Python Arcade Documentation

Release 2.6.17

Paul Vincent Craven

Aug 05, 2023

GETTING STARTED

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

ONE

GET STARTED HERE

1.1 Installation

Arcade can be installed like any other Python Package. Arcade needs support for OpenGL 3.3+. It does not run on Raspberry Pi or Wayland. If you are familiar with Python package management you can just "pip install" Arcade. For more detailed instructions see *Installation Instructions*.

1.2 Starting Tutorials

If you are already familiar with basic Python programming, follow the *Simple Platformer* or Real Python article. If you are just learning how to program, see the Learn Arcade book.

1.3 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_joystick
     * 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 Tutorial*.
- 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 - Scene Object*, 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 - Multiple Levels and Other Layers*
   - 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 - Display The Score*
    * Layers - Platformer tutorial: *Step 8 - Display The Score*
    * 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 Notes*.
  - Lights - light_demo
  - Writing shaders using "ShaderToy"
    * *Shader Toy Tutorial - Glow*

    * *Shader Toy Tutorial - 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
  - Bloom/glow effect - bloom_defender
  - Learn to do a compute shader in *Compute Shader Tutorial*.
- *Logging*

CHAPTER

TWO

INSTALLATION INSTRUCTIONS

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

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

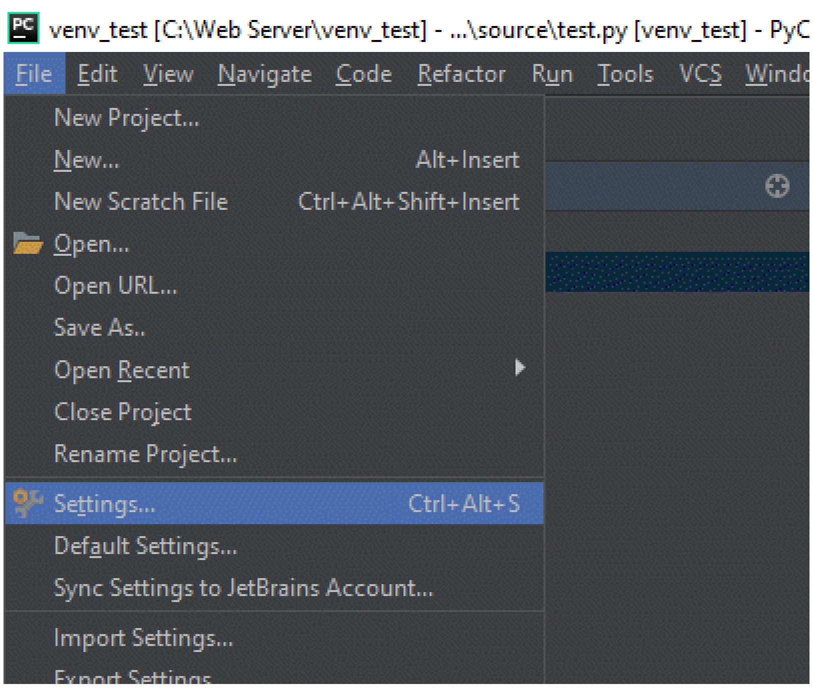
Select the instructions for your platform:

2.1 Setting Up a Virtual Environment In PyCharm

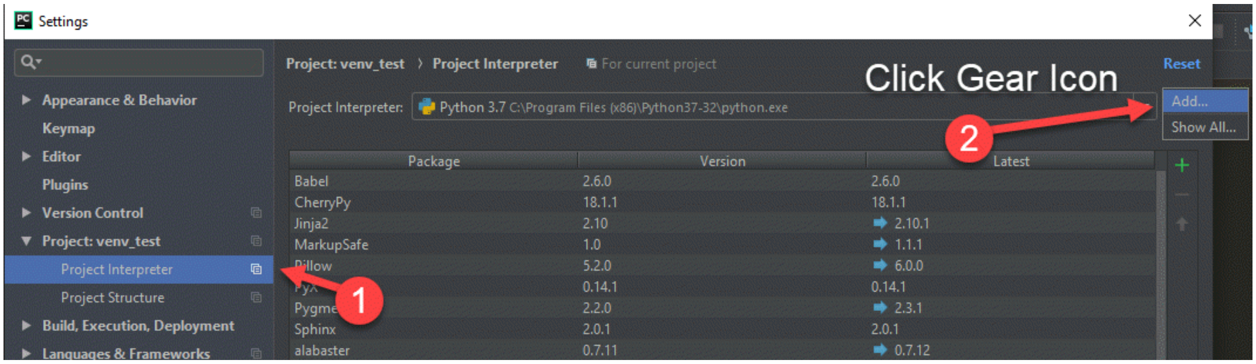
A Python virtual environment (venv) allows libraries to be installed for just a single project, rather than shared across everyone using the computer. It also does not require administrator privileges to install.

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

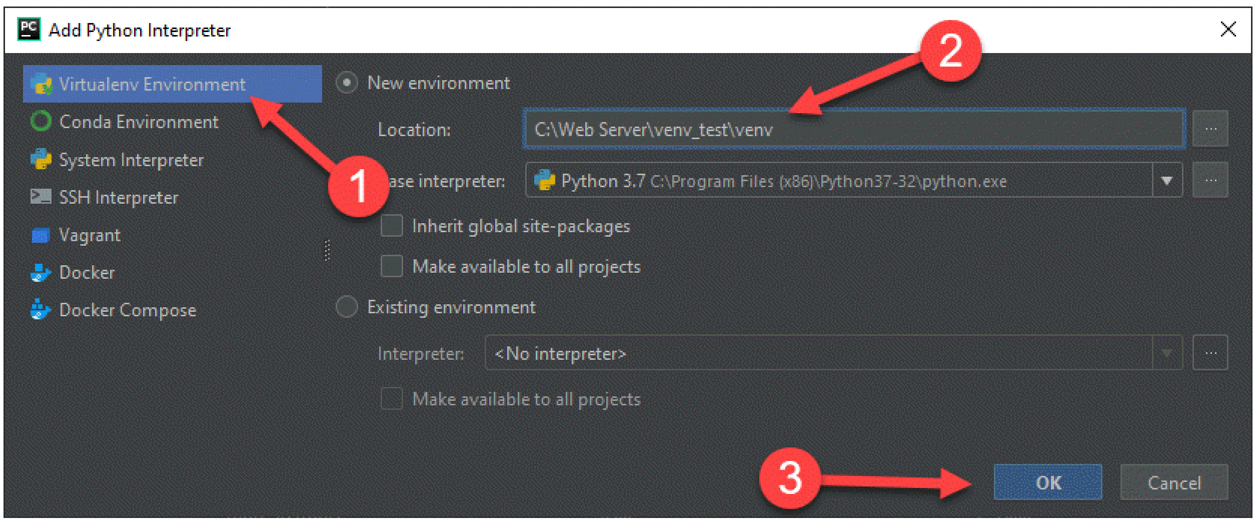
Step 1: Select File. . . Settings



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

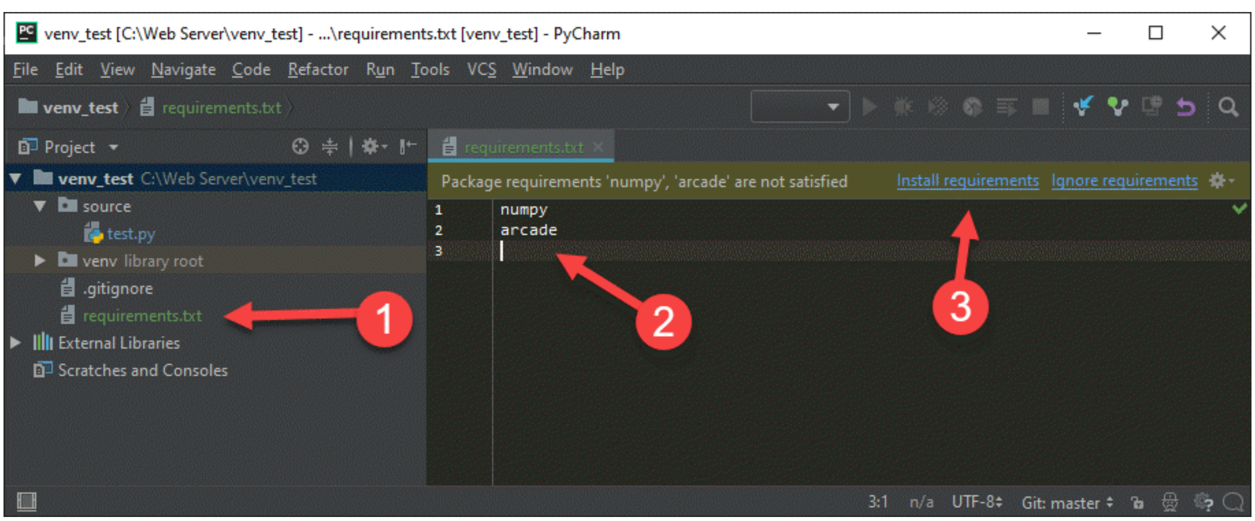


Step 3: Select Virtualenv Environment from the left. Then create a new environment. Usually it should be in a folder called `venv` in your main project. PyCharm does not always select the correct location by default, so carefully look at the path to make sure it is correct, then select "Ok".



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.



2.2 Installation on Windows

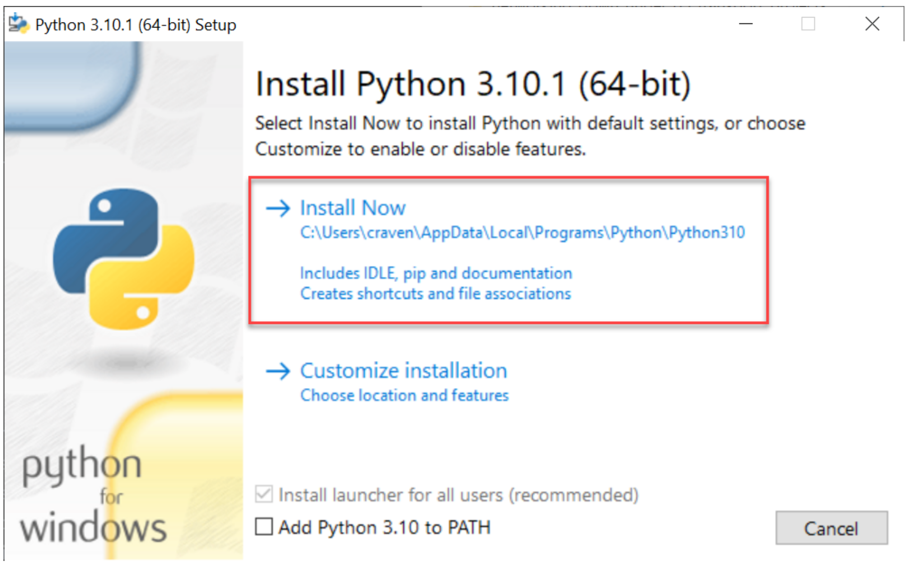
To develop with the Arcade library, we need to install Python, then install Arcade.

2.2.1 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.



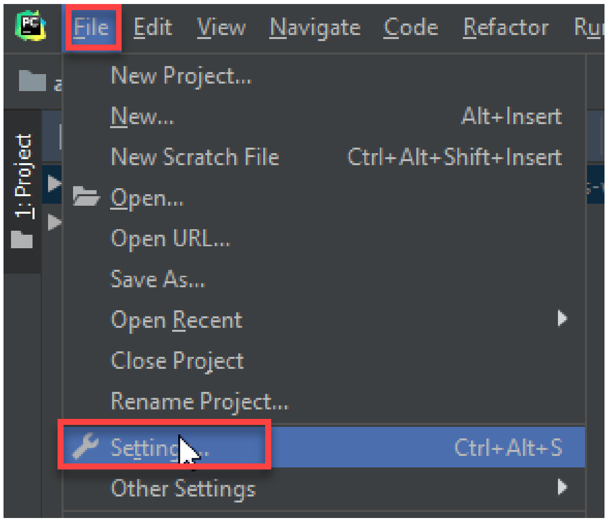
Once installed, you can just close the dialog. There's no need to increase the path length, although it doesn't hurt anything if you do.

2.2.2 Step 2: Install The Arcade Library

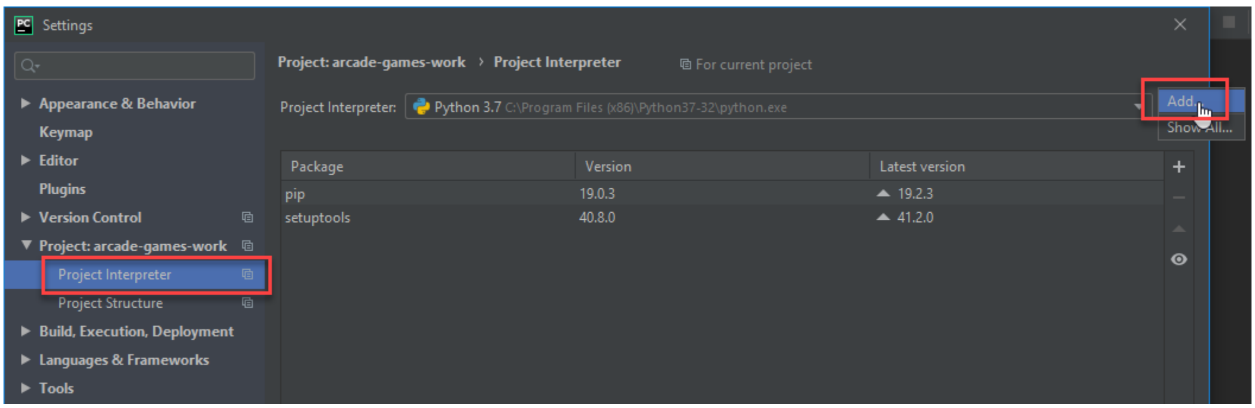
If you install Arcade as a pre-built library, there are two options on how to do it. The best way is to use a "virtual environment." This is a collection of Python libraries that only apply to your particular project. You don't have to worry about libraries for other projects conflicting with your project.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

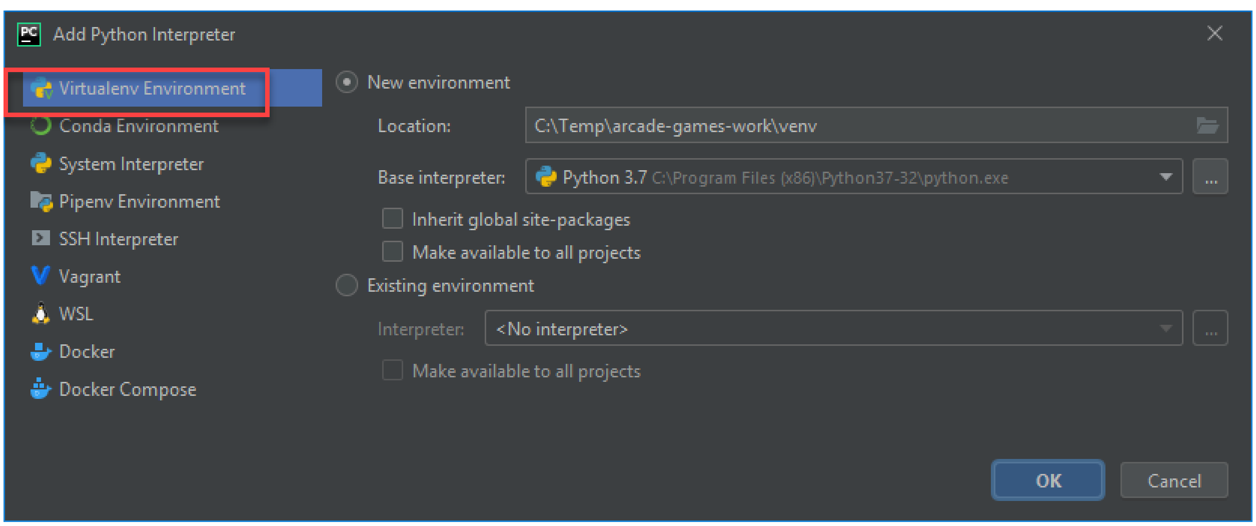
If you are using PyCharm, (the community edition works great and is free) setting up a virtual environment is easy. Once you've created your project, open up the settings:



Select project interpreter:

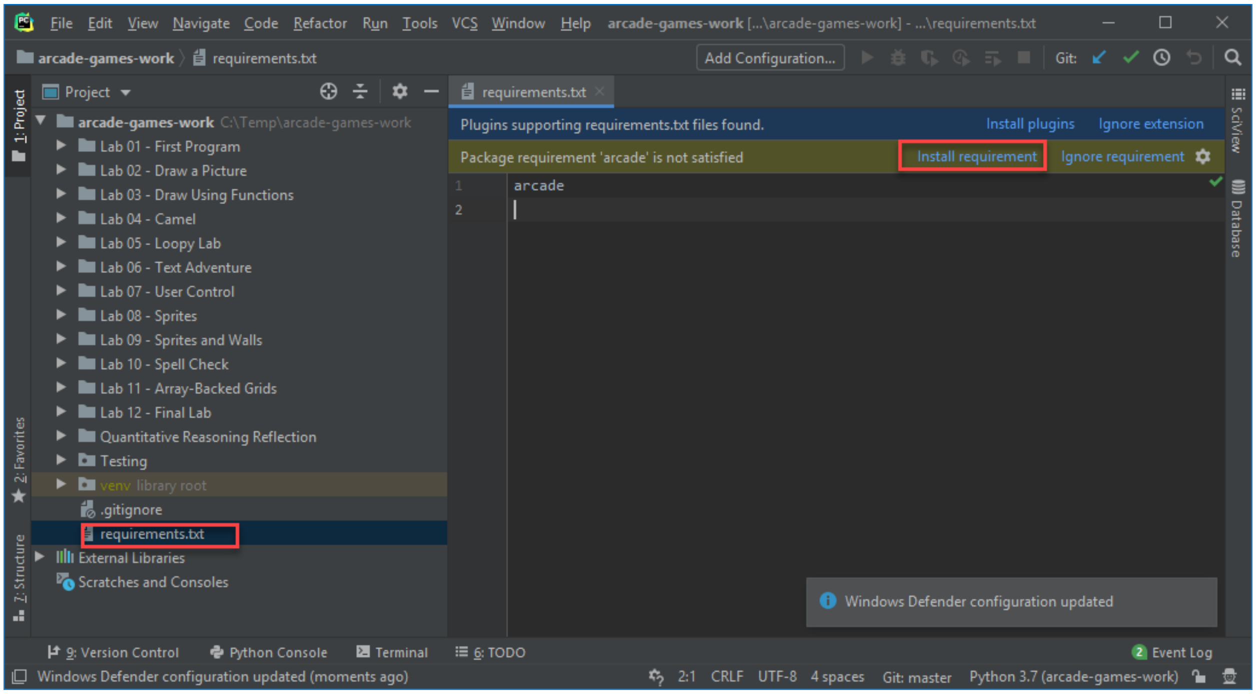


Create a new virtual environment. Make sure the venv is inside your project folder.



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.



