled_map` parameter to init function of TileMap class. It allows to pass an already parsed map from from pytiled-parser to it. Previously it could only be used with raw files and would handle the parsing automatically. If a pre-parsed map is passed to this, the `map_file` parameter will simply be ignored. This addition makes working with pre-parsed maps from a World file possible.
- Text
  - Added extra guards around text rendering calls to hopefully reduce glitchy text rendering. Work is still ongoing to fix the remaining issues with text.
- Window:
  - Added `samples` parameter so user can specify antialiasing quality.
  - The arcade window should fall back to no antialiasing if the window creation fails. Some drivers/hardware don't support it. For example when running arcade in WSL or services like Repl.it.
- SpriteList

  - Optimization: Empty spritelists created before the window or created with `lazy=True` no longer automatically initialize internal OpenGL resources for empty spritelists and will instead immediately leave the `draw()` method.

- UI
  - Add experimental UI styles dataclasses for UIWidget styling.
  - Add UISlider, which provides a general slider element with some basic functionality
  - Fix UIInputText rendering
- Sound
  - Pyglet audio drivers can now be overridden using the `ARCADE_SOUND_BACKENDS` environment variable for debug purposes. It expects a comma separated string with driver names.
- OpenGL
  - From version 2.6.6 Arcade is no longer using the `auto` garbage collection mode for OpenGL resources. This mode has the same behavior as the Python garbage collection. Instead we're now using the `context_gc` mode were resources are released every time `Window.flip()` is called (every frame by default). This solves many problems such as threads in your project or external libraries suddenly trying to garbage collect OpenGL objects while this is only possible in the main thread. This should not cause any problems for most users.
  - Added `Context.copy_framebuffer`. This can be used to copy framebuffers with or without multisampling to another framebuffer. This makes us able to do offscreen rendering with multisampling.
  - `Texture` s can now be created with multisampling by passing the `samples` parameter. This should only be used for attachments to framebuffers. The `Texture` object now also has a `samples` property (read only).
- Examples
  - Update mini-map example
  - Update scrolling camera example
  - Update google analytics code in docs
  - Remove some less-than-useful examples in the example code section
  - Update platformer example
  - Update windows install instructions
  - Update sample games to show more sample games
  - Improve CRT filter tutorial
  - New example code on how to follow a path
  - Added Game of Life example using shaders
- Documentation
  - Added API docs for `arcade.gl`
  - `ArcadeContext` should now show inherited members
  - Edge artifact page now encourage using `pixelated` argument instead of importing OpenGL enums from pyglet

Special thanks to einarf, pvcraven, Cleptomania, eruvanos, for their contributions to this release. Also, thanks to everyone on the Pyglet team! We depend heavily on Pyglet's continued development.

35.13 Version 2.6.5

Released on 2021-Nov-5

- Increased pyglet's default atlas size for text glyphs to remove text flickering and various other artifacts. This issue will be fixed in future versions of pyglet.
- Fixed as issue causing all sprites to use the same texture on some Macs.
- Improved doc for setting the viewport.

Special thanks to einarf, pushfoo, for their contributions to this release.

35.14 Version 2.6.4

Released on 2021-Nov-3

- Python 3.10 updates. Dependent library versions have been updated to include Python 3.10 support. All libraries appear to support 3.10 except Shapely 1.8.0 on the Windows platform. Until those binaries are released, 3.10 support for Windows is still not there.
- *`SpriteList`* additions:
  - A `visible` attribute has been added to this class. If set to `False`, when calling `draw()` on the SpriteList it will simply return and do nothing. Causing the SpriteList to not be drawn.
  - SpriteList now has a `lazy` (bool) parameter causing it to not create internal OpenGL resources until the first draw call or until SpriteList's *`initialize()`* is called. This means that sprite lists and sprites can now be created in threads.
  - Fixes/optimized *`reverse()`* and *`shuffle()`* methods.
  - Added *`sort()`* method. This is identical to Python's `list.sort` but are many times faster sorting your sprites.
  - Removed noisy warning message when spritelists were created before the window
  - Fixed an issue with *`insert()`* when trying to insert sprites past an index greater than the current length. It could cause inserted sprites to be invisible.
- *`Sprite`* changes:
  - Added `arcade.Sprite.visible` property for quickly making sprites visible/invisible. This is simply a shortcut for changing the alpha value.
  - Optimization: Sprites should now take ~15% less memory and be ~15% faster to create
  - *`SpriteCircle`* and `SpriteSolidColor` textures are now cached internally for better performance.
- *`PhysicsEnginePlatformer`* Optimization:

  A `walls` parameter has been added to this class. The new intention for usage of this class is for static(non-moving) sprites to be sent to the `walls` parameter, while moving platforms should be sent to the `platforms` parameter. Properly differentiating between these parameters can result in extreme performance benefits. Sprites added to `platforms` are O(n) whereas Sprites added to `walls` are O(1). This has been tested with anywhere from 100 to 500k+ Sprites, and the physics engine shows no measurable difference between those scenarios.

  We have also removed the ability to send a single Sprite to the `platforms`, `ladders`, and `walls` parameters of this class. This is a use case which results in some improper usage and unnecessary slowdowns. These parameters will now only accept SpriteLists or an iterable such as a list containing SpriteLists. If you are currently using this functionality, you just need to add your Sprite to a SpriteList and provide that instead.

The simple platformer tutorial has already been updated to make use of this optimization.

- *Scene* is additions:
  - The Scene class is now sub-scriptable, previously in order to retrieve a SpriteList from Scene, you needed to use either `Scene.name_mapping` or `Scene.get_sprite_list`. We have now added the ability to access it by sub-scripting the Scene object directly, like `spritelist = my_scene["My Layer"]`
  - Added `on_update()` method. Previously Scene only had `update()`. Both of these methods simply call the corresponding one on each SpriteList, however previously you could not do this with `on_update()`. The difference between these methods is that `on_update()` allows passing a delta time, whereas `update()` does not.
- `TileMap` additions and fixes:
  - When loading a Tiled map Arcade will now respect if layers are visible or not. If a layer is not visible in Tiled, the SpriteList created for it will use the new `visible` attribute to control it. This means that when creating a Scene from a TileMap, this will automatically be respected as well.
  - Fixed support for parallax values on layers. Currently there is no support to do anything with these out of the box, you'd need to manually pull the values and do something based on them, however previously the map would not load if the values were changed from the default. This has been fixed in pytiled-parser and we have updated our version in Arcade accordingly.
  - Removed a lingering debug tactic of printing the class name of custom SpriteList classes when loading a TileMap.
- UI
  - `UIInputText` now supports both RGB and RGBA text color
- Text
  - Several text related bugs have been resolved in pyglet, the underlying library we now use for text drawing. This has been a fairly time consuming task over several weeks and we hope the new pyglet based text system will stabilize from now on. Arcade is an early adopter of pyglet 2.0 currently using a pre-release
  - The *Text* object is now usable and is preferred over *arcade.draw_text()* in many cases for performance reasons.
  - Text related functions should now have better documentation
- Misc:
  - Added support to the *View* class for *on_resize()*
  - Many docstring improvements. Initializer docstrings have now been moved to the class docstring ensuring they will always show up in the generated api docs.
  - Added some new sections under advanced docs related to OpenGL, textures and texture atlas
  - New utility function `color_from_hex_string()` that will turn a hex string into a color.
  - Bug: Removed a lingering debug key `F12` that showed the contents of the global texture atlas
  - Several improvements to typing and PEP-8. Plus automated tests to help keep things in good shape.
  - Added `run()` shortcut in `arcade.Window`. Usage: `MyWindow().run()`
  - Addition of *PymunkException* class for throwing Pymunk errors in the Pymunk physics engine.
  - The *check_for_collision_with_lists()* function will now accept any Iterable(List, Tuple, Set, etc) containing SpriteLists.
- Lower level rendering API:

- Fixed a problem causing Geometry / VertexArray to ignore `POINTS` primitive mode when this is set as default.
- Added support for compute shaders. We support writing to textures and SSBOs (buffers). Examples can be found in `arcade/experimental/examples`
- Fixed a crash when drawing with geometry shaders due to referencing a non-existent enum

Special thanks to einarf, pvcraven, pushfoo, Cleptomania, Olliroxx, mlr07, yegarti, Jayman2000 for their contributions to this release.

Special thanks to Benjamin and caffeinepills for their help to squash bugs in pyglet 2.0.