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:

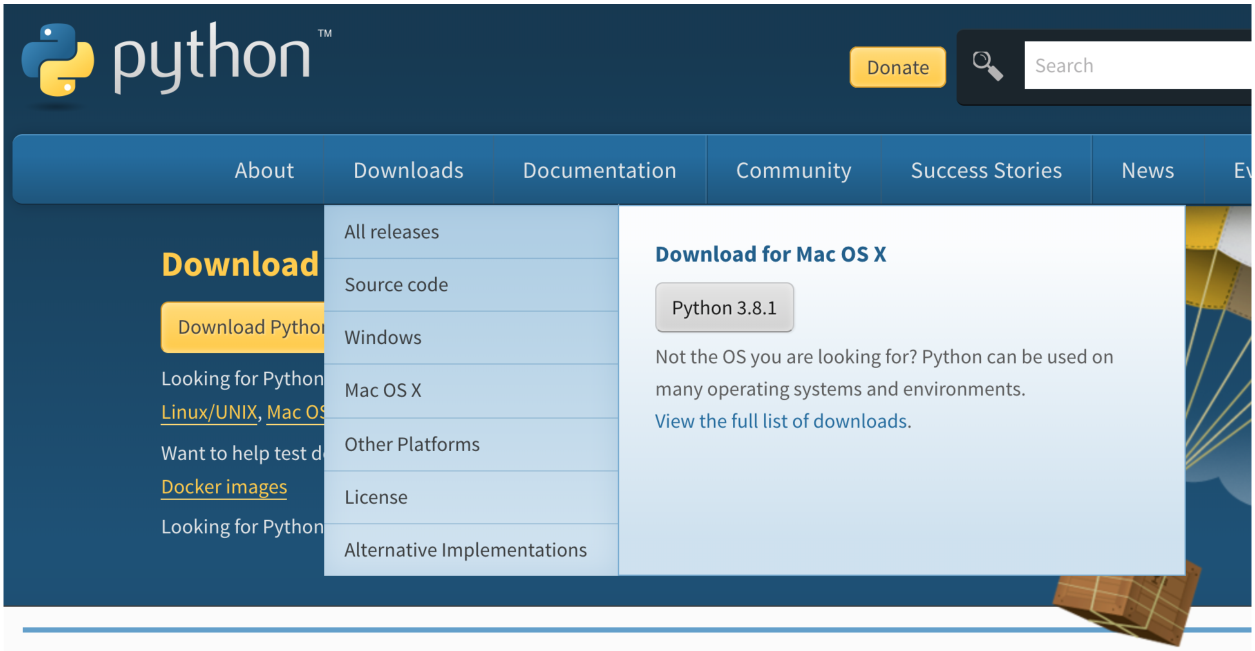
```
pip3 install arcade
```

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

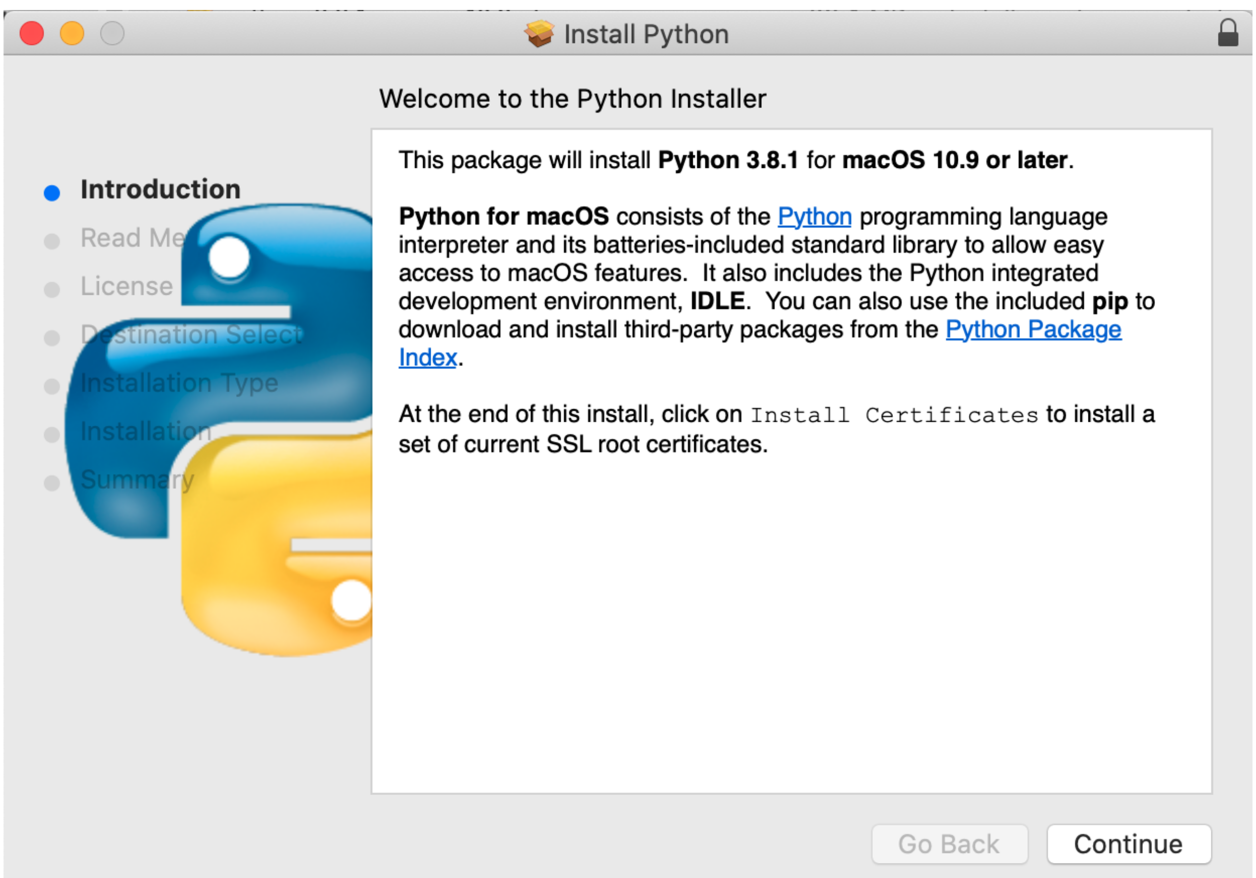
```
python3 -m pipenv install arcade
```

2.3 Installation on the Mac

Go to the Python website and download Python.



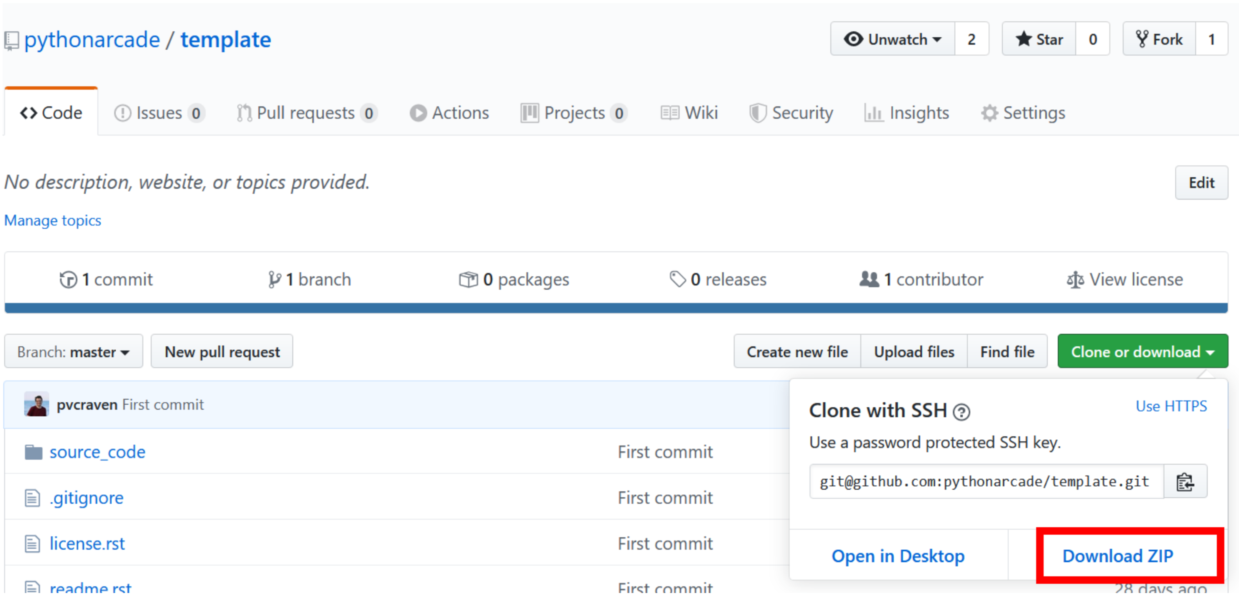
Then install it:



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.

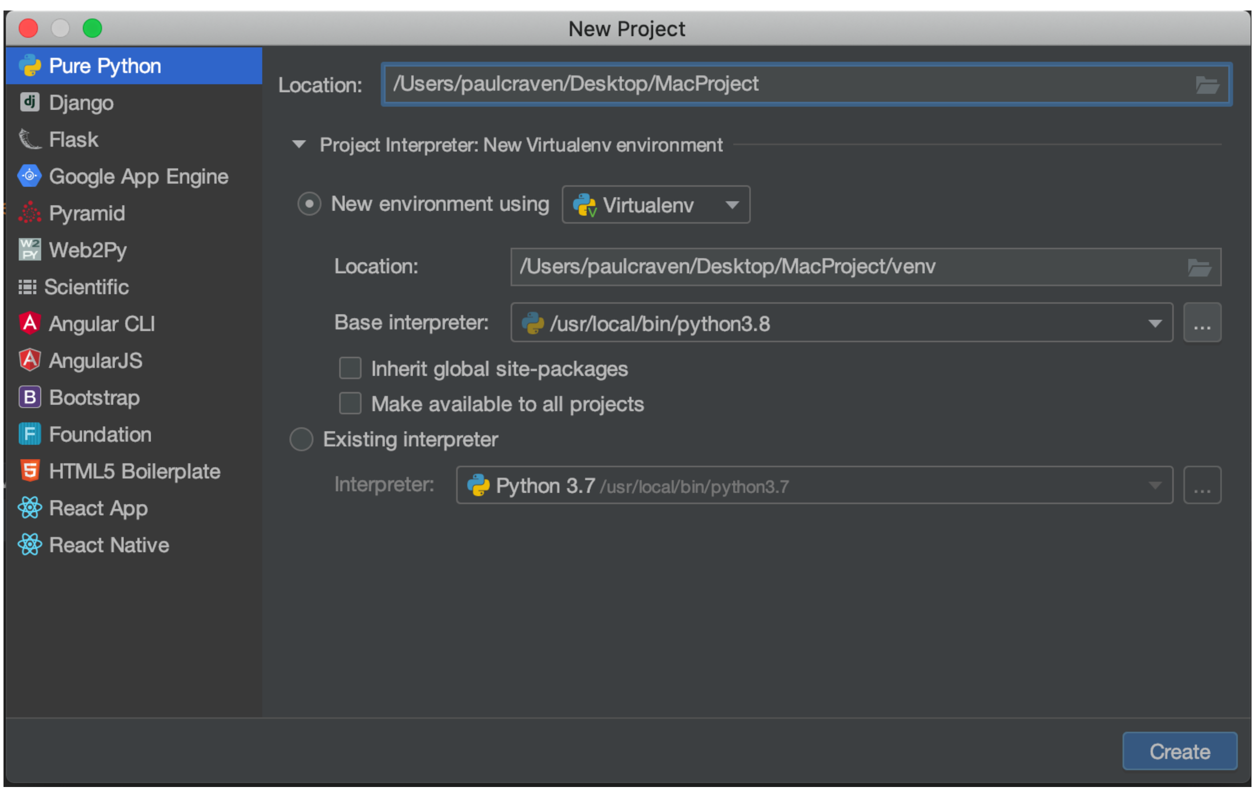
https://github.com/pythonarcade/template



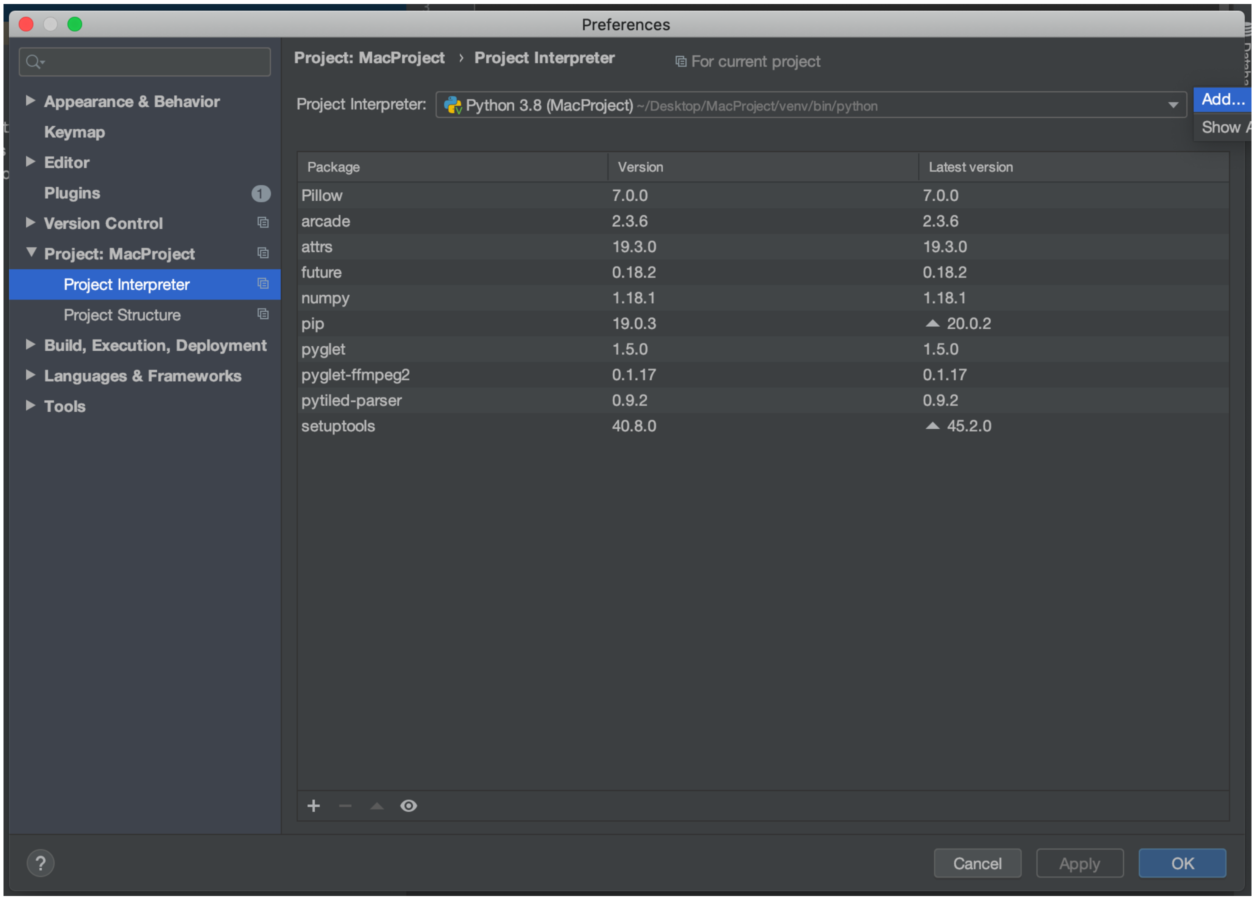
After you've downloaded it, open up the zip file, and pull out the template folder to your desktop or wherever you'd like to save it. Then rename it to your project name.

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

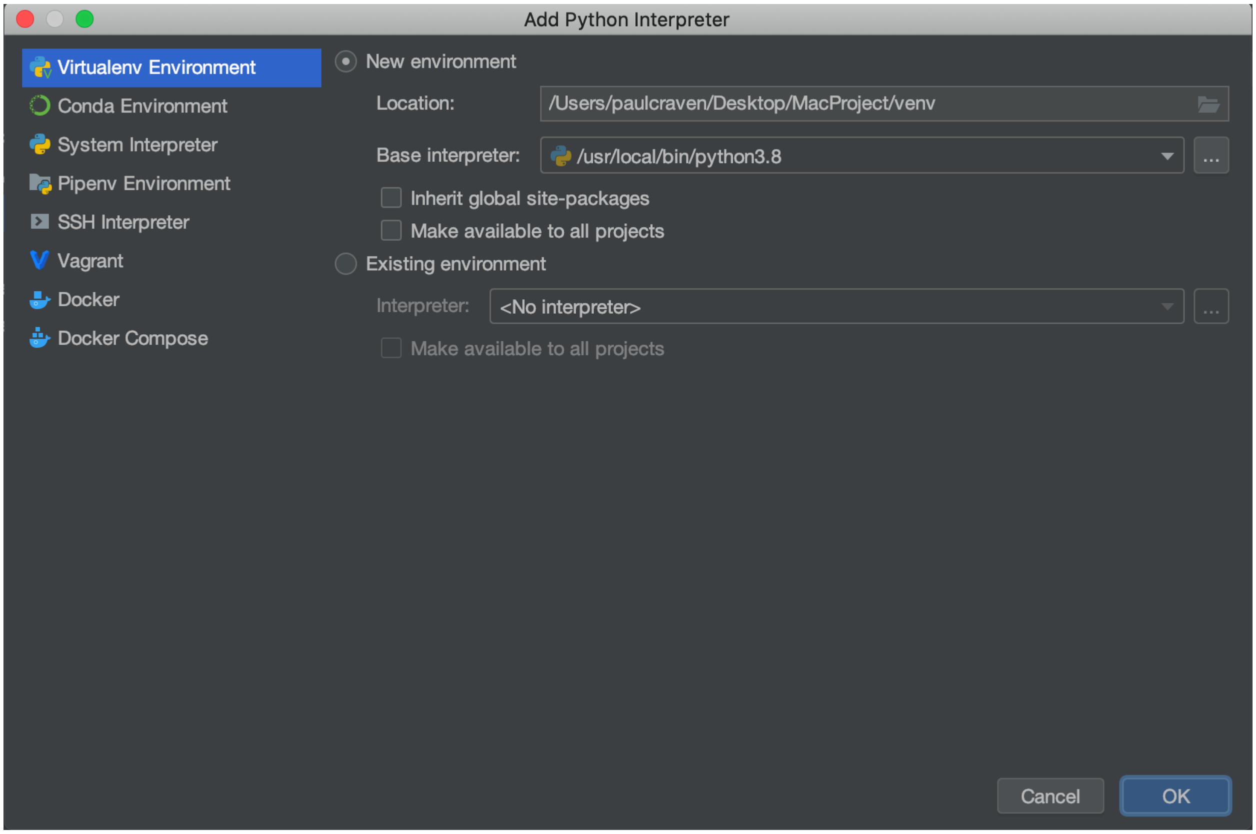
When creating opening the new project, create a virtual environment like so:



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



…Then set it like so:



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.

2.4 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
```

2.4.1 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.

2.5 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/master.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]
```

2.6 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 these if at all possible.

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

THREE

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.

Here are some comparisons between Arcade 2.6 and Pygame 2.0.1:

Table 1: Library Information

Feature	Arcade	Pygame
Website	https://arcade.academy	https://www.pygame.org
API Docs	*API Docs*	API Docs
Example code	*Example Code*	N/A
License	MIT License	LGPL
Back-end graphics engine	OpenGL 3.3+ and Pyglet	SDL 2
Back-end audio engine	ffmpeg via Pyglet	SDL 2
Example Projects	*Games Made With Arcade*	Games Made With Pygame

Table 2: Feature Comparison

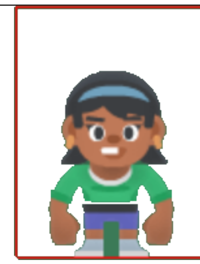
Feature	Arcade	Pygame
Drawing primitives support rotation	Yes	No[1]
Sprites support rotation	Yes	No[Page 20, 1]
Sprites support scaling	Yes	No[Page 20, 1]
Sprite image caching[2]	Yes	No
Type Hints	Yes	No
Transparency support	Yes	Must specify transparent colorkey
Camera support	Yes	No
Android support	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	No (or add 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

Table 3: Performance Comparison[4]

Feature	Arcade	Pygame
Draw 50,000 stationary sprites	0.001 seconds	0.425 seconds
Move 5,000 sprites	0.010 seconds	0.003 seconds
# sprites program can move + draw before FPS drops below 55	8500	2000
Collision detection 50,000 sprites	0.044 seconds no spatial hashing[5] 0.005 seconds with spatial hashing	0.004 seconds[6]
Draw 5,000 plain rectangles[7]	0.081 seconds	0.008 seconds
Draw 5,000 rotated rectangles[8]	0.081 seconds	0.029 seconds

[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.

[2] When creating a sprite from an image, Pygame will load the image from the disk every time. The user must cache the image with their own code for better performance. Arcade does this automatically.

[3] A programmer can achieve a similar result by drawing to a surface, then drawing the surface to the screen.

[4] Performance tests done on an Intel Core i7-9700F with GeForce GTX 980 Ti. Source code for tests available at https://github.com/pythonarcade/

Fig. 1: FPS comparison of programs drawing **stationary** sprites.

Fig. 2: FPS comparison of programs drawing **moving** sprites.

performance_tests and more detailed results at https://craven-performance-testing.s3-us-west-2.amazonaws.com/index.html

[5] Polygon hit box, rotation allowed

[6] Rectangular hit box, no rotation allowed

[7] Why is Arcade so slow here? With PyGame, most of the drawing is done on the **CPU** side. Bitmaps are created and manipulated by the CPU. It is pretty fast. With Arcade, most of the drawing happens on the **GPU** side. Sprites and drawings are batched together, and we just tell the GPU what we want to change. Or better yet, we write a "shader" program that runs completely on the GPU. This is *incredibly* fast. But if instead a CPU program runs commands to draw individual GPU items one-by-one, both sets of processors wait for a synchronous communication. That is horribly slow. Drawing individual rects and bits like PyGame does, won't work well at all on Arcade. Use sprites, shaders, or batch-drawing to get fast performance.

[8] Scaling and rotation must be done by the programmer drawing to a surface, transforming the surface, then blit'ing the surface to the screen. Arcade uses the GPU for these operations and needs no additional code or performance hits.

CHAPTER

FOUR

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:

- 2020_game_jam
- *Concept Games*
- *Simple Platformer*

4.1 Temporum

Temporum, by DragonMoffon

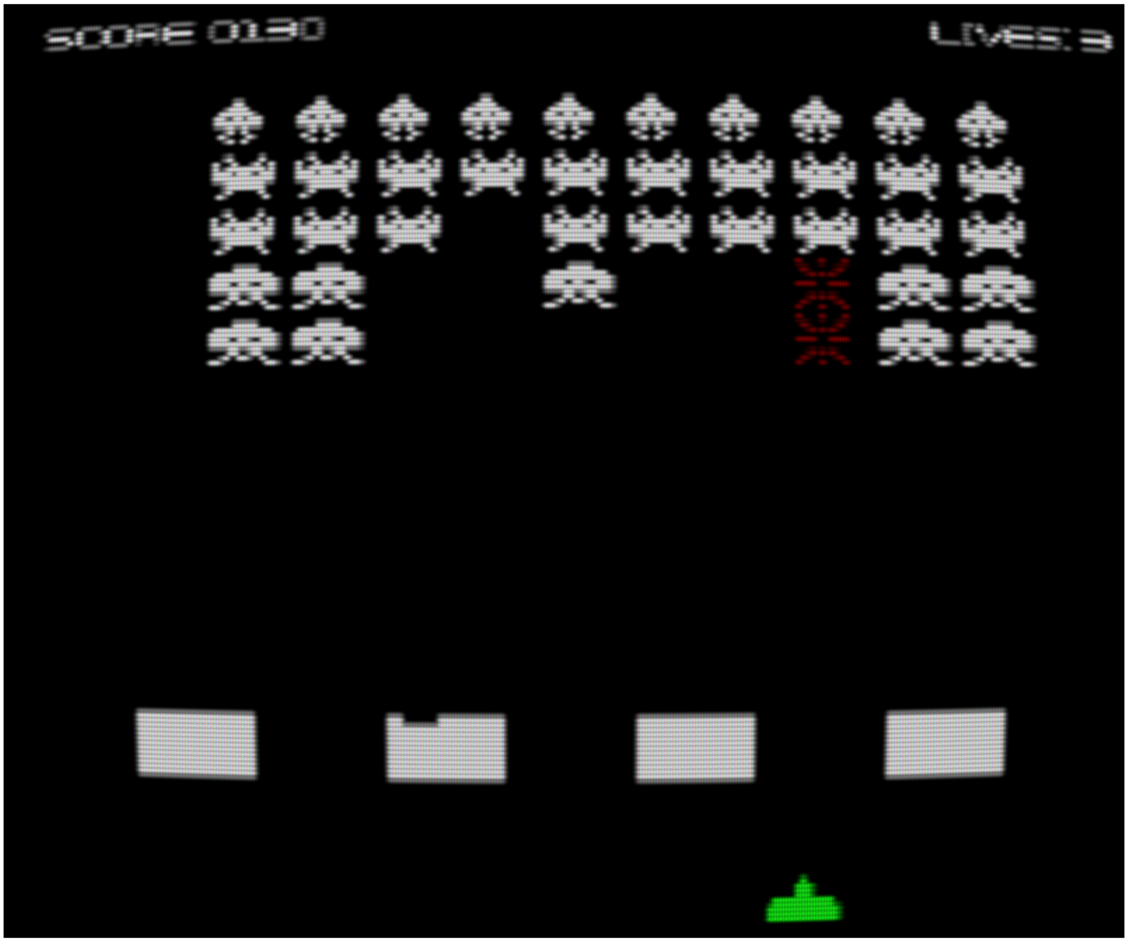
4.2 SOL Defender

SOL Defender, by DragonMoffon

4.3 Binary Defense

Binary Defense by KommentatorForAll

4.4 Space Invaders



Space Invaders

4.5 Ready or Not?

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

4.6 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.

4.7 Fishy-Game

Fishy Game by LiorAvrahami

4.8 Adventure

Adventure GitHub

4.9 Transcience Animation

Transcience Animation

4.10 Stellar Arena Demo

Stellar Arena Demo

4.11 Battle Bros

Battle Bros Mortal Kombat style game.

4.12 Rabbit Herder

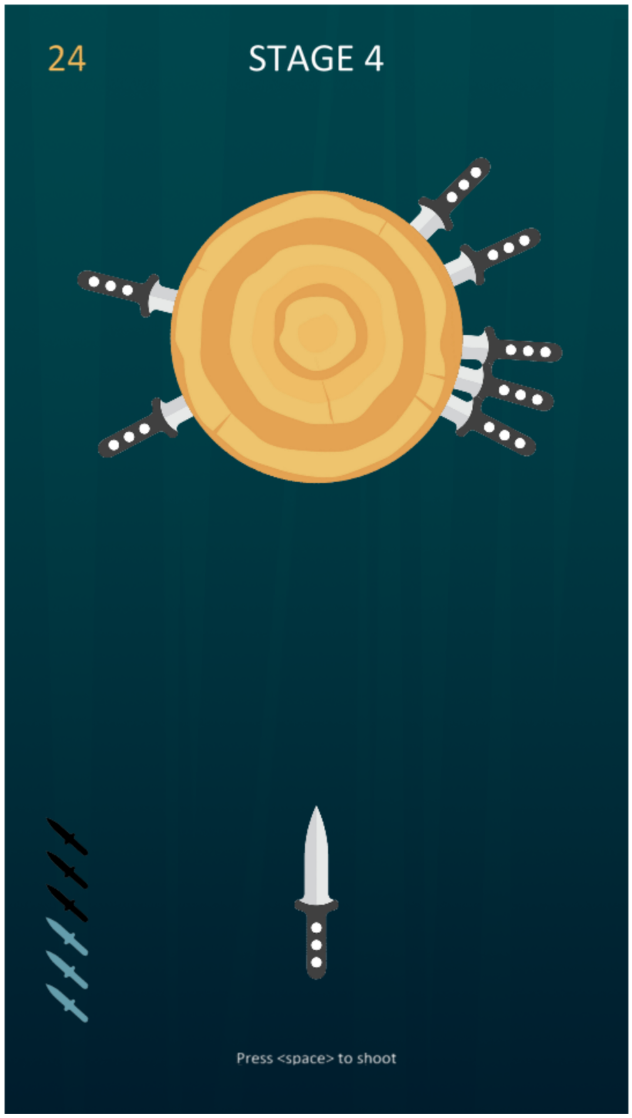
Rabbit Herder, use carrots and potions to herd a rabbit through a maze.

4.13 The Great Skeleton War

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

4.14 Python Knife Hit

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


24
STAGE 4
Press <space> to shoot

4.15 Kayzee

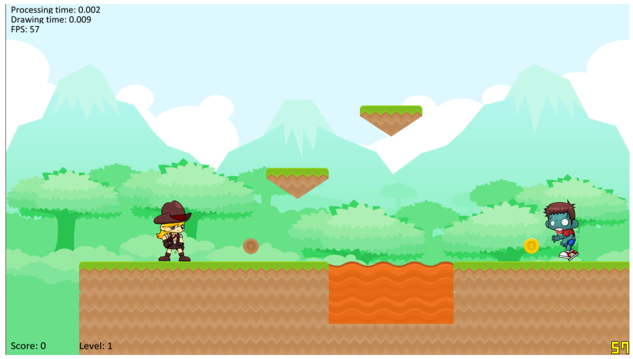


Fig. 1: Kayzee Game

4.16 lixingqiu Games

Fig. 2: An Eight planet simulation

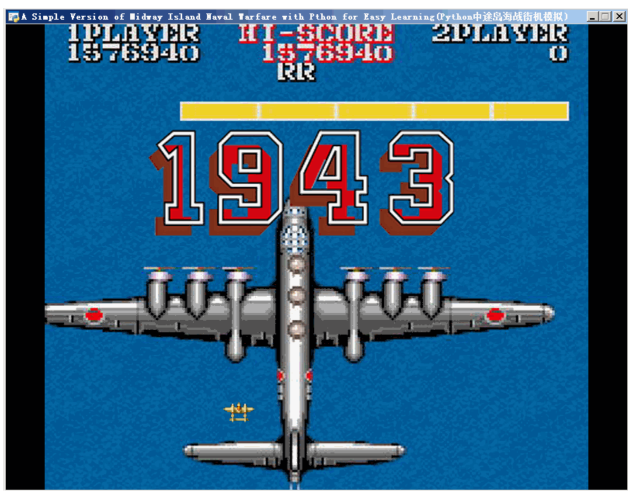
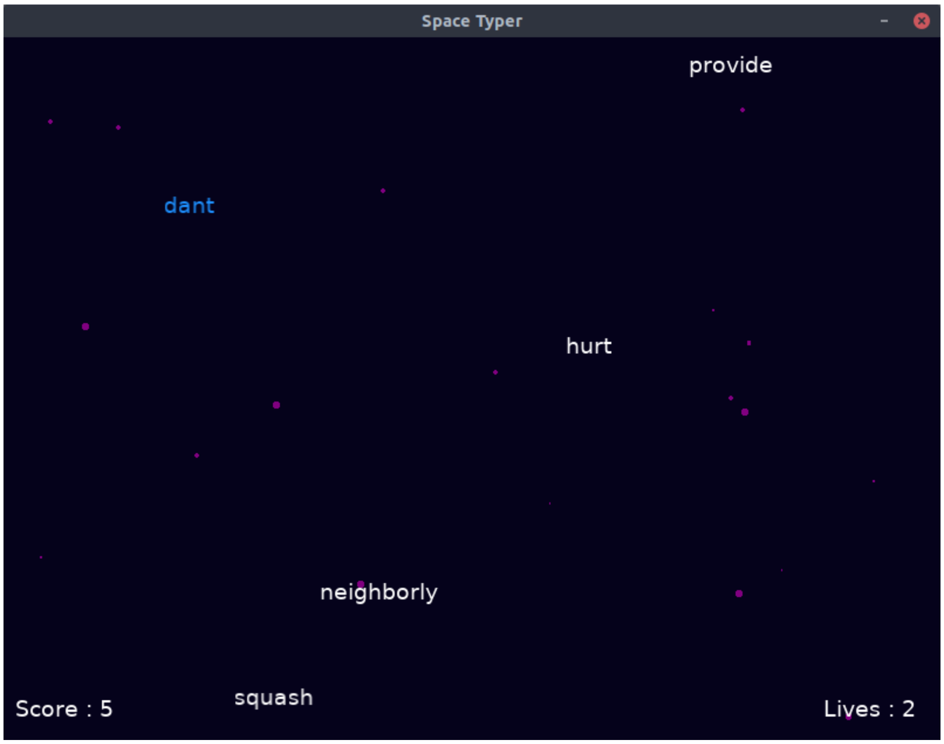


Fig. 3: Midway Island War

Fig. 4: Angry Bird

Fig. 5: Octopus

4.17 Space Typer

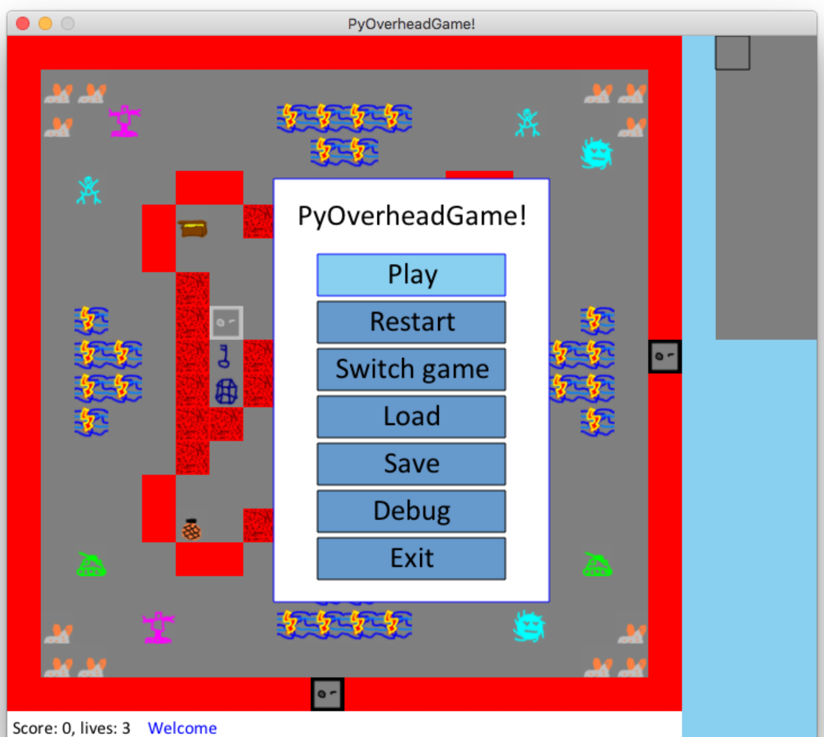


Space Typer - A typing game

4.18 FlapPy Bird

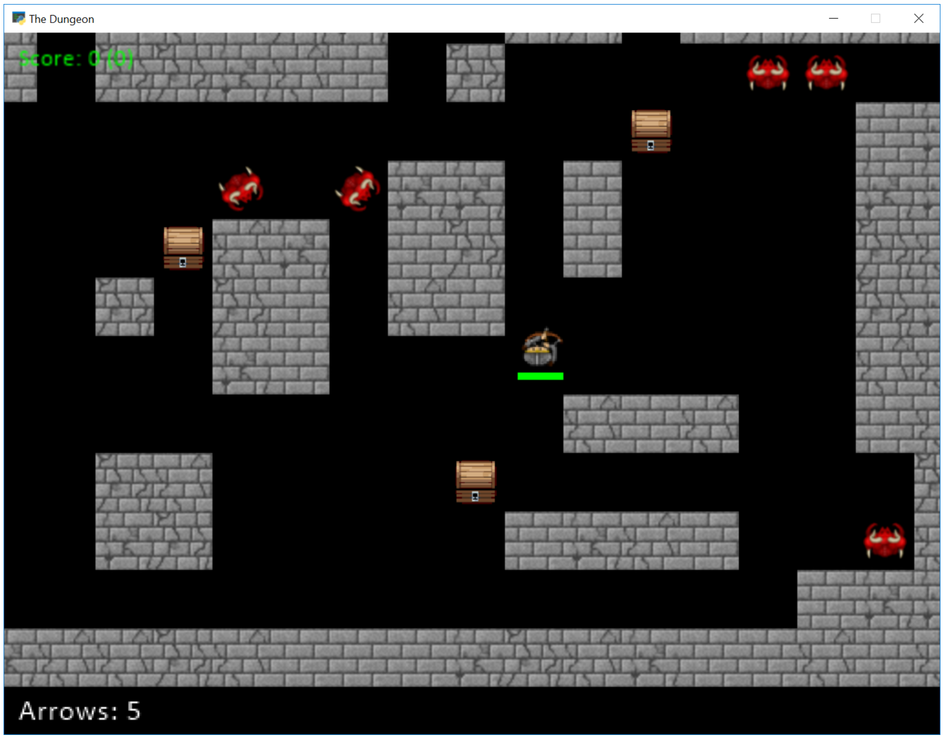
FlapPy-Bird - A bird-game clone.

4.19 PyOverheadGame



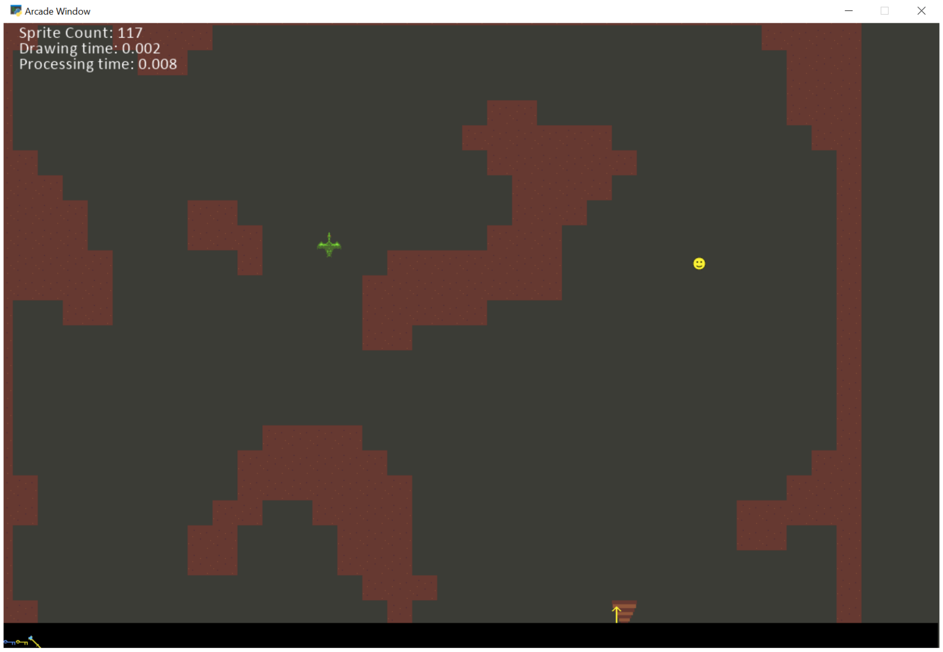
PyOverheadGame, a 2D overhead game where you go through several rooms and pick up keys and other objects.

4.20 Dungeon



Dungeon, explore a maze picking up arrows and coins.

4.21 Two Worlds



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

4.21.1 Simpson College Spring 2017 CMSC 150 Course

These games were created by first-semester programming students.