35.15 Version 2.6.3

Released on 2021-Sept-21

- Bug fix, use a signed in as the 'killed' index. #965
- Fix dead links on getting started page See #960
- Fix some doc language that mixed function/method vocabulary. See #963
- Some initial work on compute and camera shader work. Not done yet.
- Fixed a bug causing the sprite geometry shader to not compile in some platforms
- Fixed a bug related to texture bleeding with sprites. Texture atlases now pad the texture borders with repeating pixel data to combat this. It should make sprites look much better when scrolling, zooming and on hidpi displays. #959
- Added hack for some gui text not appearing (pyglet 2.0 bug)
- UIMessageBox should now respect the width and height of the widget
- `SpriteList.draw`: Added `pixelated` (bool) argument as a shortcut to setting nearest interpolation
- `SpriteList.draw`: The arguments are now better exposed in docs
- `Sprite.draw` now has the same blending and interpolation argument as `SpriteList.draw`
- Upgraded to pyglet 2.0dev9

35.16 Version 2.6.2

Released on 2021-Sept-18

- Support for custom classes that subclass Sprite for tiles in TileMap objects. See #942
- Update PymunkPhysicsEngine to work with any direction of gravity rather than just downward. See #940
- Update library versions we depend on. PIL, Pymunk, etc.
- Fix the card game example code. See #951
- Fix for drawing small circles not using enough segments. See #950
- A lot of documentation links in the .py files were old and not updated to the RTD way, fixed now.
- `arcade.key` was missing from the documentation quick index. Fixed.
- Fixed a rendering issue with sprites on M1 Macs

- Fix caret not showing up in input box
- Lots of type-hint fixes

35.17 Version 2.6.1

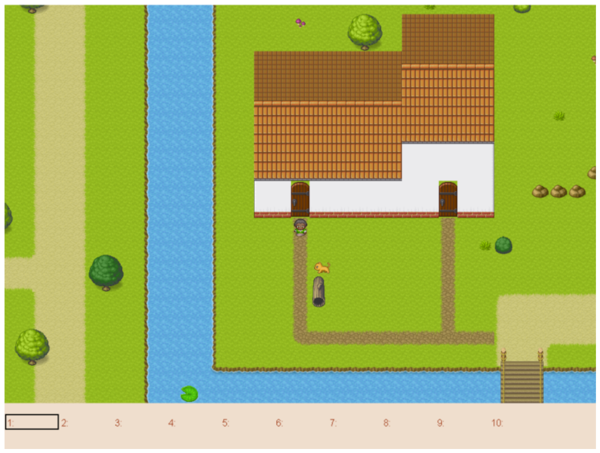
35.17.1 Fixes

- Removed type annotations which were introduced in Python 3.8 to fix compatibility with Python 3.7 and 3.6
- Fixed flickering on static drawing. See #858

35.18 Version 2.6.0

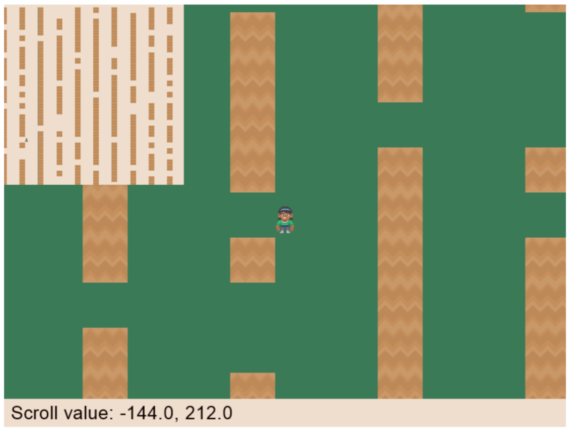
Version 2.6.0 is a major update to Arcade. It is not 100% backwards compatible with the 2.5 API. Updates were made to text rendering, tiled map support, sprites, shaders, textures, GUI system, and the documentation.

- Tiled Map Editor support has been overhauled.
  - Arcade now uses the .json file format for maps created by the Tiled Map Editor rather than the TMX format. Tile sets and other supporting files need to all be saved in .json format. The XML based formats are no longer supported by Arcade.
  - Arcade now supports a minimum version of Tiled 1.5. Maps saved with an older version of Tiled will likely work in most scenarios, but for all features the minimum version we can support is 1.5 due to changes in the Tiled map format.
  - Feature-support for Tiled maps has been improved to have near 100% parity with Tiled itself.
  - See *Simple Platformer* for a how-to, Tiled usage starts at Chapter 9.
  - See Community RPG or Community Platformer for a more complex example program.



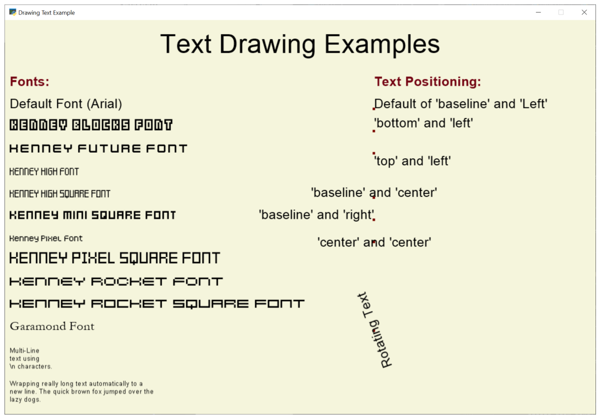
- Texture atlases have been introduced, texture management has been improved.
  - A sprite list will create and use its own texture atlas.
  - This introduces a new `arcade.TextureAtlas` class that is used internally by SpriteList.
  - Sprites with new textures can be added to a sprite list without the delay. Arcade 2.5 had a delay caused by rebuilding its internal sprite sheet.

- As a side effect, sprites can only belong to one sprite list that renders.
- The texture atlas portion of a sprite can be drawn to, and quickly updated on the GPU side.
  - To demonstrate, there is a new minimap example that creates a sprite that has a dynamic minimap projected onto it.



Scroll value: -144.0, 212.0

- Revamped text rendering done by *arcade.draw_text()*. Rather than use Pillow to render onto an image, Arcade uses Pyglet's text drawing system. Text drawing is faster, higher resolution, and not prone to memory leaks. Fonts are now specifed by the font name, rather than the file name of the font.
  - Fonts can be dynamically loaded with *arcade.load_font()*.
  - Kenney.nl's TTF are now included as build-in resources.
  - See the drawing_text example.



Drawing Text Example

Text Drawing Examples

Fonts:

Default Font (Arial)

KENNEY BLOCKS FONT

KENNEY FUTURE FONT

KENNEY HIGH FONT

KENNEY HIGH SQUARE FONT

KENNEY MINI SQUARE FONT

Kenney Pixel Font

KENNEY PIXEL SQUARE FONT

KENNEY ROCKET FONT

KENNEY ROCKET SQUARE FONT

Garamond Font

Multi-Line
text using
\n characters.

Wrapping really long text automatically to a
new line. The quick brown fox jumped over the
lazy dogs.

Text Positioning:

Default of 'baseline' and 'Left'

'bottom' and 'left'

'top' and 'left'

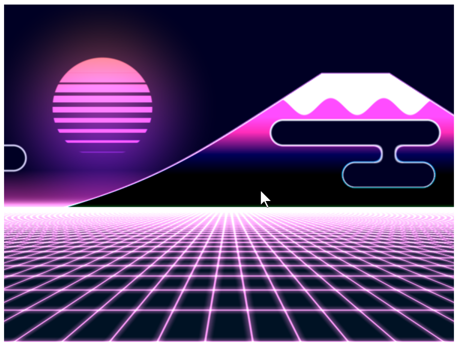
'baseline' and 'center'

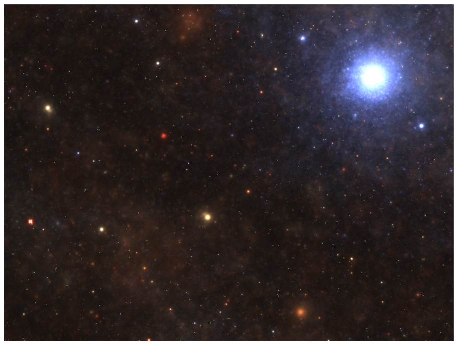
'baseline' and 'right'

'center' and 'center'

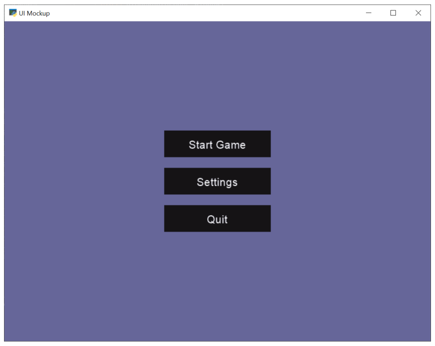
Rotating Text

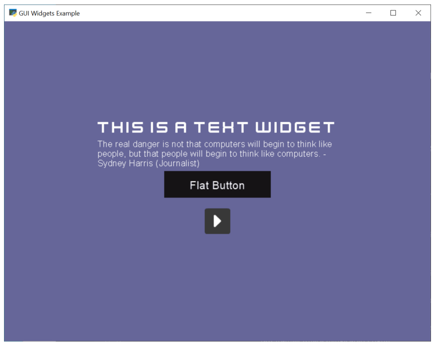
- SpriteList optimizations.
  - Sprites now draw even faster than before. On an Intel i7 with nVidia 980 Ti graphics card, 8,000+ moving sprites can be drawn while maintaining 60 FPS. The same machine can only do 2,000 sprites with Pygame before FPS drops.
- Shadertoy support.
  - Shadertoy.com is a website that makes it easier to write OpenGL shaders.
  - The new `arcade.Shadertoy` class makes it easy to run and interact with these shaders in Arcade.
  - See *Shader Toy - Glow* and Asteroids.