CHAPTER

FIVE

HOW-TO EXAMPLE CODE

5.1 Starting Templates

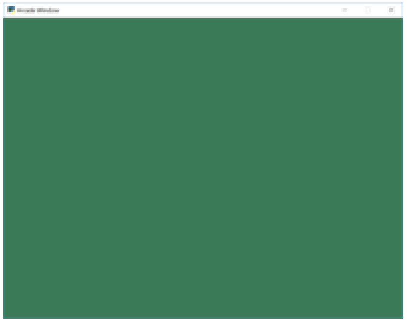


Fig. 1: starting_template

Fig. 2: template_platformer

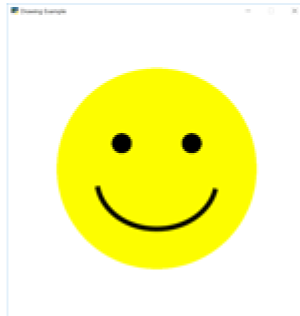


Fig. 3: happy_face

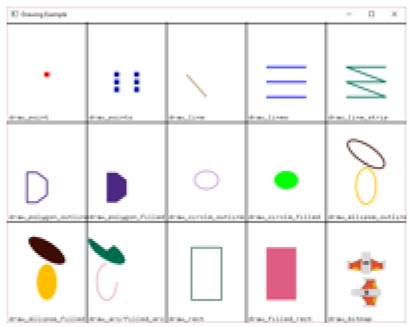


Fig. 4: drawing_primitives

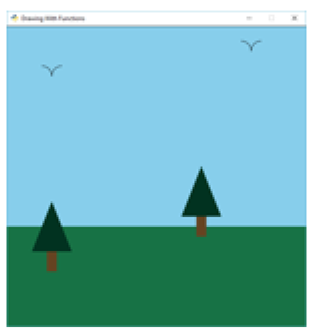


Fig. 5: drawing_with_functions

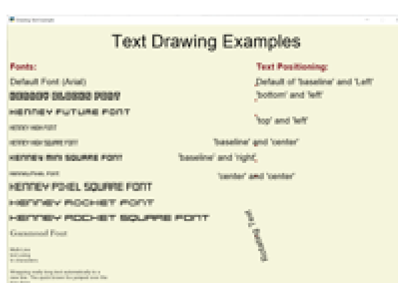


Fig. 6: drawing_text

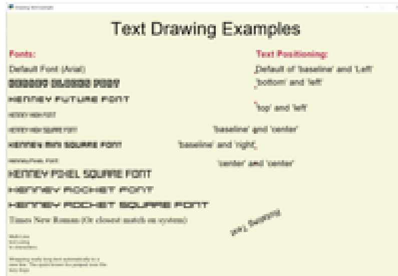


Fig. 7: drawing_text_objects

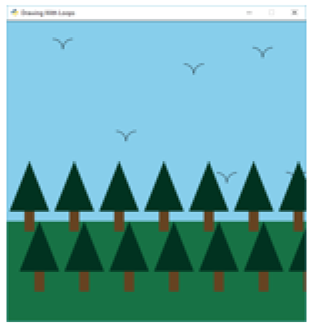


Fig. 8: drawing_with_loops

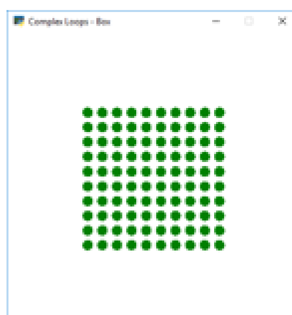


Fig. 9: nested_loops_box

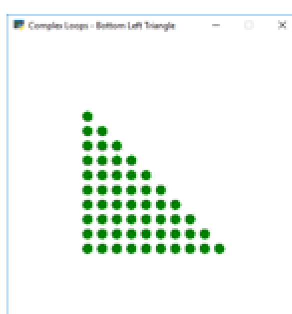


Fig. 10: nested_loops_bottom_left_triangle

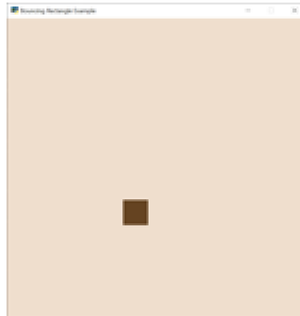


Fig. 11: bouncing_rectangle

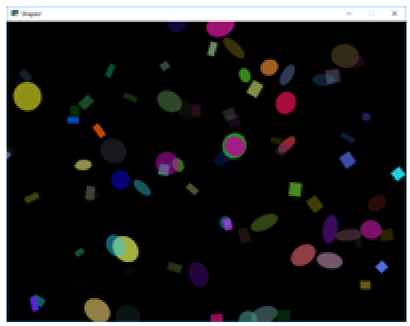


Fig. 12: shapes-slow

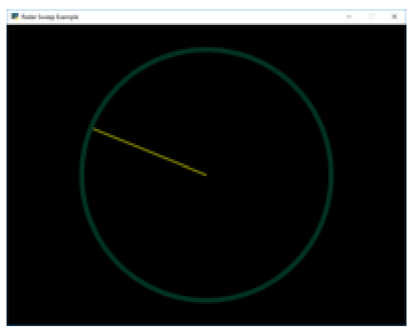


Fig. 13: radar_sweep

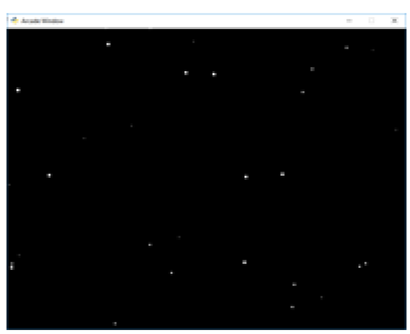


Fig. 14: snow

Fig. 15: shape_list_demo

Fig. 16: lines_buffered

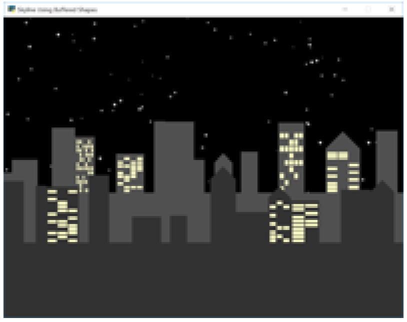


Fig. 17: shape_list_demo_skylines

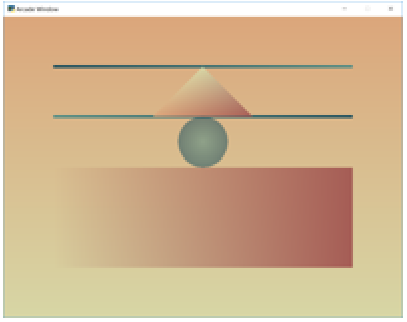


Fig. 18: gradients

5.2 Drawing

5.2.1 Drawing Primitives

5.2.2 Drawing with Loops

5.2.3 Animating Drawing Primitives

5.2.4 Faster Drawing with ShapeElementLists

5.3 Sprites

5.3.1 Sprite Player Movement

Fig. 19: sprite_collect_coins

Fig. 20: sprite_move_keyboard

Fig. 21: sprite_move_keyboard_better

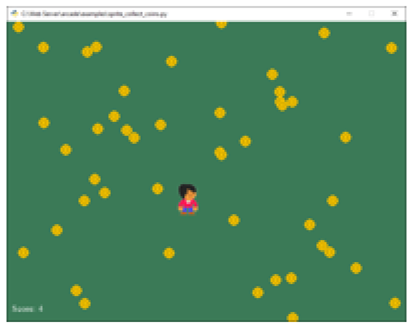


Fig. 22: sprite_move_keyboard_accel

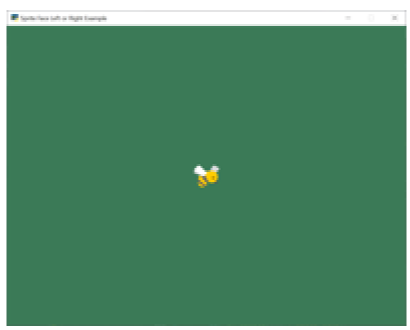


Fig. 23: sprite_face_left_or_right

Fig. 24: sprite_move_joystick

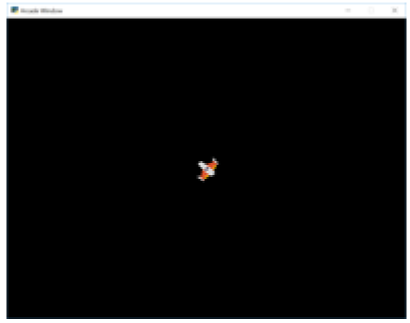


Fig. 25: sprite_move_angle

Fig. 26: dual_stick_shooter

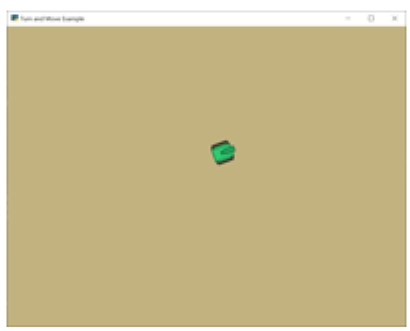


Fig. 27: turn_and_move

Fig. 28: easing_example_1

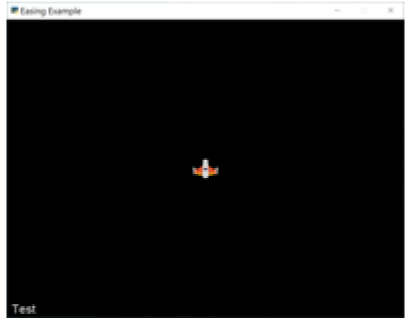


Fig. 29: easing_example_2

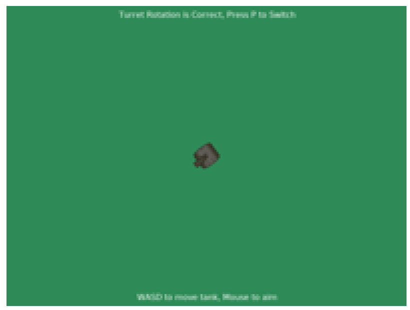


Fig. 30: sprite_rotate_around_tank

5.3.2 Sprite Non-Player Movement

Fig. 31: sprite_collect_coins_move_down

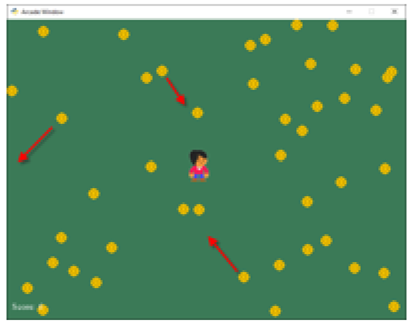


Fig. 32: sprite_collect_coins_move_bouncing

Fig. 33: sprite_bouncing_coins

Fig. 34: sprite_collect_coins_move_circle

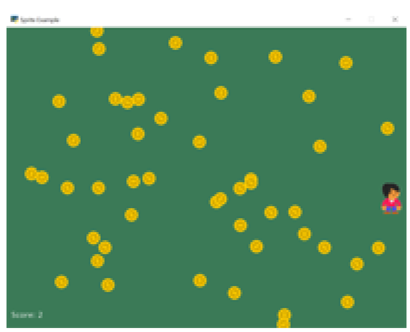


Fig. 35: sprite_collect_rotating

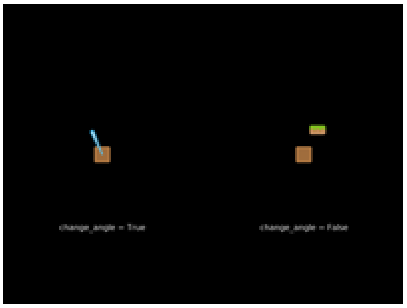


Fig. 36: sprite_rotate_around_point

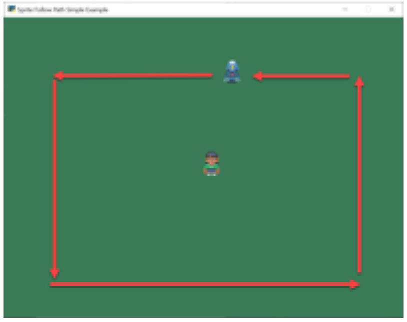


Fig. 37: follow_path

Fig. 38: sprite_follow_simple

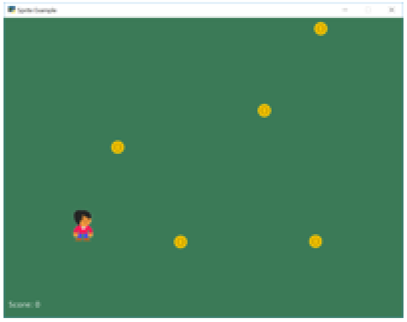


Fig. 39: sprite_follow_simple_2

Fig. 40: line_of_sight

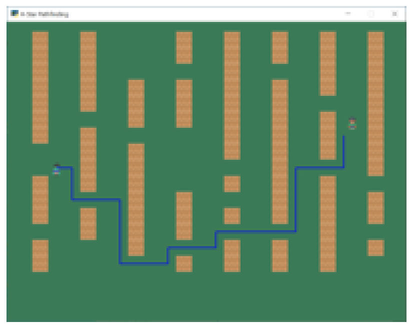


Fig. 41: astar_pathfinding

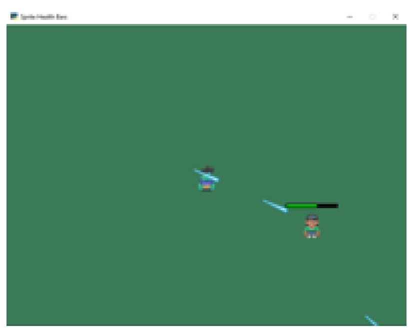


Fig. 42: sprite_health

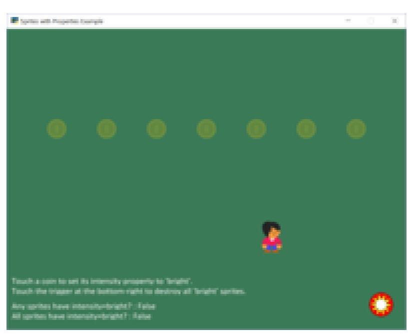


Fig. 43: sprite_properties

Fig. 44: sprite_change_coins

Fig. 45: example-sprite-collect-coins-diff-levels

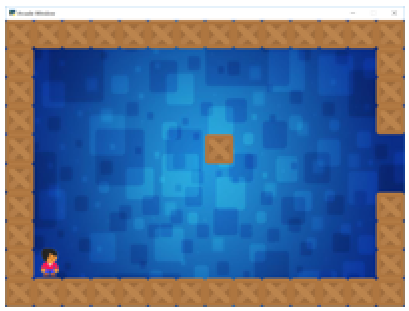


Fig. 46: sprite_rooms

Fig. 47: sprite_bullets

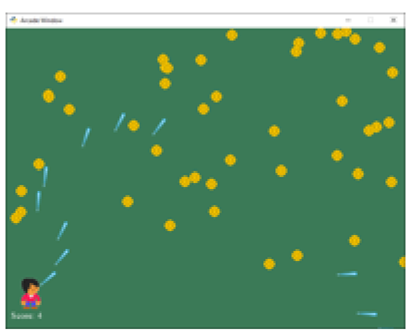


Fig. 48: sprite_bullets_aimed

Fig. 49: sprite_bullets_periodic

Fig. 50: sprite_bullets_random

Fig. 51: sprite_bullets_enemy_aims

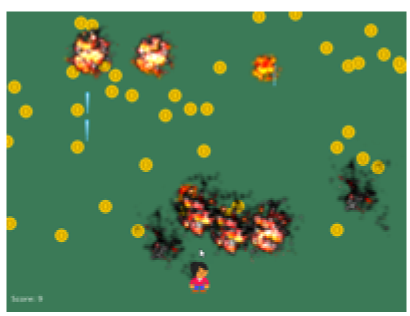


Fig. 52: sprite_explosion_bitmapped

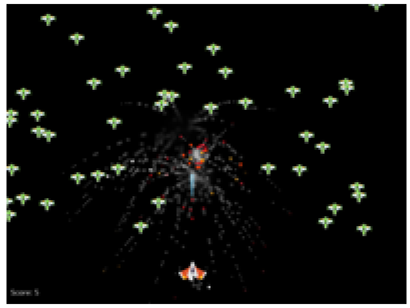


Fig. 53: sprite_explosion_particles

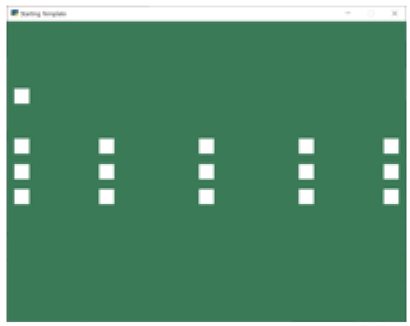


Fig. 54: sound_demo

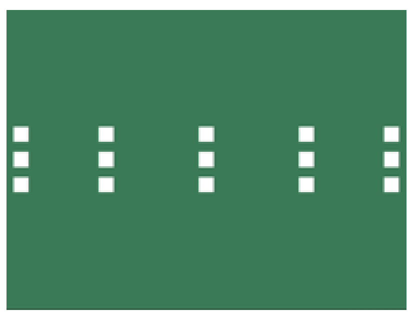


Fig. 55: sound_speed_demo

Fig. 56: music_control_demo

5.3.3 Sprite Pathing

5.3.4 Sprite Properties

5.3.5 Games with Levels

5.3.6 Shooting with Sprites

5.4 Sound

5.5 Camera Use

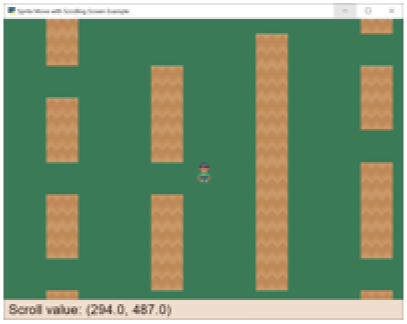


Fig. 57: sprite_move_scrolling

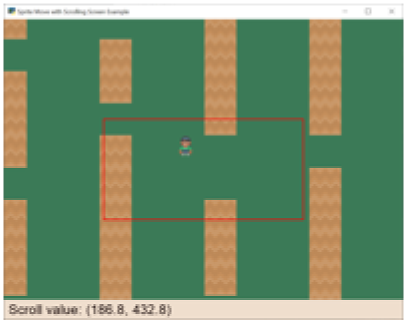


Fig. 58: sprite_move_scrolling_box

Fig. 59: sprite_move_scrolling_shake

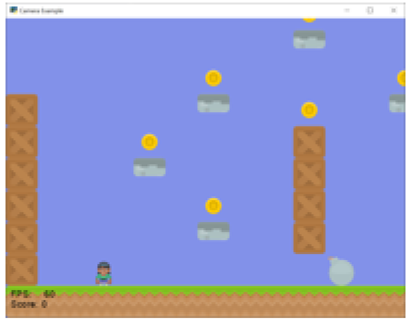


Fig. 60: camera_platform

5.6 Platformers

5.6.1 Basic Platformers

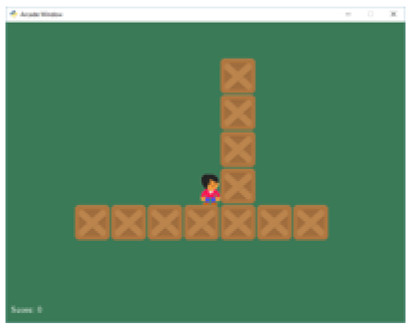


Fig. 61: sprite_move_walls

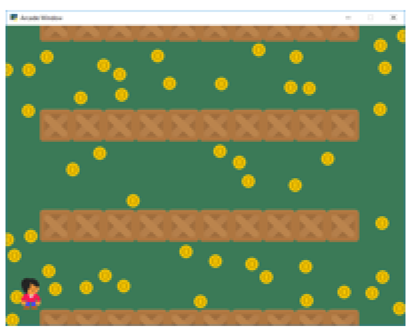


Fig. 62: sprite_no_coins_on_walls

Fig. 63: sprite_move_animation

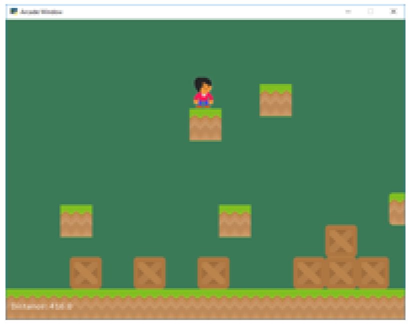


Fig. 64: sprite_moving_platforms

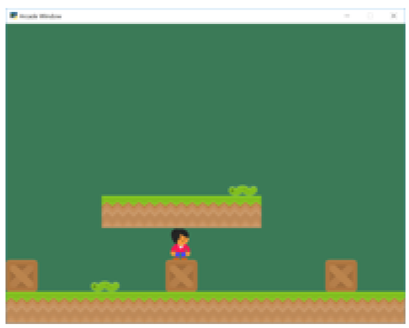


Fig. 65: sprite_enemies_in_platformer

Fig. 66: *Simple Platformer*

5.6.2 Using Tiled Map Editor to Create Maps

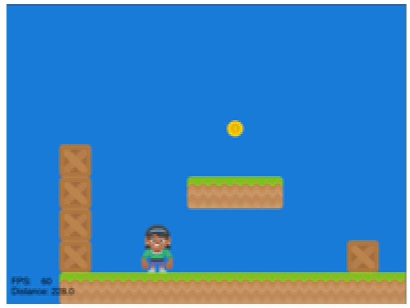


Fig. 67: sprite_tiled_map

Fig. 68: sprite_tiled_map_with_levels

5.6.3 Procedural Generation

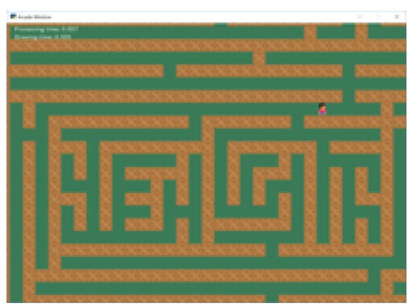


Fig. 69: maze_recursive

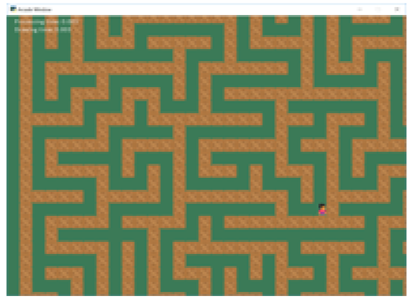


Fig. 70: maze_depth_first

Fig. 71: procedural_caves_cellular

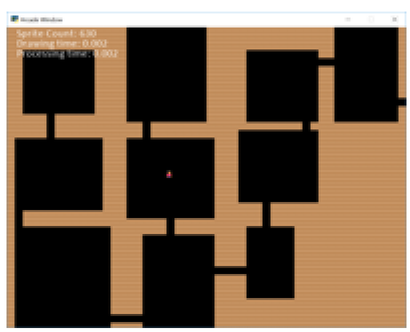


Fig. 72: procedural_caves_bsp

5.7 View Management

5.7.1 Instruction Screens and Game Over Screens

Fig. 73: view_screens_minimal

Fig. 74: view_instructions_and_game_over

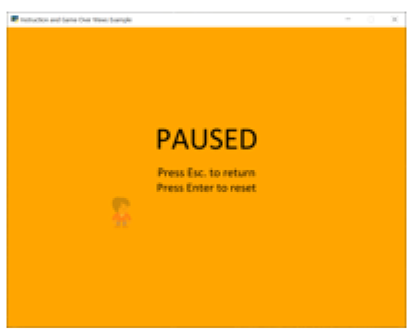


Fig. 75: view_pause_screen

Fig. 76: transitions

5.7.2 Resizable Window and Fullscreen Games

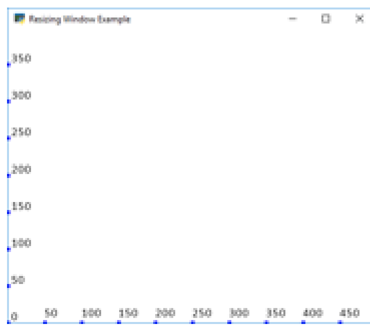


Fig. 77: resizable_window

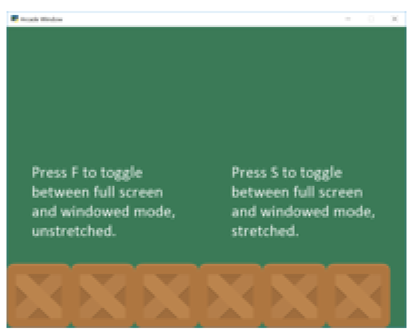


Fig. 78: full_screen_example

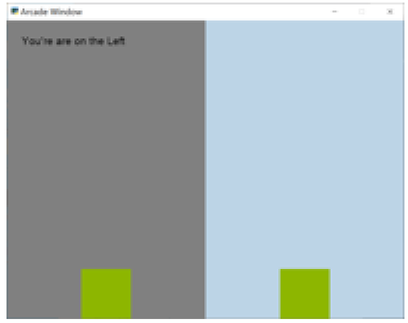


Fig. 79: sections_demo_1

Fig. 80: sections_demo_2

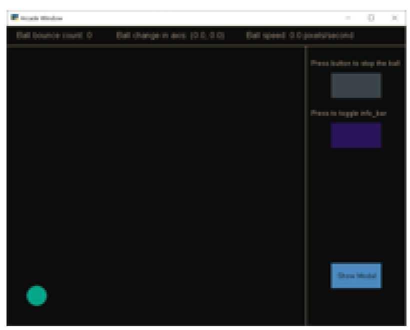


Fig. 81: sections_demo_3

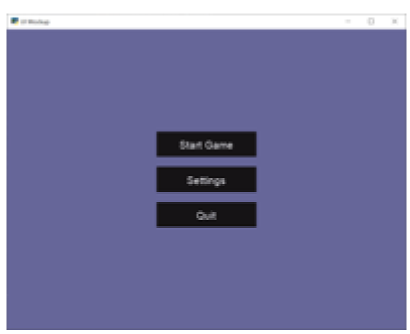


Fig. 82: gui_flat_button

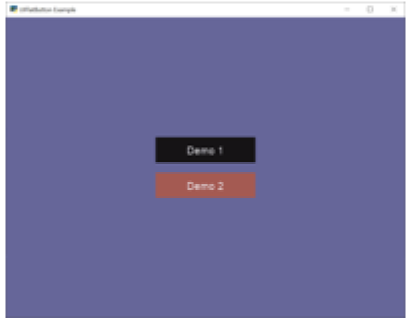


Fig. 83: gui_flat_button_styled

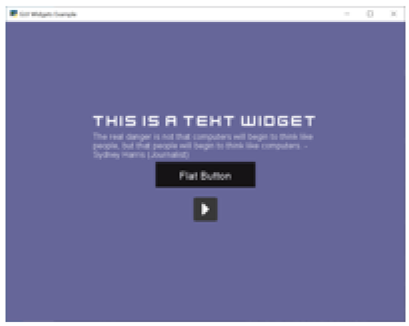


Fig. 84: gui_widgets

Fig. 85: gui_ok_messagebox

Fig. 86: gui_scrollable_text

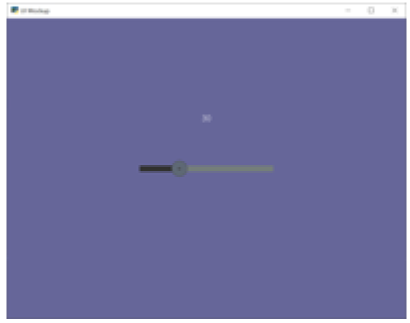


Fig. 87: gui_slider

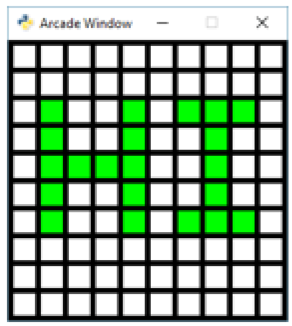


Fig. 88: array_backed_grid

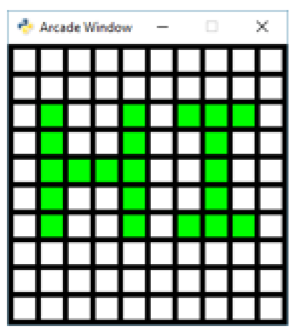


Fig. 89: array_backed_grid_buffered

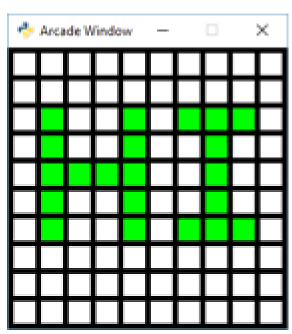


Fig. 90: array_backed_grid_sprites_1

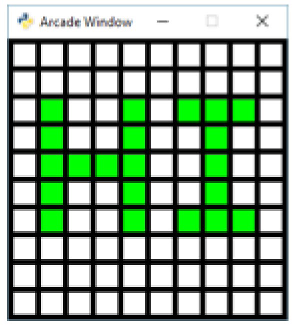


Fig. 91: array_backed_grid_sprites_2

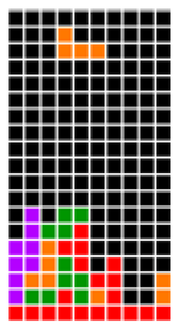


Fig. 92: tetris

Fig. 93: conway_alpha

5.7.3 Dividing a View Into Sections

5.8 Graphical User Interface

5.9 Grid-Based Games

5.10 Advanced

5.10.1 Using PyMunk for Physics

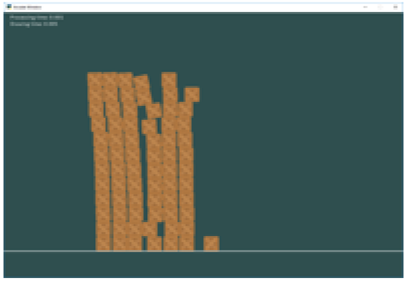


Fig. 94: pymunk_box_stacks

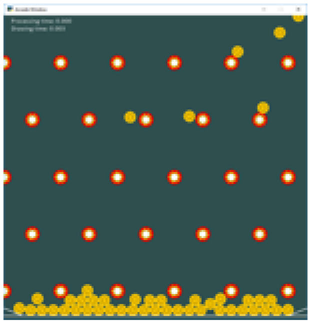


Fig. 95: pymunk_pegboard

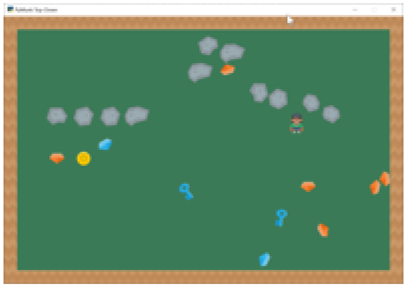


Fig. 96: pymunk_demo_top_down

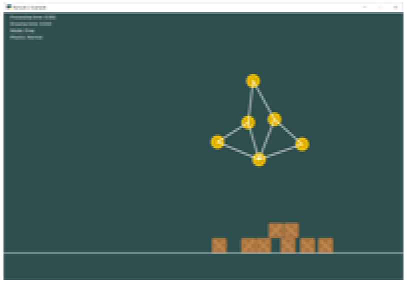


Fig. 97: pymunk_joint_builder

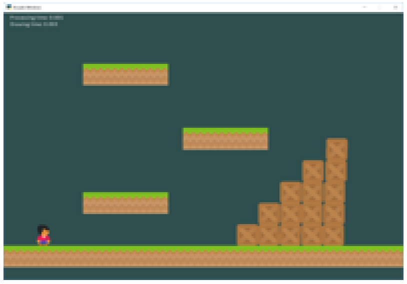


Fig. 98: *Pymunk Platformer*

5.10.2 Frame Buffers

Fig. 99: minimap

5.11 Concept Games

5.12 Odds and Ends

5.13 Tutorials

5.13.1 Particle System

5.14 Stress Tests

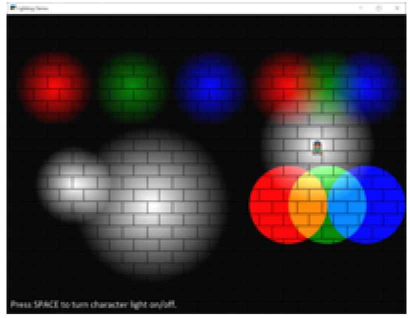


Fig. 100: light_demo

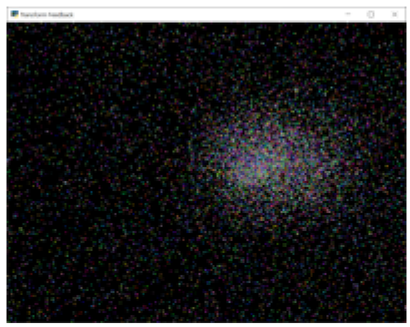


Fig. 101: transform_feedback

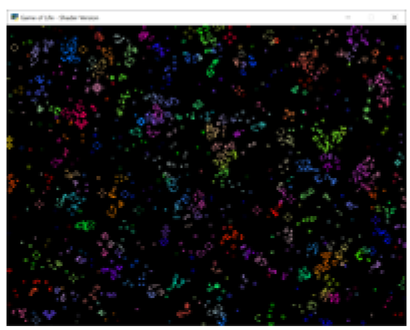


Fig. 102: game_of_life_fbo

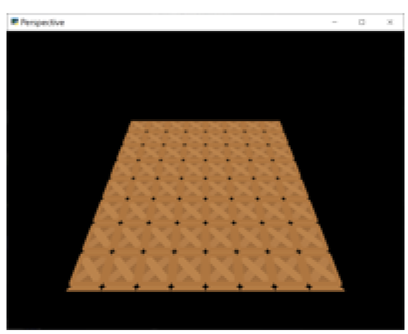


Fig. 103: perspective

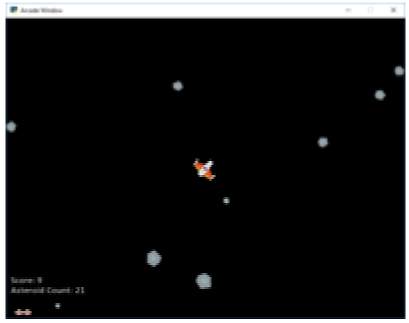


Fig. 104: asteroid_smasher

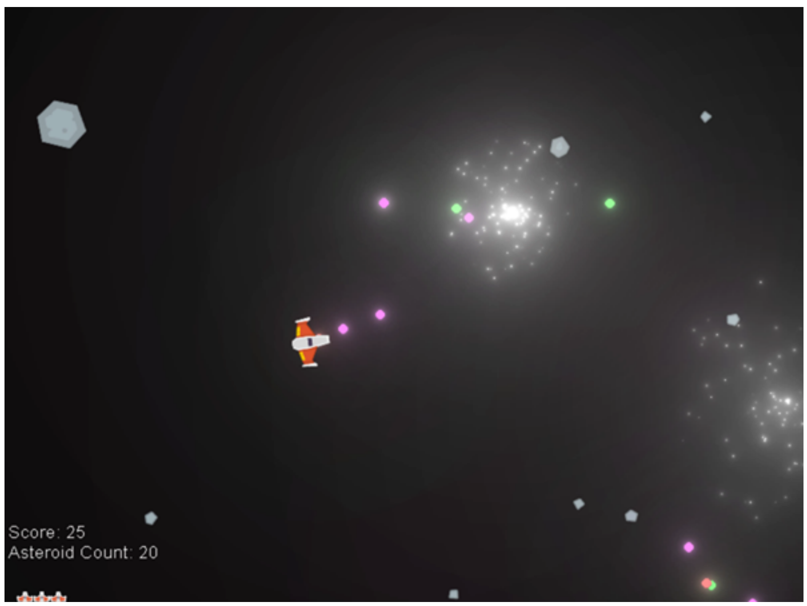


Fig. 105: Asteroids with Shaders

Fig. 106: slime_invaders

Fig. 107: Community RPG

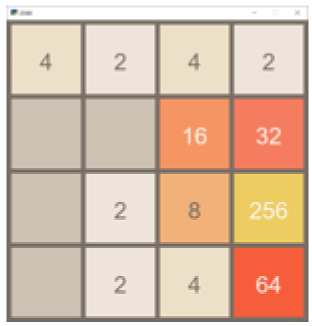


Fig. 108: 2048

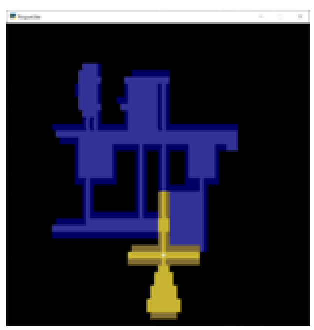


Fig. 109: Rogue-Like

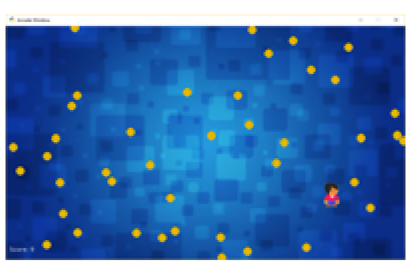


Fig. 110: sprite_collect_coins_background

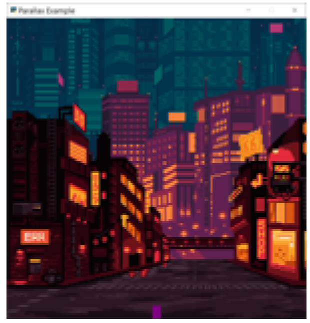


Fig. 111: parallax

Fig. 112: timer

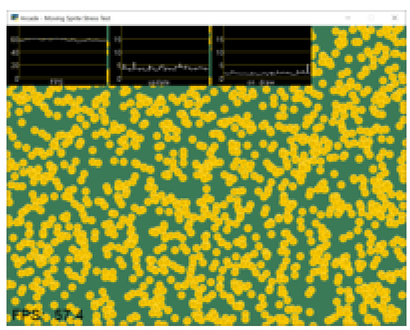


Fig. 113: performance_statistics_example

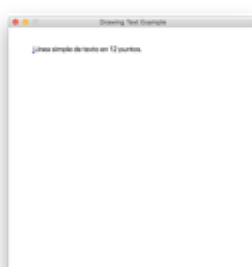


Fig. 114: text_loc_example

Fig. 115: *Simple Platformer*

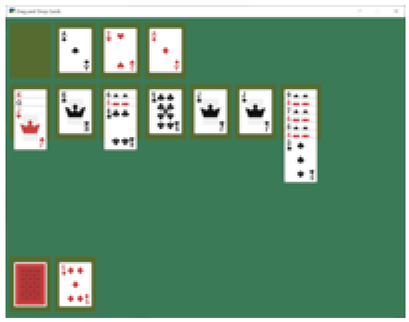


Fig. 116: *Solitaire Tutorial*

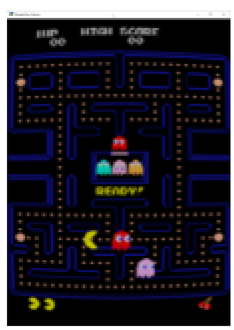


Fig. 117: *CRT Filter*

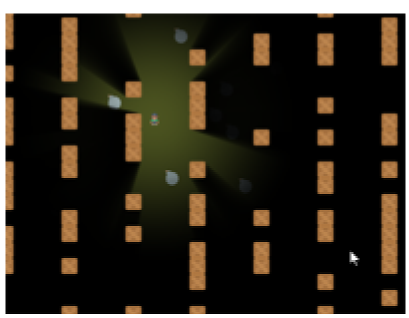


Fig. 118: *Ray-casting Shadows*

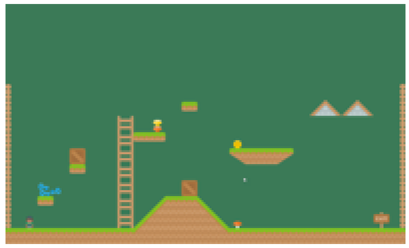


Fig. 119: *Pymunk Platformer*

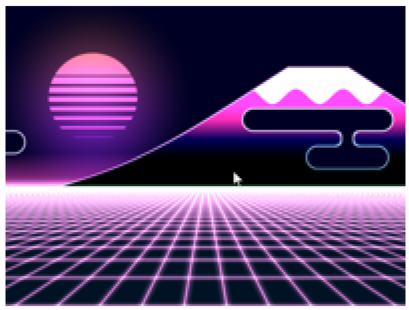


Fig. 120: *Shader Toy Tutorial - Glow*

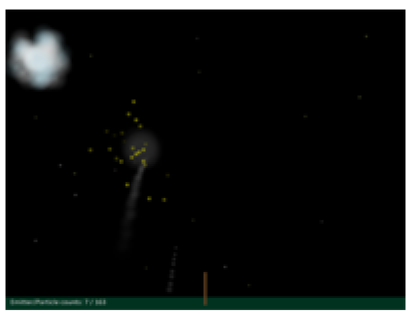


Fig. 121: particle_fireworks

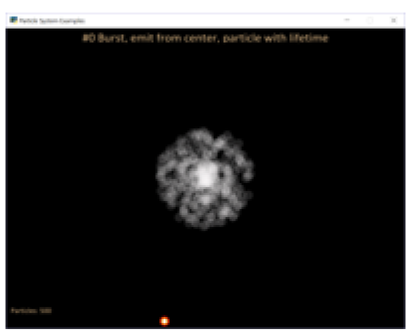


Fig. 122: particle_systems

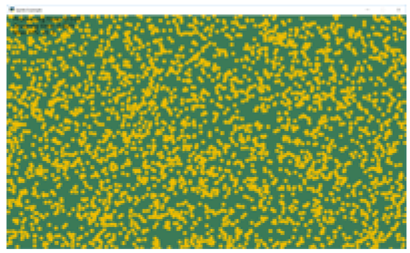


Fig. 123: stress_test_draw_moving

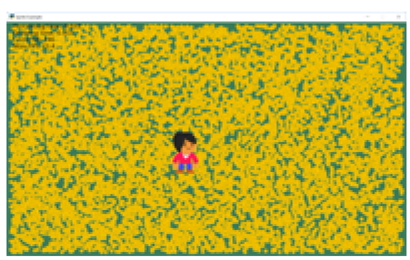


Fig. 124: stress_test_collision

CHAPTER

SIX

QUICK API INDEX

- *arcade.color package*
- *arcade.csscolor package*
- *arcade.key package*
- *Built-In Resources*

6.1 The arcade module

Name	Group
`arcade.Texture`	Texture Management
`arcade.cleanup_texture_cache()`	Texture Management
`arcade.load_spritesheet()`	Texture Management
`arcade.load_texture()`	Texture Management
`arcade.load_texture_pair()`	Texture Management
`arcade.load_textures()`	Texture Management
`arcade.make_circle_texture()`	Texture Management
`arcade.make_soft_circle_texture()`	Texture Management
`arcade.make_soft_square_texture()`	Texture Management
`arcade.trim_image()`	Texture Management
`arcade.TiledObject`	Arcade Data Types
`arcade.close_window()`	Window and View
`arcade.create_orthogonal_projection()`	Window and View
`arcade.exit()`	Window and View
`arcade.finish_render()`	Window and View
`arcade.get_display_size()`	Window and View
`arcade.get_projection()`	Window and View
`arcade.get_scaling_factor()`	Window and View
`arcade.get_viewport()`	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.set_background_color()`	Window and View
`arcade.set_viewport()`	Window and View
`arcade.set_window()`	Window and View
`arcade.start_render()`	Window and View

continues on next page

Table 1 – continued from previous page

Name	Group
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_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
arcade.get_image()	Drawing - Primitives
arcade.get_pixel()	Drawing - Primitives
arcade.create_text_image()	Text - Image/Pillow based
arcade.create_text_sprite()	Text - Image/Pillow based
arcade.calculate_hit_box_points_detailed()	Geometry Support
arcade.calculate_hit_box_points_simple()	Geometry Support
arcade.AtlasRegion	Texture Atlas
arcade.TextureAtlas	Texture Atlas
arcade.PymunkException	Physics Engines
arcade.PymunkPhysicsEngine	Physics Engines
arcade.PymunkPhysicsObject	Physics Engines
arcade.are_polygons_intersecting()	Geometry Support
arcade.is_point_in_polygon()	Geometry Support
arcade.AStarBarrierList	Pathfinding
arcade.astar_calculate_path()	Pathfinding
arcade.EasingData	Geometry Support
arcade.ease_angle()	Geometry Support
arcade.ease_angle_update()	Geometry Support
arcade.ease_in()	Geometry Support
arcade.ease_in_back()	Geometry Support
arcade.ease_in_out()	Geometry Support
arcade.ease_in_out_sin()	Geometry Support

continues on next page

Table 1 – continued from previous page

Name	Group
arcade.ease_in_sin()	Geometry Support
arcade.ease_out()	Geometry Support
arcade.ease_out_back()	Geometry Support
arcade.ease_out_bounce()	Geometry Support
arcade.ease_out_elastic()	Geometry Support
arcade.ease_out_sin()	Geometry Support
arcade.ease_position()	Geometry Support
arcade.ease_update()	Geometry Support
arcade.ease_value()	Geometry Support
arcade.easing()	Geometry Support
arcade.linear()	Geometry Support
arcade.smoothstep()	Geometry Support
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.Scene	Sprite Scenes
arcade.configure_logging()	Misc Utility Functions
arcade.EternalParticle	Particles
arcade.FadeParticle	Particles
arcade.LifetimeParticle	Particles
arcade.Particle	Particles
arcade.Shape	Drawing - Batch
arcade.ShapeElementList	Drawing - Batch
arcade.create_ellipse()	Drawing - Batch
arcade.create_ellipse_filled()	Drawing - Batch
arcade.create_ellipse_filled_with_colors()	Drawing - Batch
arcade.create_ellipse_outline()	Drawing - Batch
arcade.create_line()	Drawing - Batch
arcade.create_line_generic()	Drawing - Batch
arcade.create_line_generic_with_colors()	Drawing - Batch
arcade.create_line_loop()	Drawing - Batch
arcade.create_line_strip()	Drawing - Batch
arcade.create_lines()	Drawing - Batch
arcade.create_lines_with_colors()	Drawing - Batch
arcade.create_polygon()	Drawing - Batch
arcade.create_rectangle()	Drawing - Batch
arcade.create_rectangle_filled()	Drawing - Batch
arcade.create_rectangle_filled_with_colors()	Drawing - Batch
arcade.create_rectangle_outline()	Drawing - Batch
arcade.create_rectangles_filled_with_colors()	Drawing - Batch
arcade.create_triangles_filled_with_colors()	Drawing - Batch
arcade.get_rectangle_points()	Drawing - Batch
arcade.Text	Text - Pyglet/Glyph based
arcade.draw_text()	Text - Pyglet/Glyph based
arcade.load_font()	Text - Pyglet/Glyph based
arcade.color_from_hex_string()	Drawing - Utility
arcade.float_to_byte_color()	Drawing - Utility

continues on next page

Table 1 – continued from previous page

Name	Group
arcade.get_four_byte_color()	Drawing - Utility
arcade.get_four_float_color()	Drawing - Utility
arcade.get_points_for_thick_line()	Drawing - Utility
arcade.get_three_float_color()	Drawing - Utility
arcade.make_transparent_color()	Drawing - Utility
arcade.uint24_to_three_byte_color()	Drawing - Utility
arcade.uint32_to_four_byte_color()	Drawing - Utility
arcade.AnimatedTimeBasedSprite	Sprites
arcade.AnimatedWalkingSprite	Sprites
arcade.AnimationKeyframe	Sprites
arcade.PyMunk	Sprites
arcade.Sprite	Sprites
arcade.SpriteCircle	Sprites
arcade.SpriteSolidColor	Sprites
arcade.get_distance_between_sprites()	Sprites
arcade.load_animated_gif()	Sprites
arcade.earclip()	Geometry Support
arcade.generate_uuid_from_kwargs()	Misc Utility Functions
arcade.lerp()	Misc Utility Functions
arcade.lerp_vec()	Misc Utility Functions
arcade.rand_angle_360_deg()	Misc Utility Functions
arcade.rand_angle_spread_deg()	Misc Utility Functions
arcade.rand_in_circle()	Misc Utility Functions
arcade.rand_in_rect()	Misc Utility Functions
arcade.rand_on_circle()	Misc Utility Functions
arcade.rand_on_line()	Misc Utility Functions
arcade.rand_vec_magnitude()	Misc Utility Functions
arcade.rand_vec_spread_deg()	Misc Utility Functions
arcade.get_game_controllers()	Game Controller Support
arcade.get_joysticks()	Game Controller Support
arcade.clamp()	Geometry Support
arcade.get_angle_degrees()	Geometry Support
arcade.get_angle_radians()	Geometry Support
arcade.get_distance()	Geometry Support
arcade.rotate_point()	Geometry Support
arcade.Section	Window and View
arcade.SectionManager	Window and View
arcade.ArcadeContext	OpenGL Context
arcade.Sound	Sound
arcade.load_sound()	Sound
arcade.play_sound()	Sound
arcade.stop_sound()	Sound
arcade.EmitBurst	Particles
arcade.EmitController	Particles
arcade.EmitInterval	Particles
arcade.EmitMaintainCount	Particles
arcade.Emitter	Particles
arcade.EmitterIntervalWithCount	Particles
arcade.EmitterIntervalWithTime	Particles

continues on next page

Table 1 – continued from previous page

Name	Group
arcade.make_burst_emitter()	Particles
arcade.make_interval_emitter()	Particles
arcade.has_line_of_sight()	Pathfinding
arcade.PerfGraph	Performance Information
arcade.create_isometric_grid_lines()	Isometric Map Support (incomplete)
arcade.isometric_grid_to_screen()	Isometric Map Support (incomplete)
arcade.screen_to_isometric_grid()	Isometric Map Support (incomplete)
arcade.Camera	Camera
arcade.PhysicsEnginePlatformer	Physics Engines
arcade.PhysicsEngineSimple	Physics Engines
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.SpriteList	Sprite Lists
arcade.check_for_collision()	Sprite Lists
arcade.check_for_collision_with_list()	Sprite Lists
arcade.check_for_collision_with_lists()	Sprite Lists
arcade.get_closest_sprite()	Sprite Lists
arcade.get_sprites_at_exact_point()	Sprite Lists
arcade.get_sprites_at_point()	Sprite Lists