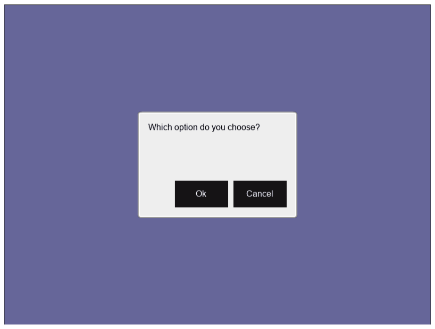


- Reworked GUI





The real danger is not that computers will begin to think like people, but that people will begin to think like computers. - Sydney Harris (Journalist)



- UIElements are replaced by UIWidgets
- Option to relative pin widgets on screen to center or border (supports resizing)
- Widgets can be placed on top of each other
- Overlapping widgets properly handle mouse interaction
- Fully typed event classes
- Events contain source widget
- ScrollableText widgets (more to come)
- Support for Sprites within Widgets
- Declarative coding style for borders and padding *widget.with_border(. . .)*
- Automatically place widgets vertically or horizontally (*UIBoxLayout*)
- Dropped support for YAML style files
- Better performance and limited memory usage
- More documentation (*GUI Concepts*)
- Available Elements:
  - `UIWidget`:
    - `UIFlatButton` - 2D flat button for simple interactions (hover, press, release, click)
    - `UITextureButton` - textured button (use `arcade.load_texture()`) for simple interactions (hover, press, release, click)
    - `UILabel` - Simple text, supports multiline
    - `UIInputText` - field to accept user text input
    - `UITextArea` - Multiline scrollable text widget.
    - `UISpriteWidget` - Embeds a Sprite within the GUI tree
  - `UILayout`:
    - `UIBoxLayout` - Places widgets next to each other (vertical or horizontal)
  - `UIWrapper`:
    - `UIPadding` - Add space around a widget
    - `UIBorder` - Add border around a widget
    - `UIAnchorWidget` - Used to position UIWidgets relative on screen
  - Constructs

            · *UIMessageBox* - Popup box with a message text and a few buttons.

        * Mixins

            · *UIDraggableMixin* - Makes a widget draggable.

            · *UIMouseFilterMixin* - Catches mouse events that occure within the widget boundaries.

            · *UIWindowLikeMixin* - Combination of *UIDraggableMixin* and *UIMouseFilterMixin*.

    - WIP * UIWidgets contain information about preferred sizes * UILayouts can grow or shrink widgets, to adjust to different screen sizes

- Scene Manager.

    - There is now a new *arcade.Scene* class that can be used to manage SpriteLists and their draw order. This can be used in place of having to draw multiple spritelists in your draw function.

    - Contains special integration with `arcade.TileMap` using *arcade.Scene.from_tilemap()* which will automatically create an entire scene from a loaded tilemap in the proper draw order.

    - See *Simple Platformer* for an introduction to this concept, and it is used heavily throughout that tutorial.

- Camera support

    - Easy scrolling with *arcade.Camera*

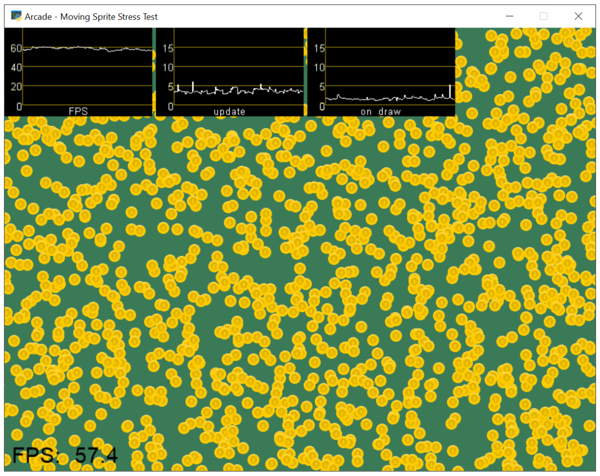
    - For an example of this see the example: sprite_move_scrolling.

    - Automatic camera shake can be added in, see the example: sprite_move_scrolling_shake.

    - Several other examples and tutorials make use of this class, like *Simple Platformer*.

- Add a set of functions to track performance statistics. See *Performance Information*.

- Added the class *arcade.PerfGraph*, a subclass of Sprite that will graph FPS or time to process a dispatch-able event line 'update' or 'on_draw'.

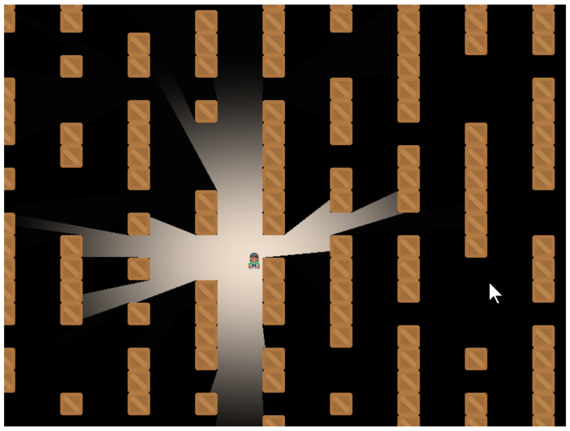


- Documentation

    - Lots of individual documentation updates for commands.

    - The *API Index* has been reorganized to be easier to find commands, and the individual API documentation pages have been broken into parts, so it isn't one large monolithic page.

    - New tutorial for *Ray-casting Shadows*.



  - New tutorial for *Shader Toy - Glow*.
  - Revamped tutorial: *Simple Platformer*.
  - Revamped minimap example: minimap.
  - Moved from AWS hosting to read-the-docs hosting so we can support multiple versions of docs.
  - New example showing how to use the new performance statistics API: performance_statistics_example
  - New example: gui_widgets
  - New example: gui_flat_button
  - New example: gui_ok_messagebox
- API commands
  - `arcade.get_pixel()` supports getting RGB and RGBA color value
  - `arcade.get_three_color_float()` Returns colors as RGB float with numbers 0.0-1.1 for each color
  - `arcade.get_four_color_float()` Returns colors as RGBA float with numbers 0.0-1.1 for each color
- Better PyInstaller Support

  Previously our PyInstaller hook only fully functioned on Windows, with a bit of functionality on Linux. Mac was just completely unsupported and would raise an UnimplementedError if you tried.

  Now we have full out of the box support for PyInstaller with Windows, Mac, and Linux.

  See *Bundling a Game with PyInstaller* for an example of how to use it.
- Sound

  The sound API remains unchanged, however general stability of the sound system has been greatly improved via updates to Pyglet.
- Fix for A-star path finding routing through walls

Special thanks to:

- einarf for performance improvements, texture atlas support, shader toy support, text drawing support, advice on GUI, and more.
- Cleptomania for Tiled Map support, sound support, and more.
- eruvanos for the original GUI and all the GUI updates.
- benmoran56 and everyone that contributes to the excellent Pyglet library we use so much.

35.19 Version 2.5.7

Released on 2021-May-25

35.19.1 Fixes

- The arcade gui should now respect the current viewport
- Fixed an issue with UILabel allocating large amounts of textures over time consuming a lot of memory
- Fixed an issue with the initial viewport sometimes being 1 pixel too small causing some artifacts
- Fixed a race condition in `Sound.stop()` sometimes causing a crash
- Fixed an issue in requirements causing issues for poetry
- Fixed an error reporting issue when reaching maximum texture size

35.19.2 New Features

replit.com

Arcade should now work out of the box on replit.com. We detect when arcade runs in replit tweaking various settings. One important setting we disable is antialiasing since this doesn't work well with software rendering.

Alternative Garbage Collection of OpenGL Resources

`arcade.gl.Context` now supports an alternative garbage collection mode more compatible with threaded applications and garbage collection of OpenGL resources. OpenGL resources can only be accessed or destroyed from the same thread the window was created. In threaded applications the Python garbage collector can in some cases try to destroy OpenGL objects possibly causing a hard crash.

This can be configured when creating the `arcade.Window` passing in a new `gc_mode` parameter. By default this parameter is `"auto"` providing the default garbage collection we have in Python.

Passing in `"context_gc"` on the other hand will move all "dead" OpenGL objects into `Context.objects`. These can be garbage collected manually by calling `Context.gc()` in a more controlled way in the the right thread.

35.20 Version 2.5.6

Version 2.5.6 was released 2021-03-28

- Fix issue with PyInstaller and Pymunk not allowing Arcade to work with bundling
- Fix some PyMunk examples
- Update some example code. Highlight PyInstaller instructions

35.21 Version 2.5.5

Version 2.5.5 was released 2021-02-23

- Fix setting an individual sprite list location to a new sprite not working

35.22 Version 2.5.4

Version 2.5.4 was released 2021-02-19

- Fix for soloud installer hook
- Add fishy game on example page
- Fix but around framebuffer creation not properly restoring active frame buffer
- Fix for but where TextureRenderTarget creates FBO twice
- Updated pinned version numbers for dependent libraries
- MyPy fixes
- Minor improvements around SpriteList list operations
- Fix for physics engine getting stuck on a corner

35.23 Version 2.5.3

Version 2.5.3 was released 2021-01-27

- Fix memory leak when removing sprites from sprite list
- Fix solitaire example using old hitbox parameter
- Fix/improve tetris example
- Fix for camera2d.scroll_x

35.24 Version 2.5.2

Version 2.5.2 was released 2020-12-27

- Improve schedule/unschedule docstrings
- Fix Sound.get_length
- Raise error if there are multiple instances of a streaming source
- Fix background music example to match new sound API
- Update main landing page for docs
- Split sprite platformer tutorial into multiple pages
- Add ‘related projects’ page
- Add ‘adventure’ sample game link
- Add resources for top-down tank images

- Add turn-and-move example
- Fix name of sandCorner_left.png
- Update tilemap to error out instead of continuing if we can't find a tile
- Improve view tutorial
- Generate error rather than warning if we can't find image or sound file
- Specify timer resolution in Windows

35.25 Version 2.5.1

Version 2.5.1 was released 2020-12-14

- Fix bug with sound where panning wasn't working on Windows machines.
- Fix for create_lines_with_colors
- Fix for pegboard example, coin image too small
- Fix for create_ellipse dimensions being too big.
- Add visible kwarg to window constructor
- Fix some type-checking errors found by mypy.
- Update API docs

35.26 Version 2.5

Version 2.5 was released 2020-12-09

(Note, libraries Arcade depends on do not work yet with Python 3.9 on Mac. Mac users will need to use Python 3.6, 3.7 or 3.8.)

- Changing to Pyglet from Soloud for Sound
- Optimize has_line_of_sight using shapely
- Update setuptools configuration to align with PEP 517/518
- Changed algorithm for checking for polygon collisions
- Fix incorrect PyInstaller data file path handling docs
- Fix for hitbox not scaling
- Add support for pyinstaller on Linux

General

- SpriteList.draw now supports a blend_function parameter. This opens up for drawing sprites with different blend modes.
- Bugfix: Sprite hit box didn't properly update when changing width or height
- GUI improvements (eruvanos needs to elaborate)
- Several examples was improved
- Improvements to the pyinstaller tutorial

- Better pin versions of depended libraries
- Fix issues with simple and platformer physics engines.

Advanced

- Added support for tessellation shaders
- `arcade.Window` now takes a `gl_version` parameter so users can request a higher OpenGL version than the default `(3, 3)` version. This only be used to advanced users.
- Bugfix: Geometry's internal vertex count was incorrect when using an index buffer
- We now support 8, 16 and 32 bit index buffers
- Optimized several draw methods by omitting `tobytes()` letting the buffer protocol do the work
- More advanced examples was added to `arcade/experimental/examples`

Documentation

- Add conway_alpha example showing how to use alpha to control display of sprites in a grid.
- Improve documentation around sound API.
- Improve documentation with FPS and timing stats example.
- Improve moving platform docs a bit in *Simple Platformer* tutorial.

35.27 Version 2.4.3

Version 2.4.3 was released 2020-09-30

General

- Added PyInstalled hook and tutorial
- ShapeLists should no longer share position between instances
- GUI improvements: new UIImageToggle

Low level rendering API (arcade.gl):

- ArcadeContext now has a load_texture method for creating opengl textures using Pillow.
- Bug: Fixed an issue related to drawing indexed geometry with offset
- Bug: Scissor box not updating when using framebuffer
- Bug: Fixed an issue with pack/unpack alignment for textures
- Bug: Transforming geometry into a target buffer should now work with byte offset
- Bug: Duplicate sprites in 'check_for_collision_with_list' Issue #763
- Improved docstrings in arcade.gl

35.28 Version 2.4.2

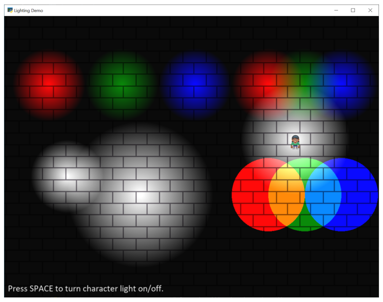
Version 2.4.2 was released 2020-09-08

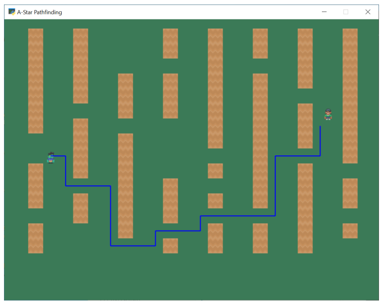
- Enhancement: `draw_hit_boxes` new method in `SpriteList`.
- Enhancement: `draw_points` now significantly faster
- Added UIToggle, on/off switch
- Add example showing how to do GPU transformations with the mouse
- Create buttons with default size/position so size can be set after creation.
- Allow checking if a sound is done playing Issue 728
- Add an early camera mock-up
- Add `finish` method to `arcade.gl.context`.
- New example arcade.experimental.examples.3d_cube (experimental)
- New example arcade.examples.camera_example
- Improved UIManager.unregister_handlers(), improves multi view setup
- Update `preload_textures` method of `SpriteList` to actually pre-load textures
- GUI code clean-up Issue 723
- Update downloadable .zip for for platformer example code to match current code in documentation.
- Bug Fix: `draw_point` calculates wrong point size
- Fixed draw_points calculates wrong point size
- Fixed create_line_loop for thickness !=
- Fixed pixel scale for offscreen framebuffers and read()
- Fixed SpriteList iterator is stateful
- Fix for pixel scale in offscreen framebuffers
- Fix for UI tests
- Fix issues with FBO binding
- Cleanup Remove old examples and code

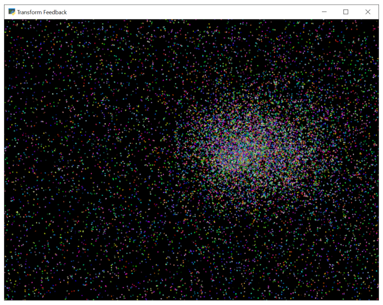
35.29 Version 2.4

Arcade 2.4.1 was released 2020-07-13.

Arcade version 2.4 is a major enhancement release to Arcade.







35.29.1 Version 2.4 Major Features

- Support for defining your own frame buffers, shaders, and more advanced OpenGL programming. New API in Arcade Open GL.
  - Support to render to frame buffer, then re-render.
  - Use frame buffers to create a 'glow' or 'bloom' effect
  - Use frame-buffers to support lights: light_demo.
- New support for style-able GUI elements.
- PyMunk engine for platformers. See tutorial: *Pymunk Platformer*.
- AStar algorithm for finding paths. See `astar_calculate_path` and `AStarBarrierList`.
  - For an example of using the A-Star algorithm, see astar_pathfinding.

35.29.2 Version 2.4 Minor Features

New functions/classes:

- Added get_display_size() to get resolution of the monitor
- Added Window.center_window() to center the window on the monitor.
- Added has_line_of_sight() to calculate if there is line-of-sight between two points.
- Added SpriteSolidColor class that makes a solid-color rectangular sprite.
- Added SpriteCircle class that makes a circular sprite, either solid or with a fading gradient.
- Added `get_distance` function to get the distance between two points.

New functionality:

- Support for logging. See *Logging*.
- Support volume and pan arguments in play_sound
- Add ability to directly assign items in a sprite list. This is particularly useful when re-ordering sprites for drawing.
- Support left/right/rotated sprites in tmx maps generated by the Tiled Map Editor.
- Support getting tmx layer by path, making it less likely reading in a tmx file will have directory confusion issues.
- Add in font searching code if we can't find default font when drawing text.
- Added `arcade.Sprite.draw_hit_box` method to draw a hit box outline.
- The *`arcade.Texture`* class, *`arcade.Sprite`* class, and `arcade.tilemap.process_layer` take in `hit_box_algorithm` and `hit_box_detail` parameters for hit box calculation.