6.2 The arcade.gui module

Name	Group
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.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.UIMessageBox	GUI
arcade.gui.UIDraggableMixin	GUI
arcade.gui.UIMouseFilterMixin	GUI

continues on next page

Table 2 – continued from previous page

Name	Group
arcade.gui.UIWindowLikeMixin	GUI
arcade.gui.UIAnchorWidget	GUI Widgets
arcade.gui.UIBorder	GUI Widgets
arcade.gui.UIBoxLayout	GUI Widgets
arcade.gui.UIDummy	GUI Widgets
arcade.gui.UIFlatButton	GUI Widgets
arcade.gui.UIInputText	GUI Widgets
arcade.gui.UIInteractiveWidget	GUI Widgets
arcade.gui.UILabel	GUI Widgets
arcade.gui.UILayout	GUI Widgets
arcade.gui.UIPadding	GUI Widgets
arcade.gui.UISpace	GUI Widgets
arcade.gui.UISpriteWidget	GUI Widgets
arcade.gui.UITextArea	GUI Widgets
arcade.gui.UITextureButton	GUI Widgets
arcade.gui.UITexturePane	GUI Widgets
arcade.gui.UIWidget	GUI Widgets
arcade.gui.UIWidgetParent	GUI Widgets
arcade.gui.UIWrapper	GUI Widgets
arcade.gui.Surface	GUI
arcade.gui.UIManager	GUI

6.3 The arcade.tilemap module

Name	Group
arcade.tilemap.TileMap	Tiled Map Reader
arcade.tilemap.load_tilemap()	Tiled Map Reader
arcade.tilemap.read_tmx()	Tiled Map Reader

CHAPTER

SEVEN

ARCADE PACKAGE API

This page documents the Application Programming Interface (API) for the Python Arcade library. See also:

- *Quick API Index*
- *How-To Example Code*

7.1 Arcade Data Types

7.1.1 arcade.TiledObject

class arcade.**TiledObject**(*shape*, *properties*, *name*, *type*)

name: Optional[str]

Alias for field number 2

properties: Optional[Dict[str, Union[float, pathlib.Path, str, bool, pytiled_parser.common_types.Color]]]

Alias for field number 1

shape: Union[Tuple[float, float], List[float], Sequence[Union[Tuple[float, float], List[float]]], Tuple[float, float, float, float]]

Alias for field number 0

type: Optional[str]

Alias for field number 3

7.2 Drawing - Primitives

7.2.1 arcade.draw_arc_filled

arcade.**draw_arc_filled**(*center_x: float*, *center_y: float*, *width: float*, *height: float*, *color: Union[Tuple[int, int, int], List[int], 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** (*float*) – x position that is the center of the arc.
- **center_y** (*float*) – y position that is the center of the arc.

- **width** (*float*) – width of the arc.
- **height** (*float*) – height of the arc.
- **color** (*Color*) – color, specified in a list of 3 or 4 bytes in RGB or RGBA format.
- **start_angle** (*float*) – start angle of the arc in degrees.
- **end_angle** (*float*) – end angle of the arc in degrees.
- **tilt_angle** (*float*) – angle the arc is tilted.
- **num_segments** (*float*) – Number of line segments used to draw arc.

7.2.2 arcade.draw_arc_outline

arcade.**draw_arc_outline**(*center_x: float, center_y: float, width: float, height: float, color: Union[Tuple[int, int, int], List[int], 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** (*float*) – x position that is the center of the arc.
- **center_y** (*float*) – y position that is the center of the arc.
- **width** (*float*) – width of the arc.
- **height** (*float*) – height of the arc.
- **color** (*Color*) – color, specified in a list of 3 or 4 bytes in RGB or RGBA format.
- **start_angle** (*float*) – start angle of the arc in degrees.
- **end_angle** (*float*) – end angle of the arc in degrees.
- **border_width** (*float*) – width of line in pixels.
- **tilt_angle** (*float*) – angle the arc is tilted.
- **num_segments** (*int*) – float of triangle segments that make up this circle. Higher is better quality, but slower render time.

7.2.3 arcade.draw_circle_filled

arcade.**draw_circle_filled**(*center_x: float, center_y: float, radius: float, color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]], tilt_angle: float = 0, num_segments: int = - 1*)

Draw a filled-in circle.

Parameters

- **center_x** (*float*) – x position that is the center of the circle.
- **center_y** (*float*) – y position that is the center of the circle.
- **radius** (*float*) – width of the circle.
- **color** (*Color*) – color, specified in a list of 3 or 4 bytes in RGB or RGBA format.
- **tilt_angle** (*float*) – Angle in degrees to tilt the circle. Useful for low segment count circles

- **num_segments** (*int*) – 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.

7.2.4 arcade.draw_circle_outline

`arcade.draw_circle_outline(center_x: float, center_y: float, radius: float, color: Union[Tuple[int, int, int], List[int], 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** (*float*) – x position that is the center of the circle.
- **center_y** (*float*) – y position that is the center of the circle.
- **radius** (*float*) – width of the circle.
- **color** (*Color*) – color, specified in a list of 3 or 4 bytes in RGB or RGBA format.
- **border_width** (*float*) – Width of the circle outline in pixels.
- **tilt_angle** (*float*) – Angle in degrees to tilt the circle. Useful for low segment count circles
- **num_segments** (*int*) – 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.

7.2.5 arcade.draw_ellipse_filled

`arcade.draw_ellipse_filled(center_x: float, center_y: float, width: float, height: float, color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]], tilt_angle: float = 0, num_segments: int = - 1)`

Draw a filled in ellipse.

Parameters

- **center_x** (*float*) – x position that is the center of the circle.
- **center_y** (*float*) – y position that is the center of the circle.
- **width** (*float*) – width of the ellipse.
- **height** (*float*) – height of the ellipse.
- **color** (*Color*) – color, specified in a list of 3 or 4 bytes in RGB or RGBA format.
- **tilt_angle** (*float*) – Angle in degrees to tilt the ellipse.
- **num_segments** (*int*) – 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.

7.2.6 arcade.draw_ellipse_outline

arcade.**draw_ellipse_outline**(*center_x: float, center_y: float, width: float, height: float, color: Union[Tuple[int, int, int], List[int], 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** (*float*) – x position that is the center of the circle.
- **center_y** (*float*) – y position that is the center of the circle.
- **width** (*float*) – width of the ellipse.
- **height** (*float*) – height of the ellipse.
- **color** (*Color*) – color, specified in a list of 3 or 4 bytes in RGB or RGBA format.
- **border_width** (*float*) – Width of the circle outline in pixels.
- **tilt_angle** (*float*) – Angle in degrees to tilt the ellipse.
- **num_segments** (*int*) – 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.
- **tilt_angle** – Tile of the circle. Useful when drawing a circle with a low segment count

7.2.7 arcade.draw_line

arcade.**draw_line**(*start_x: float, start_y: float, end_x: float, end_y: float, color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]], line_width: float = 1*)

Draw a line.

Parameters

- **start_x** (*float*) – x position of line starting point.
- **start_y** (*float*) – y position of line starting point.
- **end_x** (*float*) – x position of line ending point.
- **end_y** (*float*) – y position of line ending point.
- **color** (*Color*) – color, specified in a list of 3 or 4 bytes in RGB or RGBA format.
- **line_width** (*float*) – Width of the line in pixels.

7.2.8 arcade.draw_line_strip

arcade.**draw_line_strip**(*point_list: Sequence[Union[Tuple[float, float], List[float]]], color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]], line_width: float = 1*)

Draw a multi-point line.

Parameters

- **point_list** (*PointList*) – List of x, y points that make up this strip
- **color** (*Color*) – Color of line strip

- **line_width** (*float*) – Width of the line

7.2.9 arcade.draw_lines

arcade.**draw_lines**(*point_list: Sequence[Union[Tuple[float, float], List[float]]], color: Union[Tuple[int, int, int], List[int], 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** (*PointList*) – List of points making up the lines. Each point is in a list. So it is a list of lists.
- **color** (*Color*) – color, specified in a list of 3 or 4 bytes in RGB or RGBA format.
- **line_width** (*float*) – Width of the line in pixels.

7.2.10 arcade.draw_lrtb_rectangle_filled

arcade.**draw_lrtb_rectangle_filled**(*left: float, right: float, top: float, bottom: float, color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]]*)

Draw a rectangle by specifying left, right, top, and bottom edges.

Parameters

- **left** (*float*) – The x coordinate of the left edge of the rectangle.
- **right** (*float*) – The x coordinate of the right edge of the rectangle.
- **top** (*float*) – The y coordinate of the top of the rectangle.
- **bottom** (*float*) – The y coordinate of the rectangle bottom.
- **color** (*Color*) – The color of the rectangle.

Raises AttributeError Raised if left > right or top < bottom.

7.2.11 arcade.draw_lrtb_rectangle_outline

arcade.**draw_lrtb_rectangle_outline**(*left: float, right: float, top: float, bottom: float, color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]], border_width: float = 1*)

Draw a rectangle by specifying left, right, top, and bottom edges.

Parameters

- **left** (*float*) – The x coordinate of the left edge of the rectangle.
- **right** (*float*) – The x coordinate of the right edge of the rectangle.
- **top** (*float*) – The y coordinate of the top of the rectangle.
- **bottom** (*float*) – The y coordinate of the rectangle bottom.
- **color** (*Color*) – The color of the rectangle.
- **border_width** (*float*) – The width of the border in pixels. Defaults to one.

Raises AttributeError Raised if left > right or top < bottom.

7.2.12 arcade.draw_lrwh_rectangle_textured

arcade.**draw_lrwh_rectangle_textured**(*bottom_left_x: float*, *bottom_left_y: float*, *width: float*, *height: float*, *texture: arcade.texture.Texture*, *angle: float = 0*, *alpha: int = 255*)

Draw a texture extending from bottom left to top right.

Parameters

- **bottom_left_x** (*float*) – The x coordinate of the left edge of the rectangle.
- **bottom_left_y** (*float*) – The y coordinate of the bottom of the rectangle.
- **width** (*float*) – The width of the rectangle.
- **height** (*float*) – The height of the rectangle.
- **texture** (*int*) – identifier of texture returned from load_texture() call
- **angle** (*float*) – rotation of the rectangle. Defaults to zero.
- **alpha** (*int*) – Transparency of image. 0 is fully transparent, 255 (default) is visible

7.2.13 arcade.draw_parabola_filled

arcade.**draw_parabola_filled**(*start_x: float*, *start_y: float*, *end_x: float*, *height: float*, *color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]]*, *tilt_angle: float = 0*)

Draws a filled in parabola.

Parameters

- **start_x** (*float*) – The starting x position of the parabola
- **start_y** (*float*) – The starting y position of the parabola
- **end_x** (*float*) – The ending x position of the parabola
- **height** (*float*) – The height of the parabola
- **color** (*Color*) – The color of the parabola
- **tilt_angle** (*float*) – The angle of the tilt of the parabola

7.2.14 arcade.draw_parabola_outline

arcade.**draw_parabola_outline**(*start_x: float*, *start_y: float*, *end_x: float*, *height: float*, *color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]]*, *border_width: float = 1*, *tilt_angle: float = 0*)

Draws the outline of a parabola.

Parameters

- **start_x** (*float*) – The starting x position of the parabola
- **start_y** (*float*) – The starting y position of the parabola
- **end_x** (*float*) – The ending x position of the parabola
- **height** (*float*) – The height of the parabola
- **color** (*Color*) – The color of the parabola
- **border_width** (*float*) – The width of the parabola

- **tilt_angle** (*float*) – The angle of the tilt of the parabola

7.2.15 arcade.draw_point

arcade.**draw_point**(*x: float*, *y: float*, *color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]]*, *size: float*)

Draw a point.

Parameters

- **x** (*float*) – x position of point.
- **y** (*float*) – y position of point.
- **color** (*Color*) – color, specified in a list of 3 or 4 bytes in RGB or RGBA format.
- **size** (*float*) – Size of the point in pixels.

7.2.16 arcade.draw_points

arcade.**draw_points**(*point_list: Sequence[Union[Tuple[float, float], List[float]]]*, *color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]]*, *size: float = 1*)

Draw a set of points.

Parameters

- **point_list** (*PointList*) – List of points Each point is in a list. So it is a list of lists.
- **color** (*Color*) – color, specified in a list of 3 or 4 bytes in RGB or RGBA format.
- **size** (*float*) – Size of the point in pixels.

7.2.17 arcade.draw_polygon_filled

arcade.**draw_polygon_filled**(*point_list: Sequence[Union[Tuple[float, float], List[float]]]*, *color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]]*)

Draw a polygon that is filled in.

Parameters

- **point_list** (*PointList*) – List of points making up the lines. Each point is in a list. So it is a list of lists.
- **color** (*Color*) – The color, specified in RGB or RGBA format.

7.2.18 arcade.draw_polygon_outline

arcade.**draw_polygon_outline**(*point_list: Sequence[Union[Tuple[float, float], List[float]]]*, *color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]]*, *line_width: float = 1*)

Draw a polygon outline. Also known as a "line loop."

Parameters

- **point_list** (*PointList*) – List of points making up the lines. Each point is in a list. So it is a list of lists.

- **color** (*Color*) – color, specified in a list of 3 or 4 bytes in RGB or RGBA format.
- **line_width** (*int*) – Width of the line in pixels.

7.2.19 arcade.draw_rectangle_filled

arcade.**draw_rectangle_filled**(*center_x: float, center_y: float, width: float, height: float, color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]], tilt_angle: float = 0*)

Draw a filled-in rectangle.

Parameters

- **center_x** (*float*) – x coordinate of rectangle center.
- **center_y** (*float*) – y coordinate of rectangle center.
- **width** (*float*) – width of the rectangle.
- **height** (*float*) – height of the rectangle.
- **color** (*Color*) – color, specified in a list of 3 or 4 bytes in RGB or RGBA format.
- **tilt_angle** (*float*) – rotation of the rectangle. Defaults to zero.

7.2.20 arcade.draw_rectangle_outline

arcade.**draw_rectangle_outline**(*center_x: float, center_y: float, width: float, height: float, color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]], border_width: float = 1, tilt_angle: float = 0*)

Draw a rectangle outline.

Parameters

- **center_x** (*float*) – x coordinate of top left rectangle point.
- **center_y** (*float*) – y coordinate of top left rectangle point.
- **width** (*float*) – width of the rectangle.
- **height** (*float*) – height of the rectangle.
- **color** (*Color*) – color, specified in a list of 3 or 4 bytes in RGB or RGBA format.
- **border_width** (*float*) – width of the lines, in pixels.
- **tilt_angle** (*float*) – rotation of the rectangle. Defaults to zero.

7.2.21 arcade.draw_scaled_texture_rectangle

arcade.**draw_scaled_texture_rectangle**(*center_x: float, center_y: float, texture:* arcade.texture.Texture, *scale: float = 1.0, angle: float = 0, alpha: int = 255*)

Draw a textured rectangle on-screen.

Parameters

- **center_x** (*float*) – x coordinate of rectangle center.
- **center_y** (*float*) – y coordinate of rectangle center.

- **texture** (*int*) – identifier of texture returned from load_texture() call
- **scale** (*float*) – scale of texture
- **angle** (*float*) – rotation of the rectangle. Defaults to zero.
- **alpha** (*float*) – Transparency of image. 0 is fully transparent, 255 (default) is visible

7.2.22 arcade.draw_texture_rectangle

arcade.**draw_texture_rectangle**(*center_x: float*, *center_y: float*, *width: float*, *height: float*, *texture: arcade.texture.Texture*, *angle: float = 0*, *alpha: int = 255*)

Draw a textured rectangle on-screen.

Parameters

- **center_x** (*float*) – x coordinate of rectangle center.
- **center_y** (*float*) – y coordinate of rectangle center.
- **width** (*float*) – width of texture
- **height** (*float*) – height of texture
- **texture** (*int*) – identifier of texture returned from load_texture() call
- **angle** (*float*) – rotation of the rectangle. Defaults to zero.
- **alpha** (*float*) – Transparency of image. 0 is fully transparent, 255 (default) is visible

7.2.23 arcade.draw_triangle_filled

arcade.**draw_triangle_filled**(*x1: float*, *y1: float*, *x2: float*, *y2: float*, *x3: float*, *y3: float*, *color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]]*)

Draw a filled in triangle.

Parameters

- **x1** (*float*) – x value of first coordinate.
- **y1** (*float*) – y value of first coordinate.
- **x2** (*float*) – x value of second coordinate.
- **y2** (*float*) – y value of second coordinate.
- **x3** (*float*) – x value of third coordinate.
- **y3** (*float*) – y value of third coordinate.
- **color** (*Color*) – Color of triangle.

7.2.24 arcade.draw_triangle_outline

`arcade.draw_triangle_outline`(*x1: float, y1: float, x2: float, y2: float, x3: float, y3: float, color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]], border_width: float = 1*)

Draw a the outline of a triangle.

Parameters

- **x1** (*float*) – x value of first coordinate.
- **y1** (*float*) – y value of first coordinate.
- **x2** (*float*) – x value of second coordinate.
- **y2** (*float*) – y value of second coordinate.
- **x3** (*float*) – x value of third coordinate.
- **y3** (*float*) – y value of third coordinate.
- **color** (*Color*) – Color of triangle.
- **border_width** (*float*) – Width of the border in pixels. Defaults to 1.

7.2.25 arcade.draw_xywh_rectangle_filled

`arcade.draw_xywh_rectangle_filled`(*bottom_left_x: float, bottom_left_y: float, width: float, height: float, color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]]*)

Draw a filled rectangle extending from bottom left to top right

Parameters

- **bottom_left_x** (*float*) – The x coordinate of the left edge of the rectangle.
- **bottom_left_y** (*float*) – The y coordinate of the bottom of the rectangle.
- **width** (*float*) – The width of the rectangle.
- **height** (*float*) – The height of the rectangle.
- **color** (*Color*) – The color of the rectangle.

7.2.26 arcade.draw_xywh_rectangle_outline

`arcade.draw_xywh_rectangle_outline`(*bottom_left_x: float, bottom_left_y: float, width: float, height: float, color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]], border_width: float = 1*)

Draw a rectangle extending from bottom left to top right

Parameters

- **bottom_left_x** (*float*) – The x coordinate of the left edge of the rectangle.
- **bottom_left_y** (*float*) – The y coordinate of the bottom of the rectangle.
- **width** (*float*) – The width of the rectangle.
- **height** (*float*) – The height of the rectangle.
- **color** (*Color*) – The color of the rectangle.

- **border_width** (*float*) – The width of the border in pixels. Defaults to one.

7.2.27 arcade.get_image

arcade.**get_image**(*x: int = 0, y: int = 0, width: Optional[int] = None, height: Optional[int] = None*) → PIL.Image.Image

Get an image from the screen.

Example:

```
image = get_image()
image.save('screenshot.png', 'PNG')
```

Parameters

- **x** (*int*) – Start (left) x location
- **y** (*int*) – Start (top) y location
- **width** (*int*) – Width of image. Leave blank for grabbing the 'rest' of the image
- **height** (*int*) – Height of image. Leave blank for grabbing the 'rest' of the image

Returns A Pillow Image

Return type PIL.Image.Image

7.2.28 arcade.get_pixel

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** (*int*) – x location
- **y** (*int*) – y location
- **components** (*int*) – Number of components to fetch. By default we fetch 3 3 components (RGB). 4 componets would be RGBA.

Return type Color

7.3 Drawing - Batch

7.3.1 arcade.Shape

class arcade.**Shape**

Primitive drawing shape. This can be part of a ShapeElementList so shapes can be drawn faster in batch.

draw()

Draw this shape. Drawing this way isn't as fast as drawing multiple shapes batched together in a ShapeElementList.

7.3.2 arcade.ShapeElementList

class arcade.**ShapeElementList**

A program can put multiple drawing primitives in a ShapeElementList, and then move and draw them as one. Do this when you want to create a more complex object out of simpler primitives. This also speeds rendering as all objects are drawn in one operation.

property angle: float

Get the angle of the ShapeElementList in degrees.

append(*item: arcade.buffered_draw_commands.TShape*)

Add a new shape to the list.

property center_x: float

Get the center x coordinate of the ShapeElementList.

property center_y: float

Get the center y coordinate of the ShapeElementList.

draw()

Draw everything in the list.

move(*change_x: float*, *change_y: float*)

Move all the shapes ion the list :param change_x: Amount to move on the x axis :param change_y: Amount to move on the y axis

remove(*item: arcade.buffered_draw_commands.TShape*)

Remove a specific shape from the list.

7.3.3 arcade.create_ellipse

arcade.**create_ellipse**(*center_x: float*, *center_y: float*, *width: float*, *height: float*, *color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]]*, *border_width: float = 1*, *tilt_angle: float = 0*, *num_segments: int = 32*, *filled=True*) → *arcade.buffered_draw_commands.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.

7.3.4 arcade.create_ellipse_filled

arcade.**create_ellipse_filled**(*center_x: float*, *center_y: float*, *width: float*, *height: float*, *color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]]*, *tilt_angle: float = 0*, *num_segments: int = 128*) → *arcade.buffered_draw_commands.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.

7.3.5 arcade.create_ellipse_filled_with_colors

arcade.**create_ellipse_filled_with_colors**(*center_x: float, center_y: float, width: float, height: float, outside_color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]], inside_color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]], tilt_angle: float = 0, num_segments: int = 32*) → *arcade.buffered_draw_commands.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** (*float*) –
- **center_y** (*float*) –
- **width** (*float*) –
- **height** (*float*) –
- **outside_color** (*Color*) –
- **inside_color** (*float*) –
- **tilt_angle** (*float*) –
- **num_segments** (*int*) –

Returns Shape

7.3.6 arcade.create_ellipse_outline

arcade.**create_ellipse_outline**(*center_x: float, center_y: float, width: float, height: float, color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]], border_width: float = 1, tilt_angle: float = 0, num_segments: int = 128*) → *arcade.buffered_draw_commands.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.

7.3.7 arcade.create_line

arcade.**create_line**(*start_x: float*, *start_y: float*, *end_x: float*, *end_y: float*, *color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]]*, *line_width: float = 1*) → *arcade.buffered_draw_commands.Shape*

Create a 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.

Parameters

- **start_x** (*float*) –
- **start_y** (*float*) –
- **end_x** (*float*) –
- **end_y** (*float*) –
- **color** (*Color*) –
- **line_width** (*float*) –

Returns Shape

7.3.8 arcade.create_line_generic

arcade.**create_line_generic**(*point_list: Sequence[Union[Tuple[float, float], List[float]]]*, *color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]]*, *shape_mode: int*, *line_width: float = 1*) → *arcade.buffered_draw_commands.Shape*

This function is used by `create_line_strip` and `create_line_loop`, just changing the OpenGL type for the line drawing.

7.3.9 arcade.create_line_generic_with_colors

arcade.**create_line_generic_with_colors**(*point_list: Sequence[Union[Tuple[float, float], List[float]]]*, *color_list: Iterable[Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]]]*, *shape_mode: int*, *line_width: float = 1*) → *arcade.buffered_draw_commands.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** (*PointList*) –
- **color_list** (*Iterable[Color]*) –
- **shape_mode** (*float*) –
- **line_width** (*float*) –

Returns Shape

7.3.10 arcade.create_line_loop

arcade.**create_line_loop**(*point_list: Sequence[Union[Tuple[float, float], List[float]]], color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]], line_width: float = 1*)

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** (*PointList*) –
- **color** (*Color*) –
- **line_width** (*float*) –

Returns Shape

7.3.11 arcade.create_line_strip

arcade.**create_line_strip**(*point_list: Sequence[Union[Tuple[float, float], List[float]]], color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]], line_width: float = 1*)

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** (*PointList*) –
- **color** (*Color*) –
- **line_width** (*PointList*) –

Returns Shape

7.3.12 arcade.create_lines

arcade.**create_lines**(*point_list: Sequence[Union[Tuple[float, float], List[float]]], color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]], line_width: float = 1*)

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** (*PointList*) –
- **color** (*Color*) –
- **line_width** (*float*) –

Returns Shape

7.3.13 arcade.create_lines_with_colors

`arcade.create_lines_with_colors`(*point_list: Sequence[Union[Tuple[float, float], List[float]]], color_list: Sequence[Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]]], line_width: float = 1*)

7.3.14 arcade.create_polygon

`arcade.create_polygon`(*point_list: Sequence[Union[Tuple[float, float], List[float]]], color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]]*)

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** (*PointList*) –
- **color** –

Returns Shape

7.3.15 arcade.create_rectangle

`arcade.create_rectangle`(*center_x: float, center_y: float, width: float, height: float, color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]], border_width: float = 1, tilt_angle: float = 0, filled=True*) → *arcade.buffered_draw_commands.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** (*float*) –
- **center_y** (*float*) –
- **width** (*float*) –
- **height** (*float*) –
- **color** (*Color*) –
- **border_width** (*float*) –
- **tilt_angle** (*float*) –
- **filled** (*bool*) –

7.3.16 arcade.create_rectangle_filled

arcade.**create_rectangle_filled**(*center_x: float, center_y: float, width: float, height: float, color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]], tilt_angle: float = 0*) → *arcade.buffered_draw_commands.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** (*float*) –
- **center_y** (*float*) –
- **width** (*float*) –
- **height** (*float*) –
- **color** (*Color*) –
- **tilt_angle** (*float*) –

Returns Shape

7.3.17 arcade.create_rectangle_filled_with_colors

arcade.**create_rectangle_filled_with_colors**(*point_list, color_list*) → *arcade.buffered_draw_commands.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.

7.3.18 arcade.create_rectangle_outline

arcade.**create_rectangle_outline**(*center_x: float, center_y: float, width: float, height: float, color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]], border_width: float = 1, tilt_angle: float = 0*) → *arcade.buffered_draw_commands.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** (*float*) –
- **center_y** (*float*) –

- **width** (*float*) –
- **height** (*float*) –
- **color** (*Color*) –
- **border_width** (*Color*) –
- **tilt_angle** (*float*) –

Returns: Shape

7.3.19 arcade.create_rectangles_filled_with_colors

arcade.**create_rectangles_filled_with_colors**(*point_list*, *color_list*) → *arcade.buffered_draw_commands.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.

7.3.20 arcade.create_triangles_filled_with_colors

arcade.**create_triangles_filled_with_colors**(*point_list*, *color_list*) → *arcade.buffered_draw_commands.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.

7.3.21 arcade.get_rectangle_points

arcade.**get_rectangle_points**(*center_x: float*, *center_y: float*, *width: float*, *height: float*, *tilt_angle: float = 0*) → Sequence[Union[Tuple[float, float], List[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** (*float*) –
- **center_y** (*float*) –
- **width** (*float*) –
- **height** (*float*) –
- **tilt_angle** (*float*) –

Returns: PointList

7.4 Drawing - Utility

7.4.1 arcade.color_from_hex_string

`arcade.color_from_hex_string`(*code: str*) → Union[Tuple[int, int, int, int], List[int]]

Make a color from a hex code (3, 4, 6 or 8 characters of hex, normally with a hashtag)

7.4.2 arcade.float_to_byte_color

`arcade.float_to_byte_color`(*color: Union[Tuple[float, float, float, float], Tuple[float, float, float]]*) → Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]]

Converts a float colors to a byte color. This works for 3 of 4-component colors.

7.4.3 arcade.get_four_byte_color

`arcade.get_four_byte_color`(*color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]]*) → Union[Tuple[int, int, int, int], List[int]]

Given a RGB list, it will return RGBA. Given a RGBA list, it will return the same RGBA.

Parameters **color** (*Color*) – Three or four byte tuple

Returns return: Four byte RGBA tuple

7.4.4 arcade.get_four_float_color

`arcade.get_four_float_color`(*color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]]*) → Tuple[float, float, float, float]

Given a 3 or 4 RGB/RGBA color where each color goes 0-255, this returns a RGBA tuple where each item is a scaled float from 0 to 1.

Parameters **color** (*Color*) – Three or four byte tuple

Returns Four floats as a RGBA tuple

7.4.5 arcade.get_points_for_thick_line

`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 think line must be two triangles that make up a rectangle. This calculates those points.

7.4.6 arcade.get_three_float_color

arcade.**get_three_float_color**(*color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]]*) → Tuple[float, float, float]

Given a 3 or 4 RGB/RGBA color where each color goes 0-255, this returns a RGBA tuple where each item is a scaled float from 0 to 1.

Parameters **color** (*Color*) – Three or four byte tuple

Returns Three floats as a RGB tuple

7.4.7 arcade.make_transparent_color

arcade.**make_transparent_color**(*color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]], transparency: float*)

Given a RGB color, along with an alpha, returns a RGBA color tuple.

Parameters

- **color** (*Color*) – Three or four byte RGBA color
- **transparency** (*float*) – Transparency

7.4.8 arcade.uint24_to_three_byte_color

arcade.**uint24_to_three_byte_color**(*color: int*) → Union[Tuple[int, int, int], List[int]]

Given an int between 0 and 16777215, return a RGB color tuple.

Parameters **color** (*int*) – 3 byte int

7.4.9 arcade.uint32_to_four_byte_color

arcade.**uint32_to_four_byte_color**(*color: int*) → Union[Tuple[int, int, int, int], List[int]]

Given an int between 0 and 4294967295, return a RGBA color tuple.

Parameters **color** (*int*) – 4 byte int

7.5 Sprites

7.5.1 arcade.AnimatedTimeBasedSprite

class arcade.**AnimatedTimeBasedSprite**(*filename: Optional[str] = None, scale: float = 1, image_x: float = 0, image_y: float = 0, image_width: float = 0, image_height: float = 0, center_x: float = 0, center_y: float = 0, _repeat_count_x=1, _repeat_count_y=1*)

Sprite for platformer games that supports animations. These can be automatically created by the Tiled Map Editor.

update_animation(*delta_time: float = 0.016666666666666666*)

Logic for selecting the proper texture to use.

7.5.2 arcade.AnimatedWalkingSprite

class arcade.**AnimatedWalkingSprite**(*scale: float = 1, image_x: float = 0, image_y: float = 0, center_x: float = 0, center_y: float = 0*)

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 - Add Character Animations, and Better Keyboard Control*.

update_animation(*delta_time: float = 0.016666666666666666*)

Logic for selecting the proper texture to use.

7.5.3 arcade.AnimationKeyframe

class arcade.**AnimationKeyframe**(*tile_id: int, duration: int, texture: arcade.texture.Texture*)

Used in animated sprites.

7.5.4 arcade.PyMunk

class arcade.**PyMunk**

Object used to hold pymunk info for a sprite.

7.5.5 arcade.Sprite

class arcade.**Sprite**(*filename: Optional[str] = None, scale: float = 1, image_x: float = 0, image_y: float = 0, image_width: float = 0, image_height: float = 0, center_x: float = 0, center_y: float = 0, repeat_count_x: int = 1, repeat_count_y: int = 1, flipped_horizontally: bool = False, flipped_vertically: bool = False, flipped_diagonally: bool = False, hit_box_algorithm: Optional[str] = 'Simple', hit_box_detail: float = 4.5, texture: Optional[arcade.texture.Texture] = None, angle: float = 0*)

Class that represents a 'sprite' on-screen. Most games center around sprites. For examples on how to use this class, see: https://api.arcade.academy/en/latest/examples/index.html#sprites

Parameters

- **filename** (*str*) – Filename of an image that represents the sprite.
- **scale** (*float*) – Scale the image up or down. Scale of 1.0 is none.
- **image_x** (*float*) – X offset to sprite within sprite sheet.
- **image_y** (*float*) – Y offset to sprite within sprite sheet.
- **image_width** (*float*) – Width of the sprite
- **image_height** (*float*) – Height of the sprite
- **center_x** (*float*) – Location of the sprite
- **center_y** (*float*) – Location of the sprite
- **flipped_horizontally** (*bool*) – Mirror the sprite image. Flip left/right across vertical axis.

- **flipped_vertically** (*bool*) – Flip the image up/down across the horizontal axis.
- **flipped_diagonally** (*bool*) – Transpose the image, flip it across the diagonal.
- **hit_box_algorithm** (*str*) – One of None, 'None', 'Simple' or 'Detailed'. Defaults to 'Simple'. Use 'Simple' for the *PhysicsEngineSimple*, *PhysicsEnginePlatformer* and 'Detailed' for the *PymunkPhysicsEngine*.
- **texture** (Texture) – Specify the texture directly.
- **angle** (*float*) – The initial rotation of the sprite in degrees

This will ignore all hit box and image size arguments.

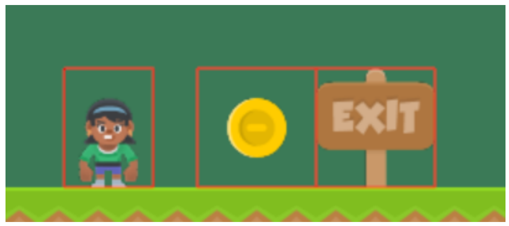


Fig. 1: hit_box_algorithm = "None"

Fig. 2: hit_box_algorithm = "Simple"

Fig. 3: hit_box_algorithm = "Detailed"

Parameters **hit_box_detail** (*float*) – Float, defaults to 4.5. Used with 'Detailed' to hit box

Attributes:

alpha Transparency of sprite. 0 is invisible, 255 is opaque.

angle Rotation angle in degrees. Sprites rotate counter-clock-wise.

radians Rotation angle in radians. Sprites rotate counter-clock-wise.

bottom Set/query the sprite location by using the bottom coordinate. This will be the 'y' of the bottom of the sprite.

boundary_left Used in movement. Left boundary of moving sprite.

boundary_right Used in movement. Right boundary of moving sprite.

boundary_top Used in movement. Top boundary of moving sprite.

boundary_bottom Used in movement. Bottom boundary of moving sprite.

center_x X location of the center of the sprite

center_y Y location of the center of the sprite

change_x Movement vector, in the x direction.

change_y Movement vector, in the y direction.

change_angle Change in rotation.

color Color tint the sprite

collision_radius Used as a fast-check to see if this item is close enough to another item. If this check works, we do a slower more accurate check. You probably don't want to use this field. Instead, set points in the hit box.

cur_texture_index Index of current texture being used.

guid Unique identifier for the sprite. Useful when debugging.

height Height of the sprite.

force Force being applied to the sprite. Useful when used with Pymunk for physics.

hit_box Points, in relation to the center of the sprite, that are used for collision detection. Arcade defaults to creating a hit box via the 'simple' hit box algorithm that encompass the image. If you are creating a ramp or making better hit-boxes, you can custom-set these.

left Set/query the sprite location by using the left coordinate. This will be the 'x' of the left of the sprite.

position A list with the (x, y) of where the sprite is.

right Set/query the sprite location by using the right coordinate. This will be the 'y=x' of the right of the sprite.

sprite_lists List of all the sprite lists this sprite is part of.

texture `arcade.Texture` class with the current texture. Setting a new texture does **not** update the hit box of the sprite. This can be done with `my_sprite.hit_box = my_sprite.texture.hit_box_points`. New textures will be centered on the current center_x/center_y.

textures List of textures associated with this sprite.

top Set/query the sprite location by using the top coordinate. This will be the 'y' of the top of the sprite.

scale Scale the image up or down. Scale of 1.0 is original size, 0.5 is 1/2 height and width.

velocity Change in x, y expressed as a list. (0, 0) would be not moving.

width Width of the sprite

It is common to over-ride the *update* method and provide mechanics on movement or other sprite updates.

add_spatial_hashes()

Add spatial hashes for this sprite in all the sprite lists it is part of.

property alpha: int

Return the alpha associated with the sprite.

property angle: float

Get the angle of the sprite's rotation.

append_texture(*texture*: arcade.texture.Texture)

Appends a new texture to the list of textures that can be applied to this sprite.

Parameters **texture** (`arcade.Texture`) – Texture to add to the list of available textures

property bottom: float

Return the y coordinate of the bottom of the sprite.

property center_x: float

Get the center x coordinate of the sprite.

property center_y: float

Get the center y coordinate of the sprite.

property change_x: float

Get the velocity in the x plane of the sprite.

property change_y: float

Get the velocity in the y plane of the sprite.

clear_spatial_hashes()

Search the sprite lists this sprite is a part of, and remove it from any spatial hashes it is a part of.

collides_with_list(*sprite_list*: SpriteList) → list

Check if current sprite is overlapping with any other sprite in a list

Parameters **sprite_list** (`SpriteList`) – SpriteList to check against

Returns SpriteList of all overlapping Sprites from the original SpriteList

Return type *SpriteList*

collides_with_point(*point*: *Union[Tuple[float, float], List[float]]*) → bool

Check if point is within the current sprite.

Parameters **point** (*Point*) – Point to check.

Returns True if the point is contained within the sprite's boundary.

Return type bool

collides_with_sprite(*other*: arcade.sprite.Sprite) → bool

Will check if a sprite is overlapping (colliding) another Sprite.

Parameters **other** (`Sprite`) – the other sprite to check against.

Returns True or False, whether or not they are overlapping.

Return type bool

property collision_radius: float

Get the collision radius.

Note: Final collision checking is done via geometry that was set in get_points/set_points. These points are used in the check_for_collision function. This collision_radius variable is used as a "pre-check." We do a super-fast check with collision_radius and see if the sprites are close. If they are, then we look at the geometry and figure if they really are colliding.

property color: Union[Tuple[int, int, int], List[int]]

Return the RGB color associated with the sprite.

draw(**, filter=None, pixelated=None, blend_function=None*)

Draw the sprite.

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_box(*color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]] = (0, 0, 0), line_thickness: float = 1*)

Draw a sprite's hit-box.

The 'hit box' drawing is cached, so if you change the color/line thickness later, it won't take.

Parameters

- **color** – Color of box
- **line_thickness** – How thick the box should be

face_point(*point: Union[Tuple[float, float], List[float]]*)

Face the sprite towards a point. Assumes sprite image is facing upwards.

Parameters **point** (*Point*) – Point to face towards.

forward(*speed: float = 1.0*)

Adjusts a Sprite's movement vector forward. This method does not actually move the sprite, just takes the current change_x/change_y and adjusts it by the speed given.

Parameters **speed** – speed factor

get_adjusted_hit_box() → Sequence[Union[Tuple[float, float], List[float]]]

Get the points that make up the hit box for the rect that makes up the sprite, including rotation and scaling.

get_hit_box() → Sequence[Union[Tuple[float, float], List[float]]]

Use the hit_box property to get or set a sprite's hit box. Hit boxes are specified assuming the sprite's center is at (0, 0). Specify hit boxes like:

```
mySprite.hit_box = [[-10, -10], [10, -10], [10, 10]]
```

Specify a hit box unadjusted for translation, rotation, or scale. You can get an adjusted hit box with `arcade.Sprite.get_adjusted_hit_box`.

property height: float

Get the height in pixels of the sprite.

kill()

Alias of *remove_from_sprite_lists*

property left: float

Return the x coordinate of the left-side of the sprite's hit box.

on_update(*delta_time: float = 0.016666666666666666*)

Update the sprite. Similar to update, but also takes a delta-time.

property position: Union[Tuple[float, float], List[float]]

Get the center x and y coordinates of the sprite.

Returns: (center_x, center_y)

property properties: Dict[str, Any]

Get or set custom sprite properties.

Return type Dict[str, Any]

property pymunk: *arcade.sprite.PyMunk*

Get or set the Pymunk property objects. This is used by the pymunk physics engine.

pymunk_moved(*physics_engine, dx, dy, d_angle*)

Called by the pymunk physics engine if this sprite moves.

property radians: float

Converts the degrees representation of self.angle into radians. :return: float

register_physics_engine(*physics_engine*)

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.

register_sprite_list(*new_list:* SpriteList)

Register this sprite as belonging to a list. We will automatically remove ourselves from the the list when kill() is called.

remove_from_sprite_lists()

Remove the sprite from all sprite lists.

rescale_relative_to_point(*point: Union[Tuple[float, float], List[float]], factor: float*) → None

Rescale the sprite relative to a different point than its center.

reverse(*speed: float = 1.0*)

Adjusts a Sprite's movement vector backwards. This method does not actually move the sprite, just takes the current change_x/change_y and adjusts it by the speed given.

Parameters speed – speed factor

property right: float

Return the x coordinate of the right-side of the sprite's hit box.

property scale: float

Get the scale of the sprite.

set_hit_box(*points: Sequence[Union[Tuple[float, float], List[float]]]*)

Set a sprite's hit box. Hit box should be relative to a sprite's center, and with a scale of 1.0. Points will be scaled with get_adjusted_hit_box.

set_position(*center_x: float, center_y: float*)

Set a sprite's position

Parameters

- **center_x** (*float*) – New x position of sprite
- **center_y** (*float*) – New y position of sprite

set_texture(*texture_no: int*)

Sets texture by texture id. Should be renamed because it takes a number rather than a texture, but keeping this for backwards compatibility.

stop()

Stop the Sprite's motion.

strafe(*speed: float = 1.0*)

Adjusts a Sprite's movement vector sideways. This method does not actually move the sprite, just takes the current change_x/change_y and adjusts it by the speed given.

Parameters speed – speed factor

property top: float

Return the y coordinate of the top of the sprite.

turn_left(*theta: float = 90.0*)

Rotate the sprite left by the passed number of degrees.

Parameters theta – change in angle, in degrees

turn_right(*theta: float = 90.0*)

Rotate the sprite right by the passed number of degrees.

Parameters theta – change in angle, in degrees

update()

Update the sprite.

update_animation(*delta_time: float = 0.016666666666666666*)

Override this to add code that will change what image is shown, so the sprite can be animated.