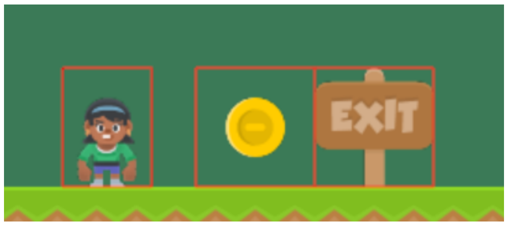


Fig. 1: hit_box_algorithm = "None"

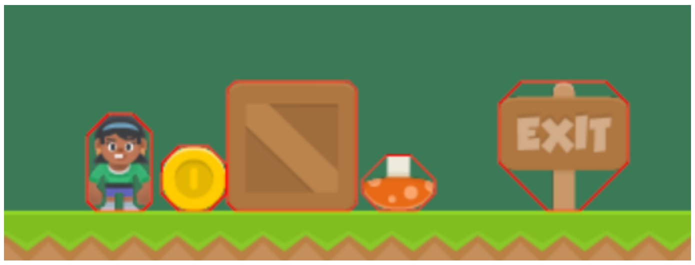


Fig. 2: hit_box_algorithm = "Simple"

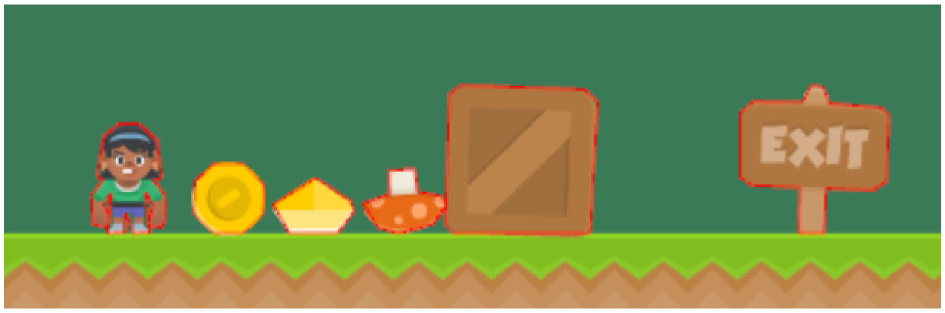


Fig. 3: hit_box_algorithm = "Detailed"

35.29.3 Version 2.4 Under-the-hood improvements

General

- Simple Physics engine is less likely to 'glitch' out.
- Anti-aliasing should now work on windows if `antialiasing=True` is passed in the window constructor.
- Major speed improvements to drawing of shape primitives, such as lines, squares, and circles by moving more of the work to the graphics processor.
- Speed improvements for sprites including gpu-based sprite culling (don't draw sprites outside the screen).
- Speed improvements due to shader caching. This should be especially noticeable on Mac OS.
- Speed improvements due to more efficient ways of setting rendering states such as projection.
- Speed improvements due to less memory copying in the lower level rendering API.

OpenGL API

A brand new low level rendering API wrapping OpenGL 3.3 core was added in this release. It's loosely based on the ModernGL API, so ModernGL users should be able to pick it up fast. This API is used by arcade for all the higher level drawing functionality, but can also be used by end users to really take advantage of their GPU. More guides and tutorials around this is likely to appear in the future.

A simplified list of features in the new API:

- A *`Context`* and *`arcade.ArcadeContext`* object was introduced and can be found through the `window.ctx` property. This object offers methods to create opengl resources such as textures, programs/shaders, framebuffers, buffers and queries. It also has shortcuts for changing various context states. When working with OpenGL in arcade you are encouraged to use `arcade.gl` instead of `pyglet.gl`. This is important as the context is doing quite a bit of bookkeeping to make our life easier.
- New `Texture` class supporting a wide variety of formats such as 8/16/32 bit integer, unsigned integer and float values. New convenient methods and properties was also added to change filtering, repeat mode, read and write data, building mipmaps etc.
- New *`Buffer`* class with methods for manipulating data such as simple reading/writing and copying data from other buffers. This buffer can also now be bound as a uniform buffer object.
- New *`Framebuffer`* wrapper class making us able to render any content into one more more textures. This opens up for a lot of possibilities.
- The *`Program`* has been expanded to support geometry shaders and transform feedback (rendering to a buffer instead of a screen). It also exposes a lot of new properties due to much more details introspection during creation. We also able to assign binding locations for uniform blocks.

- A simple glsl wrapper/parser was introduced to sanity check the glsl code, inject preprocessor values and auto detect out attributes (used in transforms).
- A higher level type `Geometry` was introduced to make working with shaders/programs a lot easier. It supports using a subset of attributes defined in your buffer description by inspecting the the program's attributes generating and caching compatible variants internally.
- A `Query` class was added for easy access to low level measuring of opengl rendering calls. We can get the number samples written, number of primitives processed and time elapsed in nanoseconds.
- Added support for the buffer protocol. When `arcade.gl` requires byte data we can also pass objects like numpy array of pythons `array.array` directly not having to convert this data to bytes.

35.29.4 Version 2.4 New Documentation

- New Tutorial: *Pymunk Platformer*
- New Tutorial: *Using Views for Start/End Screens*
- New Tutorial: *Solitaire*
- New Tutorial: *GPU Particle Burst*
- Several new and updated examples on *How-To Example Code*
- New performance testing project
- A lot of improvements to https://learn.arcade.academy
- Instructional videos added to for https://learn.arcade.academy

35.29.5 Version 2.4 'Experimental'

There is now an `arcade.experimental` module that holds code still under development. Any code in this module might still have API changes.

35.29.6 Special Thanks

Special thanks to Einar Forselv and Maic Siemering for their significant work in helping put this release together.

35.30 Version 2.3.15

Release Date: Apr-14-2020

- Bug Fix: Fix invalid empty text width Issue 633
- Bug Fix: Make sure file name is string before checking resources Issue 636
- Enhancement: Implement Size and Rotation for Tiled Objects Issue 638
- Documentation: Fix incorrect link to 'sprites following player' example

35.31 Version 2.3.14

Release Date: Apr-9-2020

- Bug Fix: Another attempt at fixing sprites with different dimensions added to same SpriteList didn't display correctly Issue 630
- Add lots of unit tests around Sprites and texture loading.

35.32 Version 2.3.13

Release Date: Apr-8-2020

- Bug Fix: Sprites with different dimensions added to same SpriteList didn't display correctly Issue 630

35.33 Version 2.3.12

Release Date: Apr-8-2020

- Enhancement: Support more textures in a SpriteList Issue 332

35.34 Version 2.3.11

Release Date: Apr-5-2020

- Bug Fix: Fix procedural_caves_bsp.py
- Bug Fix: Improve Windows install docs Issue 623

35.35 Version 2.3.10

Release Date: Mar-31-2020

- Bug Fix: Remove unused AudioStream and PlaysoundException from __init__
- Remove attempts to load ffmpeg library
- Add background music example
- Bug Fix: Improve Windows install docs Issue 619
- Add tutorial on edge artifacts Issue 418
- Bug Fix: Can't remove sprite from multiple lists Issue 621
- Several documentation updates

35.36 Version 2.3.9

Release Date: Mar-25-2020

- Bug Fix: Fix for calling SpriteList.remove Issue 613
- Bug Fix: get_image not working correctly on hi-res macs Issue 594
- Bug Fix: Fix for "shiver" in simple physics engine Issue 614
- Bug Fix: Fix for create_line_strip Issue 616
- Bug Fix: Fix for volume control Issue 610
- Bug Fix: Fix for loading SoLoud under Win64 Issue 615
- Fix jumping/falling texture in platformer example
- Add tests for gui.theme Issue 605
- Fix bad link to arcade.color docs

35.37 Version 2.3.8

Release Date: Mar-09-2020

- Major enhancement to sound. Uses SoLoud cross-platform library. New features include support for sound volume, sound stop, and pan left/right.

35.38 Version 2.3.7

Release Date: Feb-27-2020

- Bug Fix: If setting color of sprite with 4 ints, also set alpha
- Enhancement: Add image for code page 437
- Bug Fix: Fixes around hit box calcs Issue 601
- Bug Fix: Fixes for animated tiles and loading animated tiles from tile maps Issue 603

35.39 Version 2.3.6

Release Date: Feb-17-2020

- Enhancement: Add texture transformations Issue 596
- Bug Fix: Fix off-by-one issue with default viewport
- Bug Fix: Arcs are drawn double-sized Issue 598
- Enhancement: Add `get_sprites_at_exact_point` function
- Enhancement: Add code page 437 to default resources

35.40 Version 2.3.5

Release Date: Feb-12-2020

- Bug Fix: Calling sprite.draw wasn't drawing the sprite if scale was 1 Issue 575
- Add unit test for Issue 575
- Bug Fix: Changing sprite scale didn't cause sprite to redraw in new scale Issue 588
- Add unit test for Issue 588
- Enhancement: Simplify using built-in resources Issue 576
- Fix for failure on on_resize(), which pyglet was quietly ignoring
- Update `rotate_point` function to make it more obvious it takes degrees

35.41 Version 2.3.4

Release Date: Feb-08-2020

- Bug Fix: Sprites weren't appearing Issue 585

35.42 Version 2.3.3

Release Date: Feb-08-2020

- Bug Fix: set_scale checks height rather than scale Issue 578
- Bug Fix: Window flickers for drawing when not derived from Window class Issue 579
- Enhancement: Allow joystick selection in dual-stick shooter Issue 571
- Test coverage reporting now working correctly with TravisCI
- Improved test coverage
- Improved documentation and typing with Texture class
- Improve minimal View example

35.43 Version 2.3.2

Release Date: Feb-01-2020

- Remove scale as a parameter to load_textures because it is not unused
- Improve documentation
- Add example for acceleration/friction

35.44 Version 2.3.1

Release Date: Jan-30-2020

- Don't auto-update sprite hit box with animated sprite
- Fix issues with sprite.draw
- Improve error message given when trying to do a collision check and there's no hit box set on the sprite.

35.45 Version 2.3.0

Release Date: Jan-30-2020

- Backwards Incompatability: arcade.Texture no longer has a scale property. This property made things confusing as Sprites had their own scale attribute. This seemingly small change required a lot of rework around sprites, sprite lists, hit boxes, and drawing of textured rectangles.
- Include all the things that were part of 2.2.8, but hopefully working now.
- Bug Fix: Error when calling Sprite.draw() Issue 570
- Enhancement: Added Sprite.draw_hit_box to visually draw the hit box. (Kind of slow, but useful for debugging.)

35.46 Version 2.2.9

Release Date: Jan-28-2020

- Roll back to 2.2.7 because bug fixes in 2.2.8 messed up scaling

35.47 Version 2.2.8

Release Date: Jan-27-2020

- Version number now contained in one file, rather than three.
- Enhancement: Move several GitHub-listed enhancements to the .rst enhancement list
- Bug Fix: Texture scale not accounted for when getting height Issue 516
- Bug Fix: Issue with text cut off if it goes below baseline Issue 515
- Enhancement: Allow non-cached texture creation, fixing issue with resizing Issue 506
- Enhancement: Physics engine supports rotation
- Bug Fix: Need to better resolve collisions so sprite doesn't get hyper-spaces to new weird spot Issue 569
- Bug Fix: Hit box not getting properly created when working with multi-texture player sprite. Issue 568
- Bug Fix: Issue with text_sprite and anchor y of top Issue 567
- Bug Fix: Issues with documentation

35.48 Version 2.2.7

Release Date: Jan-25-2020

- Enhancement: Have draw_text return a sprite Issue 565
- Enhancement: Improve speed when changing alpha of text Issue 563
- Enhancement: Add dual-stick shooter example Issue 301
- Bug Fix: Fix for Pyglet 2.0dev incompatability Issue 560
- Bug Fix: Fix broken particle_systems.py example Issue 558
- Enhancement: Added mypy check to TravisCI build Issue 557
- Enhancement: Fix typing issues Issue 537
- Enhancement: Optimize load font in draw_text Issue 525
- Enhancement: Reorganize examples
- Bug Fix: get_pixel not working on MacOS Issue 539

35.49 Version 2.2.6

Release Date: Jan-20-2020

- Bug Fix: particle_fireworks example is not running with 2.2.5 Issue 555
- Bug Fix: Sprite.pop isn't reliable Issue 531
- Enhancement: Raise error if default font not found on system Issue 432
- Enhancement: Add space invaders clone to example list Issue 526
- Enhancement: Add sitemap to website
- Enhancement: Improve performance, error handling around setting a sprite's color
- Enhancement: Implement optional filtering parameter to SpriteList.draw Issue 405
- Enhancement: Return list of items hit during physics engine update Issue 401
- Enhancement: Update resources documentation Issue 549
- Enhancement: Add on_update to sprites, which includes delta_time Issue 266
- Enhancement: Close enhancement-related github issues and reference them in the new enhancement list.

35.50 Version 2.2.5

Release Date: Jan-17-2020

- Enhancement: Improved speed when rendering non-buffered drawing primitives
- Bug fix: Angle working in radians instead of degrees in 2.2.4 Issue 552
- Bug fix: Angle and color of sprite not updating in 2.2.4 Issue 553

35.51 Version 2.2.4

Release Date: Jan-15-2020

- Enhancement: Moving sprites now 20% more efficient.

35.52 Version 2.2.3

Release Date: Jan-12-2020

- Bug fix: Hit boxes not getting updated with rotation and scaling. Issue 548 This update depricates Sprite.points and instead uses Sprint.hit_box and Sprint.get_adjusted_hit_box
- Major internal change around not having `__init__` do `import *` but specifically name everything. Issue 537 This rearranded a lot of files and also reworked the quickindex in documentation.

35.53 Version 2.2.2

Release Date: Jan-09-2020

- Bug fix: Arcade assumes tiles in tileset are same sized Issue 550

35.54 Version 2.2.1

Release Date: Dec-22-2019

- Bug fix: Resource folder not included in distribution Issue 541

35.55 Version 2.2.0

*Release Date: Dec-19-2019**

- Major Enhancement: Add built-in resources support Issue 209 This also required many changes to the code samples, but they can be run now without downloading separate images.
- Major Enhancement: Auto-calculate hit box points by trimming out the transparency
- Major Enhancement: Sprite sheet support for the tiled map editor works now
- Enhancement: Added `load_spritesheet` for loading images from a sprite sheet
- Enhancement: Updates to physics engine to better handle non-rectangular sprites
- Enhancement: Add SpriteSolidColor class, for creating a single-color rectangular sprite
- Enhancement: Expose type hints to modules that depend on arcade via PEP 561 Issue 533 and Issue 534
- Enhancement: Add font_color to gui.TextButton init Issue 521
- Enhancement: Improve error messages around loading tilemaps
- Bug fix: Turn on vsync as it sometimes was limiting FPS to 30.
- Bug fix: get_tile_by_gid() incorrectly assumes tile GID cannot exceed tileset length Issue 527

- Bug fix: Tiles in object layers not placed properly Issue 536
- Bug fix: Typo when loading font Issue 518
- Updated `requirements.txt` file
- Add robots.txt to documentation

Please also update pyglet, pyglet_ffmpeg2, and pytiled_parser libraries.

Special tanks to Jon Fincher, Mr. Gallo, SirGnip, lubie0kasztanki, and EvgeniyKrysanoc for their contributions to this release.

35.56 Version 2.1.7

- Enhancement: Tile set support. Issue 511
- Bug fix, search file tile images relative to tile map. Issue 480

35.57 Version 2.1.6

- Fix: Lots of fixes around positioning and hitboxes with tile maps Issue 503
- Documentation updates, particularly using *on_update* instead of *update* and *remove_from_sprite_lists* instead of *kill*. Issue 381
- Remove/adjust some examples using csvs for maps

35.58 Version 2.1.5

- Fix: Default font sometimes not pulling on mac Issue 488
- Documentation updates, particularly around examples for animated characters on platformers
- Fix to Sprite class to better support character animation around ladders

35.59 Version 2.1.4

- Fix: Error when importing arcade on Raspberry Pi 4 Issue 485
- Fix: Transparency not working in draw functions Issue 489
- Fix: Order of parameters in draw_ellipse documentation Issue 490
- Raise better error on data classes missing
- Lots of code cleanup from SirGnip Issue 484
- New code for buttons and dialog boxes from wamiqurrehman093 Issue 476

35.60 Version 2.1.3

- Fix: Ellipses drawn to incorrect dimensions Issue 479
- Enhancement: Add unit test for debugging Issue 478
- Enhancement: Add more descriptive error when file not found Issue 472
- Enhancement: Explicitly state delta time is in seconds Issue 473
- Fix: Add missing 'draw' function to view Issue 470

35.61 Version 2.1.2

- Fix: Linked to wrong version of Pyglet Issue 467

35.62 Version 2.1.1

- Added pytiled-parser as a dependency in setup.py

35.63 Version 2.1.0

- New file reader for tmx files http://arcade.academy/arcade.html#module-arcade.tilemap
- Add new view switching framework http://arcade.academy/example_code/how_to_examples/index.html#view-management
- Fix and Re-enable TravisCI builds https://travis-ci.org/pvcraven/arcade/builds
- New: Collision methods to Sprite Issue 434
- Fix: make_circle_texture Issue 431
- Fix: Points drawn as triangles rather than rects Issue 429
- Fix: Fix screen update rate issue Issue 424
- Fix: Typo Issue 422
- Put in exampel Kayzee game
- Fix: Add links to PyCon video Issue 414
- Fix: Docstring Issue 409
- Fix: Typo Issue 403

Thanks to SirGnip, Mr. Gallow, and Christian Clauss for their contributions.

35.64 Version 2.0.9

- Fix: Unable to specify path to .tsx file for tiled spritesheet Issue 360
- Fix: TypeError: __init__() takes from 3 to 11 positional arguments but 12 were given in text.py Issue 373
- Fix: Test create_line_strip Issue 379
- Fix: TypeError: draw_rectangle_filled() got an unexpected keyword argument 'border_width' Issue 385
- Fix: See about creating a localization/internationalization example Issue 391
- Fix: Glitch when you die in the lava in 09_endgame.py Issue 392
- Fix: No default font found on ArchLinux and no error message (includes patch) Issue 402
- Fix: Update docs around batch drawing and array_backed_grid.py example Issue 403

35.65 Version 2.0.8

- Add example code from lixingque
- Fix: Drawing primitives example broke in prior release Issue 365
- Update: Improve automated testing of all code examples Issue 326
- Update: raspberry pi instructions, although it still doesn't work yet
- Fix: Some buffered draw commands not working Issue 368
- Remove yml files for build environments that don't work because of OpenGL
- Update requirement.txt files
- Fix mountain examples
- Better error handling when playing sounds
- Remove a few unused example code files

35.66 Version 2.0.7

- Last release improperly required pyglet-ffmpeg, updated to pyglet-ffmpeg2
- Fix: The alpha value seems NOT work with draw_texture_rectangle Issue 364
- Fix: draw_xywh_rectangle_textured error Issue 363

35.67 Version 2.0.6

- Improve ffmpeg support. Think it works on MacOS and Windows now. Issue 350
- Improve buffered drawing command support
- Improve PEP-8 compliance
- Fix for tiled map reader, Issue 360
- Fix for animated sprites Issue 359

- Remove unused avbin library for mac

35.68 Version 2.0.5

- Issue if scale is set for a sprite that doesn't yet have a texture set. Issue 354
- Fix for `Sprite.set_position` not working. Issue 356

35.69 Version 2.0.4

- Fix for drawing with a border width of 1 Issue 352

35.70 Version 2.0.3

Version 2.0.2 was compiled off the wrong branch, so it had a bunch of untested code. 2.0.3 is what 2.0.2 was supposed to be.

35.71 Version 2.0.2

- Fix for loading a wav file Issue 344
- Fix Linux only getting 30 fps Issue 342
- Fix error on window creation Issue 340
- Fix for graphics cards not supporting multi-sample Issue 339
- Fix for set view error on mac Issue 336
- Changing scale attribute on Sprite now dynamically changes sprite scale Issue 331

35.72 Version 2.0.1

- Turn on multi-sampling so lines could be anti-aliased Issue 325

35.73 Version 2.0.0

Released 2019-03-10

Lots of improvements in 2.0.0. Too many to list, but the two main improvements:

- Using shaders for sprites, making drawing sprites incredibly fast.
- Using ffmpeg for sound.

35.74 Version 1.3.7

Released 2018-10-28

- Fix for Issue 275 where sprites can get blurry.

35.75 Version 1.3.6

Released 2018-10-10

- Bux fix for spatial hashing
- Implement commands for getting a pixel, and image from screen

35.76 Version 1.3.5

Released 08-23-2018

Bug fixes for spatial hashing and sound.

35.77 Version 1.3.4

Released 28-May-2018

35.77.1 New Features

- Issue 197: Add new set of color names that match CSS color names
- Issue 203: Add on_update as alternative to update
- Add ability to read .tmx files.

35.77.2 Bug Fixes

- Issue 159: Fix array backed grid buffer example
- Issue 177: Kind of fix issue with gi sound library
- Issue 180: Fix up API docs with sound
- Issue 198: Add start of isometric tile support
- Issue 210: Fix bug in MacOS sound handling
- Issue 213: Update code with gi streamer
- Issue 214: Fix issue with missing images in animated sprites
- Issue 216: Fix bug with venv
- Issue 222: Fix get_window when using a Window class

35.77.3 Documentation

- Issue 217: Fix typo in doc string
- Issue 198: Add example showing start of isometric tile support

35.78 Version 1.3.3

Released 2018-May-05

35.78.1 New Features

- Issue 184: For sound, wav, mp3, and ogg should work on Linux and Windows. wav and mp3 should work on Mac.

35.78.2 Updated Examples

- Add happy face drawing example

35.79 Version 1.3.2

Released 2018-Apr-20

35.79.1 New Features

- Issue 189: Add spatial hashing for faster collision detection
- Issue 191: Add function to get the distance between two sprites
- Issue 192: Add function to get closest sprite in a list to another sprite
- Issue 193: Improve decorator support

35.79.2 Updated Documentation

- Link the class methods in the quick index to class method documentation
- Add mountain midpoint displacement example
- Improve CSS
- Add “Two Worlds” example game

35.79.3 Updated Examples

- Update `sprite_collect_coints_move_down.py` to not use `all_sprites_list`
- Update `sprite_bullets_aimed.py` to add a warning about how to manage text on a scrolling screen
- Issue 194: Fix for calculating distance traveled in scrolling examples

35.80 Version 1.3.1

Released 2018-Mar-31

35.80.1 New Features

- Update `create_rectangle` code so that it uses color buffers to improve performance
- Issue 185: Add support for repeating textures
- Issue 186: Add support for repeating textures on Sprites
- Issue 184: Improve sound support
- Issue 180: Improve sound support
- Work on improving sound support

35.80.2 Updated Documentation

- Update quick-links on homepage of http://arcade.academy
- Update Sprite class documentation
- Update copyright date to 2018

35.80.3 Updated Examples

- Update PyMunk example code to use keyboard constants rather than hard-coded values
- New sample code showing how to avoid placing coins on walls when randomly placing them
- Improve listing/organization of sample code
- Work at improving sample code, specifically try to avoid using `all_sprites_list`
- Add PyMunk platformer sample code
- Unsuccessful work at getting TravisCI builds to work
- Add new sample for using shape lists
- Create sample code showing difference in speed when using ShapeLists.
- Issue 182: Use explicit imports in sample PyMunk code
- Improve sample code for using a graphic background
- Improve collect coins example
- New sample code for creating caves using cellular automata

- New sample code for creating caves using Binary Space Partitioning
- New sample code for explosions

35.81 Version 1.3.0

Released 2018-February-11.

35.81.1 Enhancements

- Issue 126: Initial support for decorators.
- Issue 167: Improve audio support.
- Issue 169: Code cleanup in SpriteList.move()
- Issue 174: Support for gradients.

35.82 Version 1.2.5

Released 2017-December-29.

35.82.1 Bug Fixes

- Issue 173: JPGs not included in examples

35.82.2 Enhancements

- Issue 171: Clean up sprite list code

35.83 Version 1.2.4

Released 2017-December-23.

35.83.1 Bug Fixes

- Issue 170: Unusually high CPU

35.84 Version 1.2.3

Released 2017-December-20.

35.84.1 Bug Fixes

- Issue 44: Improve wildcard imports
- Issue 150: “Shapes” example refers to chapter that does not exist
- Issue 157: Different levels example documentation hook is messed up.
- Issue 160: sprite_collect_coins example fails to run
- Issue 163: Some examples aren’t loading images

35.84.2 Enhancements

- Issue 84: Allow quick running via -m
- Issue 149: Need better error message with check_for_collision
- Issue 151: Need example showing how to go between rooms
- Issue 152: Standardize name of main class in examples
- Issue 154: Improve GitHub compatibility
- Issue 155: Improve readme documentation
- Issue 156: Clean up root folder
- Issue 162: Add documentation with performance tips
- Issue 164: Create option for a static sprite list where we don’t check to see if things moved.
- Issue 165: Improve error message with physics engine

35.85 Version 1.2.2

Released 2017-December-02.

35.85.1 Bug Fixes

- Issue 143: Error thrown when using scroll wheel
- Issue 128: Fix infinite loop in physics engine
- Issue 127: Fix bug around warning with Python 3.6 when imported
- Issue 125: Fix bug when creating window on Linux

35.85.2 Enhancements

- Issue 147: Fix bug building documentation where two image files where specified incorrectly
- Issue 146: Add release notes to documentation
- Issue 144: Add code to get window and viewport dimensions
- Issue 139: Add documentation on what `collision_radius` is
- Issue 131: Add example code on how to do full-screen games
- Issue 113: Add example code showing enemy turning around when hitting a wall
- Issue 67: Improved support and documentation for joystick/game controllers

CHAPTER

THIRTYSIX

WAYS TO CONTRIBUTE

We would love to have you contribute to the project! There are several ways that you can do so.

36.1 How to contribute without coding

- **Community** - Post your projects, code, screen-shots, and discuss the Arcade library on the Python Arcade Sub-Reddit.
- Try coding your own animations and games. Write down notes on anything that is difficult to implement or understand about the library.
- **Suggest improvements** - Post bugs and enhancement requests at the Github Issue List.

36.2 How to contribute code

First, take some time to understand the project layout:

- *Directory Structure*
- *One-time build*
- *How to Submit Changes*

Then, perform the following steps:

1. Make sure you have Python 3.9+ installed rather than 3.8+ to meet dev tool requirements.
2. Make a virtual environment.
3. Run *pip install -e .[dev]* to perform a dev install.

Afterwards, you can improve these parts of the project:

- **Document** - Edit the reStructuredText and docstrings to make the Arcade documentation better.
- **Test** - Improve the unit testing.
- **Code** - Contribute bug fixes and enhancements to the code.

CHAPTER

THIRTYSEVEN

CONTRIBUTING TO ARCADE

Arcade welcomes contributions, including:

- Bug reports & feature suggestions
- Bug fixes
- Implementations of requested features
- Corrections & additions to the documentation
- Improvements to the tests

If you're looking for a way to contribute, try checking the currently active issues for one that needs work.

37.1 Before Making Changes

Before working on an improvement, please make sure to open an issue if one does not already exist for it.

Tips:

1. Try to keep individual PRs to reasonable sizes
2. If you want to make large changes, please discuss them with Arcade's developers beforehand

Discussion can happen in a GitHub issue's comments or on Arcade's Discord server.

37.2 After Making Changes

After you finish your changes, you should do the following:

1. Test your changes according to the *Testing* section below
2. Submit a pull request from your fork to Arcade's development branch.

The rest of the guide will help you get to this point & explain how to test in more detail.

37.3 Requirements

Although using arcade only requires Python 3.8 or higher, development currently requires 3.9 or higher.

The rest of this guide assumes you've already done the following:

1. Installed Python 3.9+ with pip
2. Installed git
3. Forked the repo on GitHub
4. Cloned your fork locally
5. Changed directories into your local Arcade clone's folder

Creating & using a virtual environment is also strongly recommended.

37.4 Installing Arcade in Editable Mode

To install all necessary development dependencies, run this command in your terminal from inside the top level of the arcade directory:

```
# For Unix-like shells (Linux, macOS)
pip install -e '.[dev]'
```

```
# For Windows
pip install -e .[dev]
```

If you get an error like the one below, you probably need to update your pip version:

```
ERROR: File "setup.py" not found. Directory cannot be installed in editable mode: /home/
↪user/Projects/arcade
(A "pyproject.toml" file was found, but editable mode currently requires a setup.py␣
↪based build.)
```

Upgrade by running the following command:

```
pip install --upgrade pip
```

Mac & Linux users can improve their development experience further by following the optional steps at the end of this document.

37.5 Testing

You should test your changes locally before submitting a pull request to make sure they work correctly & don't break anything.

Ideally, you should also write unit tests for new features. See the tests folder in this repo for current tests.

37.5.1 Testing Code Changes

First, run the below command to run our linting tools automatically. This will run Mypy and Ruff against Arcade. The first run of this may take some as MyPy will not have any caches built up. Sub-sequent runs will be much faster.

```
python make.py lint
```

If you want to run either of these tools individually, you can do

```
python make.py ruff
```

or

```
python make.py mypy
```

Now you run the framework's unit tests with the following command:

```
python make.py test
```

37.5.2 Building & Testing Documentation

Automatic Rebuild with Live Reload

You can build & preview documentation locally using the following steps.

Run the doc build to build the web page files, and host a webserver to preview:

```
python make.py serve
```

You can now open http://localhost:8000 in your browser to preview the docs.

The `doc/build/html` directory will contain the generated website files. When you change source files, it will automatically regenerate, and browser tabs will automatically refresh to show your updates.

If you suspect the automatic rebuilds are failing to detect changes, you can run a simpler one-time build using the following instructions.

One-time build

Run the doc build to build the web page files:

```
python make.py html
```

The `doc/build/html` directory will contain the generated website files.

Start a local web server to preview the doc:

```
python -m http.server -d doc/build/html
```

You can now open http://localhost:8000 in your browser to preview the doc.

Be sure to re-run build & refresh to update after making changes!

37.6 Optional: Improve Ergonomics on Mac and Linux

37.6.1 make.py shorthand

On Mac & Linux, you can run the make script as `./make.py` instead of `python make.py`.

For example, this command:

```
python make.py lint
```

can now be run this way:

```
./make.py lint
```

37.6.2 Enable Shell Completions

On Mac & Linux, you can enable tab completion for commands on the following supported shells:

- `bash` (the most common default shell)
- `zsh`
- `fish`
- `powershell`
- `powersh`

For example, if you have typed the following…

```
./make.py h
```

Tab completion would allow you to press tab to auto-complete the command:

```
./make.py html
```

Note that this may interfere if you work on other projects that also have a *make.py* file.

To enable this feature, most users can follow these steps:

1. Run `./make.py whichshell` to find out what your default shell is
2. If it is one of the supported shells, run `./make.py --install-completion $(basename "$SHELL")`
3. Restart your terminal

If your default shell is not the shell you prefer using for arcade development, you may need to specify it to the command above directly instead of using auto-detection.

CHAPTER

THIRTYEIGHT

DIRECTORY STRUCTURE

Directory	Description
\arcade	Source code for the arcade library. including various sub-modules
\arcadeexamples	Example code showing how to use Arcade.
\arcadeexperimental	Experimental features and more advanced examples
\tests	Unit tests. Most unit tests are part of the docstrings.
\doc	Arcade documentation. Note that API documentation is in docstrings along with the source.
\doc\tutorials	Tutorial pages and code
\doc\images	Images used in the documentation.
\doc\build\html	After making the documentation, all the HTML code goes here. Look at this in a web browser to see what the documentation will look like.
\build	All built code from the compile script goes here.
\dist	Distributable Python wheels go here after the build script has run.

Also see *One-time build*.

CHAPTER

THIRTYNINE

HOW TO SUBMIT CHANGES

First, you should open up an issue or enhancement request on the Github Issue List.

Next, create your own fork of the Arcade library. The Arcade library is at:

https://github.com/pythonarcade/arcade

Follow the *One-time build* on how to build the code.

You can submit changes with a "pull request." With a pull request you ask that another repository (in this case the Arcade library) "pull" your changes into the main code base.

If you aren't familiar with how to do pull requests, the Stack Overflow discussion on pull requests is good.

CHAPTER

FORTY

RELEASE CHECKLIST

1. Check for updated libraries, and if we need to pin a more recent version.
2. Run `ruff arcade`
3. Run `mypy arcade`
4. In docs folder, type `make clean` then `make html` and confirm no warnings/errors.
5. Run unit tests in `tests` folder.
6. Run `tests/test_examples/run_all_examples.py`
7. Make sure `arcade/examples/asteroid_smasher.py` is playable.
8. Make sure `arcade/examples/platform_tutorial/17_views.py` is playable.
9. Update version number in `arcade/version.py`
10. Update *Release Notes* with release dates and any additional info needed.
11. Make sure last check-in ran clean on github actions, viewable on Discord
12. Merge development branch into maintenance.
13. Add label to release
14. Push code. Check for clean compile on github.
15. Type `make clean`
16. Type `make dist`
17. Type `make deploy_pypi`
18. Confirm release notes appear on website.
19. Announce on Arcade Discord, Python Discord, Reddit Python Arcade, etc.

CHAPTER FORTYONE

SOCIAL

- Discord (most active spot)
- Reddit /r/pythonarcade
- Twitter @ArcadeLibrary
- Instagram @PythonArcadeLibrary
- Facebook @ArcadeLibrary
- diversity_statement

CHAPTER FORTYTWO

LEARNING RESOURCES

- Book - Learn to program with Arcade
- Peer To Peer Gaming With Arcade and Python Banyan
- US PyCon 2022 Talk
- US PyCon 2019 Tutorial
- Aus PyCon 2018 Multiplayer Games
- US PyCon 2018 Talk

INDEX