Parameters delta_time (*float*) – Time since last update.

property visible: bool

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
```

Return type bool

property width: float

Get the width of the sprite.

7.5.6 arcade.SpriteCircle

class arcade.**SpriteCircle**(*radius: int, color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]], soft: bool = False*)

This sprite is just an elliptical sprite of one solid color. No need to use an image file.

Parameters

- **radius** (*float*) – Radius of the circle
- **color** (*Color*) – Color of the circle
- **soft** (*bool*) – If True, will add a alpha gradient

7.5.7 arcade.SpriteSolidColor

class arcade.**SpriteSolidColor**(*width: int, height: int, color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]]*)

This sprite is just a rectangular sprite of one solid color. No need to use an image file.

Parameters

- **width** (*int*) – Width of the sprite
- **height** (*int*) – Height of the sprite
- **color** (*Color*) – Color of the sprite

7.5.8 arcade.get_distance_between_sprites

arcade.**get_distance_between_sprites**(*sprite1: arcade.sprite.Sprite, sprite2: arcade.sprite.Sprite*) → float

Returns the distance between the center of two given sprites

Parameters

- **sprite1** (Sprite) – Sprite one
- **sprite2** (Sprite) – Sprite two

Returns Distance

Return type float

7.5.9 arcade.load_animated_gif

`arcade.load_animated_gif(resource_name)`

Given an animated gif, return a AnimatedTimeBasedSprite.

Support for transparency in animated gifs in Python is lacking. There are a lot of older animated gifs that are saved weird. The end result is that the often the first frame of an animated gif is the only frame that we correctly get the transparency on. Until the Pillow library better handles this, loading animated gifs will be pretty buggy.

7.6 Sprite Lists

7.6.1 arcade.SpriteList

class `arcade.SpriteList`(*use_spatial_hash=None*, *spatial_hash_cell_size=128*, *is_static=False*, *atlas: TextureAtlas = None*, *capacity: int = 100*, *lazy: bool = False*, *visible: bool = True*)

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 spriteslist 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** (*bool*) – 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** (*int*) – The cell size of the spatial hash (default: 128)
- **is_static** (*bool*) – DEPRECATED. This parameter has no effect.
- **atlas** (`TextureAtlas`) – (Advanced) The texture atlas for this sprite list. If no atlas is supplied the global/default one will be used.
- **capacity** (*int*) – (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** (*bool*) – (Advanced) Enabling lazy spritelists ensures no internal OpenGL resources are created until the first draw call or `initialize()` is called. This can be useful when making spritelists in threads because only the main thread is allowed to interact with OpenGL.
- **visible** (*bool*) – Setting this to False will cause the SpriteList to not be drawn. When draw is called, the method will just return without drawing.

property alpha: int

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.

property alpha_normalized: float

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.

Return type float

append(*sprite: arcade.sprite_list.sprite_list._SpriteType*)

Add a new sprite to the list.

Parameters **sprite** (Sprite) – Sprite to add to the list.

property atlas: ***TextureAtlas***

Get the texture atlas for this sprite list

property buffer_angles: ***arcade.gl.buffer.Buffer***

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`.

property buffer_colors: ***arcade.gl.buffer.Buffer***

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`.

property buffer_indices: ***arcade.gl.buffer.Buffer***

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.

property buffer_positions: ***arcade.gl.buffer.Buffer***

Get the internal OpenGL position buffer for this spritelist.

The buffer contains 32 bit float values with x and y positions. These are the center postions for each sprite.

This buffer is attached to the *geometry* instance with name `in_pos`.

property buffer_sizes: ***arcade.gl.buffer.Buffer***

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`.

property buffer_textures: ***arcade.gl.buffer.Buffer***

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.

property center: Tuple[float, float]

Get the mean center coordinates of all sprites in 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.

property color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]]

Get or set the spritelist color. This will affect all sprites in the list. Individual sprites can also be assigned a color. These colors are converted into floating point colors (0.0 -> 1.0) and multiplied together.

The final color of the sprite is:

```
texture_color * sprite_color * spritelist_color
```

Return type Color

property color_normalized: Tuple[float, float, float, float]

Get or set the spritelist color in normalized form (0.0 -> 1.0 floats). This property works the same as *color*.

disable_spatial_hashing() → None

Turn off spatial hashing.

draw(**, filter=None, pixelated=None, blend_function=None*)

Draw this list of sprites.

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: Union[Tuple[int, int, int], List[int], 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=128*)

Turn on spatial hashing.

extend(*sprites: Union[list, arcade.sprite_list.sprite_list.SpriteList]*)

Extends the current list with the given list

Parameters **sprites** (*list*) – list of Sprites to add to the list

property geometry: ***arcade.gl.vertex_array.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;
```

index(*sprite: arcade.sprite.Sprite*) → int

Return the index of a sprite in the spritelist

Parameters **sprite** (Sprite) – Sprite to find and return the index of

Return type int

initialize()

Create the internal OpenGL resources. This can be done if the sprite list is lazy or was created before the window / context. The initialization will happen on the first draw if this method is not called. This is acceptable for most people, but this method gives you the ability to pre-initialize to potentially void initial stalls during rendering.

Calling this otherwise will have no effect. Calling this method in another thread will result in an OpenGL error.

insert(*index: int, sprite: arcade.sprite_list.sprite_list._SpriteType*)

Inserts a sprite at a given index.

Parameters

- **index** (*int*) – The index at which to insert
- **sprite** (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** (*float*) – Amount to change all x values by
- **change_y** (*float*) – 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*) → *arcade.sprite.Sprite*

Pop off the last sprite, or the given index, from the list

Parameters **index** (*int*) – 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** (*array*) – List of textures.

remove(*sprite: arcade.sprite_list.sprite_list._SpriteType*)

Remove a specific sprite from the list. :param Sprite 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=None, 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** (*bool*) – If set to `True` the sprites will be sorted in reverse

swap(*index_1: int, index_2: int*)

Swap two sprites by index :param int index_1: Item index to swap :param int index_2: Item index to swap

update() → None

Call the update() method on each sprite in the list.

update_angle(*sprite: arcade.sprite.Sprite*)

Called by the Sprite class to update the angle in this sprite. Necessary for batch drawing of items.

Parameters **sprite** (Sprite) – Sprite to update.

update_animation(*delta_time: float = 0.016666666666666666*)

Call the update_animation in every sprite in the sprite list.

update_color(*sprite: arcade.sprite.Sprite*) → None

Called by the Sprite class to update position, angle, size and color of the specified sprite. Necessary for batch drawing of items.

Parameters **sprite** (Sprite) – Sprite to update.

update_height(*sprite: arcade.sprite.Sprite*)

Called by the Sprite class to update the size/scale in this sprite. Necessary for batch drawing of items.

Parameters **sprite** (Sprite) – Sprite to update.

update_location(*sprite: arcade.sprite.Sprite*)

Called by the Sprite class to update the location in this sprite. Necessary for batch drawing of items.

Parameters **sprite** (Sprite) – Sprite to update.

update_position(*sprite: arcade.sprite.Sprite*) → None

Called when setting initial position of a sprite when added or inserted into the SpriteList.

`update_location` should be called to move them once the sprites are in the list.

Parameters **sprite** (Sprite) – Sprite to update.

update_size(*sprite: arcade.sprite.Sprite*) → None

Called by the Sprite class to update the size/scale in this sprite. Necessary for batch drawing of items.

Parameters **sprite** (Sprite) – Sprite to update.

update_texture(*sprite*) → None

Make sure we update the texture for this sprite for the next batch drawing

update_width(*sprite: arcade.sprite.Sprite*)

Called by the Sprite class to update the size/scale in this sprite. Necessary for batch drawing of items.

Parameters **sprite** (Sprite) – Sprite to update.

property use_spatial_hash: bool

Boolean variable that controls if this sprite list is using a spatial hash. If spatial hashing is turned on, it takes longer to move a sprite, and less time to see if that sprite is colliding with another sprite.

property visible: bool

Get or set the visible flag for this spritelist. If visible is `False` the `draw()` has no effect.

Return type bool

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).

7.6.2 arcade.check_for_collision

arcade.**check_for_collision**(*sprite1: arcade.sprite.Sprite*, *sprite2: arcade.sprite.Sprite*) → bool

Check for a collision between two sprites.

Parameters

- **sprite1** – First sprite
- **sprite2** – Second sprite

Returns True or False depending if the sprites intersect.

Return type bool

7.6.3 arcade.check_for_collision_with_list

arcade.**check_for_collision_with_list**(*sprite: arcade.sprite.Sprite*, *sprite_list: arcade.sprite_list.sprite_list.SpriteList*, *method=0*) → List[*arcade.sprite.Sprite*]

Check for a collision between a sprite, and a list of sprites.

Parameters

- **sprite** (Sprite) – Sprite to check
- **sprite_list** (SpriteList) – SpriteList to check against
- **method** (*int*) – 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.

Return type list

7.6.4 arcade.check_for_collision_with_lists

arcade.**check_for_collision_with_lists**(*sprite: arcade.sprite.Sprite*, *sprite_lists: Iterable[arcade.sprite_list.sprite_list.SpriteList]*, *method=1*) → List[*arcade.sprite.Sprite*]

Check for a collision between a Sprite, and a list of SpriteLists.

Parameters

- **sprite** (Sprite) – Sprite to check
- **sprite_lists** (*List[SpriteList]*) – SpriteLists to check against
- **method** (*int*) – 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.

Return type list

7.6.5 arcade.get_closest_sprite

`arcade.get_closest_sprite`(*sprite: arcade.sprite.Sprite*, *sprite_list: arcade.sprite_list.sprite_list.SpriteList*) → Optional[Tuple[*arcade.sprite.Sprite*, float]]

Given a Sprite and SpriteList, returns the closest sprite, and its distance.

Parameters

- **sprite** (`Sprite`) – Target sprite
- **sprite_list** (`SpriteList`) – 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`.

Return type Optional[Tuple[*Sprite*, float]]

7.6.6 arcade.get_sprites_at_exact_point

`arcade.get_sprites_at_exact_point`(*point: Union[Tuple[float, float], List[float]]*, *sprite_list: arcade.sprite_list.sprite_list.SpriteList*) → List[*arcade.sprite.Sprite*]

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`*) – Point to check
- **sprite_list** (`SpriteList`) – SpriteList to check against

Returns List of sprites colliding, or an empty list.

Return type list

7.6.7 arcade.get_sprites_at_point

`arcade.get_sprites_at_point`(*point: Union[Tuple[float, float], List[float]]*, *sprite_list: arcade.sprite_list.sprite_list.SpriteList*) → List[*arcade.sprite.Sprite*]

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`*) – Point to check
- **sprite_list** (`SpriteList`) – SpriteList to check against

Returns List of sprites colliding, or an empty list.

Return type list

7.7 Sprite Scenes

7.7.1 arcade.Scene

class arcade.Scene

Class that represents a *scene* object. Most games will use Scenes to render their Sprites. For examples on how to use this class, see: https://api.arcade.academy/en/latest/tutorials/views/index.html

Attributes:

sprite_lists A list of *SpriteList* objects. The order of this list is the order in which they will be drawn.

name_mapping A dictionary of *SpriteList* objects. This contains the same lists as the *sprite_lists* attribute, but is a mapping of them by name. This is not necessarily in the same order as the *sprite_lists* attribute.

add_sprite(*name: str*, *sprite:* arcade.sprite.Sprite) → None

Add a Sprite to a SpriteList in the Scene with the specified name.

If the desired SpriteList does not exist, it will automatically be created and added to the Scene. This will default the SpriteList to be added to the end of the draw order, and created with no extra options like using spatial hashing.

If you need more control over where the SpriteList goes or need it to use Spatial Hash, then the SpriteList should be added separately and then have the Sprites added.

Parameters

- **name** (*str*) – The name of the *SpriteList* to add to or create.
- **sprite** (Sprite) – The *Sprite* to add.

add_sprite_list(*name: str*, *use_spatial_hash: bool = False*, *sprite_list: Optional[*arcade.sprite_list.sprite_list.SpriteList*] = None*) → None

Add a SpriteList to the scene with the specified name.

This will add a new SpriteList to the scene at the end of the draw order.

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** (*str*) – The name to give the SpriteList.
- **use_spatial_hash** (*bool*) – Wether or not to use spatial hash if creating a new SpriteList.
- **sprite_list** (SpriteList) – The SpriteList to add, optional.

add_sprite_list_after(*name: str*, *after: str*, *use_spatial_hash: bool = False*, *sprite_list: Optional[*arcade.sprite_list.sprite_list.SpriteList*] = None*) → None

Add a SpriteList to the scene with the specified name after a specific SpriteList.

This will add a new SpriteList to the scene after the specified SpriteList in the draw order.

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** (*str*) – The name to give the SpriteList.

- **after** (*str*) – The name of the SpriteList to place this one after.
- **use_spatial_hash** (*bool*) – Wether or not to use spatial hash if creating a new SpriteList.
- **sprite_list** (SpriteList) – The SpriteList to add, optional.

add_sprite_list_before(*name: str*, *before: str*, *use_spatial_hash: bool = False*, *sprite_list: Optional[arcade.sprite_list.sprite_list.SpriteList] = None*) → None

Add a SpriteList to the scene with the specified name before a specific SpriteList.

This will add a new SpriteList to the scene before the specified SpriteList in the draw order.

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** (*str*) – The name to give the SpriteList.
- **before** (*str*) – The name of the SpriteList to place this one before.
- **use_spatial_hash** (*bool*) – Wether or not to use spatial hash if creating a new SpriteList.
- **sprite_list** (SpriteList) – The SpriteList to add, optional.

draw(*names: Optional[List[str]] = None*, ***kwargs*) → None

Draw the Scene.

If *names* parameter is provided then only the specified SpriteLists will be drawn. They will be drawn in the order that the names in the list were arranged. If *names* is not provided, then every SpriteList in the scene will be drawn according the order of the main sprite_lists attribute of the Scene.

Parameters

- **names** (*Optional[List[str]]*) – A list of names of SpriteLists to draw.
- **filter** – Optional parameter to set OpenGL filter, such as *gl.GL_NEAREST* to avoid smoothing.
- **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: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]] = (0, 0, 0, 255)*, *line_thickness: float = 1*, *names: Optional[List[str]] = None*) → None

Draw hitboxes for all sprites in the scene.

If *names* parameter is provided then only the specified SpriteLists will be drawn. They will be drawn in the order that the names in the list were arranged. If *names* is not provided, then every SpriteList in the scene will be drawn according to the order of the main sprite_lists attribute of the Scene.

classmethod from_tilemap(*tilemap: arcade.tilemap.tilemap.TileMap*) → *arcade.scene.Scene*

Create a new Scene from a *TileMap* object.

This will look at all the SpriteLists in a TileMap object and create a Scene with them. This will automatically keep SpriteLists in the same order as they are defined in the TileMap class, which is the order that they are defined within Tiled.

Parameters tilemap (TileMap) – The *TileMap* object to create the scene from.

get_sprite_list(*name: str*) → *arcade.sprite_list.sprite_list.SpriteList*

Helper function to retrieve a *SpriteList* by name.

The name mapping can be accessed directly, this is just here for ease of use.

Parameters **name** (*str*) – The name of the *SpriteList* to retrieve.

move_sprite_list_after(*name: str*, *after: str*) → None

Move a given SpriteList in the scene to after another given SpriteList.

This will adjust the render order so that the SpriteList specified by *name* is placed after the one specified by *after*.

Parameters

- **name** (*str*) – The name of the SpriteList to move.
- **after** (*str*) – The name of the SpriteList to place it after.

move_sprite_list_before(*name: str*, *before: str*) → None

Move a given SpriteList in the scene to before another given SpriteList.

This will adjust the render order so that the SpriteList specified by *name* is placed before the one specified by *before*.

Parameters

- **name** (*str*) – The name of the SpriteList to move.
- **before** (*str*) – The name of the SpriteList to place it before.

on_update(*delta_time: float = 0.016666666666666666*, *names: Optional[List[str]] = None*) → None

Used to call on_update of SpriteLists contained in the scene. Similar to update() but allows passing a delta_time variable.

If *names* parameter is provided then only the specified spritelists will be updated. If *names* is not provided, then every SpriteList in the scene will have on_update called.

Parameters

- **delta_time** (*float*) – Time since last update.
- **names** (*Optional[List[str]]*) – A list of names of SpriteLists to update.

remove_sprite_list_by_name(*name: str*) → None

Remove a SpriteList by it's name.

This function serves to completely remove the SpriteList from the Scene.

Parameters **name** (*str*) – The name of the SpriteList to remove.

update(*names: Optional[List[str]] = None*) → None

Used to update SpriteLists contained in the scene.

If *names* parameter is provided then only the specified spritelists will be updated. If *names* is not provided, then every SpriteList in the scene will be updated.

Parameters **names** (*Optional[List[str]]*) – A list of names of SpriteLists to update.

update_animation(*delta_time: float*, *names: Optional[List[str]] = None*) → None

Used to update the animation of SpriteLists contained in the scene.

If *names* parameter is provided then only the specified spritelists will be updated. If *names* is not provided, then every SpriteList in the scene will be updated.

Parameters

- **delta_time** (*float*) – The delta time for the update.
- **names** (*Optional[List[str]]*) – A list of names of SpriteLists to update.

7.8 Camera

7.8.1 arcade.Camera

class arcade.**Camera**(*viewport_width: int = 0*, *viewport_height: int = 0*, *window: Optional[arcade.application.Window] = None*)

The Camera class is used for controlling the visible viewport. 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_width** (*int*) – Width of the viewport. If not set the window width will be used.
- **viewport_height** (*int*) – Height of the viewport. If not set the window height will be used.
- **window** (Window) – Window to associate with this camera, if working with a multi-window program.

move(*vector: pyglet.math.Vec2*)

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: pyglet.math.Vec2*, *speed: float = 1.0*)

Sets the goal position of the camera.

The camera will lerp towards this position based on the provided speed, updating its position everytime the use() function is called.

Parameters

- **vector** (*Vec2*) – Vector to move the camera towards.
- **speed** (*Vec2*) – How fast to move the camera, 1.0 is instant, 0.1 moves slowly

resize(*viewport_width: int*, *viewport_height: int*)

Resize the camera's viewport. Call this when the window resizes.

Parameters

- **viewport_width** (*int*) – Width of the viewport
- **viewport_height** (*int*) – Height of the viewport

set_projection()

Update the projection matrix of the camera. This creates an orthogonal projection based on the viewport size of the camera.

shake(*velocity: pyglet.math.Vec2*, *speed: float = 1.5*, *damping: float = 0.9*)

Add a camera shake.

Parameters

- **velocity** (*Vec2*) – Vector to start moving the camera. Needs to be a pyglet.math.Vec2.
- **speed** (*float*) – How fast to shake
- **damping** (*float*) – How fast to stop shaking

update()

Update the camera's viewport to the current settings.

use()

Select this camera for use. Do this right before you draw.

zoom(*change: float*)

Zoom the camera in or out. Or not. This will currently raise an error TODO implement

7.9 Text - Image/Pillow based

7.9.1 arcade.create_text_image

arcade.**create_text_image**(*text: str, text_color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]], font_size: float = 12, width: int = 0, align: str = 'left', valign: str = 'top', font_name: Union[str, Tuple[str, ...]] = ('calibri', 'arial'), background_color: Optional[Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]]] = None, height: int = 0*) → PIL.Image.Image

Create a PIL.Image containing text.

> **Warning:** This method can be fairly slow. We recommend creating images on initialization or infrequently later on.

Parameters

- **text** (*str*) – The text to render to the image
- **text_color** (*Color*) – Color of the text
- **font_size** (*float*) – Size of the font
- **width** (*int*) – The width of the image in pixels
- **align** (*str*) – "left" or "right" aligned
- **valign** (*str*) – "top" or "bottom" aligned
- **font_name** (*str*) – The font to use
- **background_color** (*Color*) – The background color of the image
- **height** (*int*) – the height of the image in pixels

7.9.2 arcade.create_text_sprite

arcade.**create_text_sprite**(*text: str, start_x: float, start_y: float, color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]], font_size: float = 12, width: int = 0, align: str = 'left', font_name: Union[str, Tuple[str, ...]] = ('calibri', 'arial'), bold: bool = False, italic: bool = False, anchor_x: str = 'left', anchor_y: str = 'baseline', rotation: float = 0*) → *arcade.sprite.Sprite*

Creates a sprite with a text texture using `create_text_image()`.

Internally this works by creating an image, and using the Pillow library to draw the text to it. Then use that image to create a sprite. We cache the sprite (so we don't have to recreate over and over, which is slow) and use it to draw text to the screen.

This implementation does not support bold/italic like the older Pyglet-based implementation of draw_text. However if you specify the 'italic' or 'bold' version of the font via the font name, you will get that font. Just the booleans do not work.

Parameters

- **text** (*str*) – Text to draw
- **start_x** (*float*) – x coordinate of the lower-left point to start drawing text
- **start_y** (*float*) – y coordinate of the lower-left point to start drawing text
- **color** (*Color*) – Color of the text
- **font_size** (*float*) – Size of the text
- **width** (*float*) – Width of the text-box for the text to go into. Used with alignment.
- **align** (*str*) – Align left, right, center
- **font_name** (*Union[str, Tuple[str, ...]]*) – Font name, or list of font names in order of preference
- **bold** (*bool*) – Bold the font (currently unsupported)
- **italic** (*bool*) – Italicize the font (currently unsupported)
- **anchor_x** (*str*) – Anchor the font location, defaults to 'left'
- **anchor_y** (*str*) – Anchor the font location, defaults to 'baseline'
- **rotation** (*float*) – Rotate the text

7.10 Text - Pyglet/Glyph based

7.10.1 arcade.Text

`class arcade.Text`(*text: str, start_x: float, start_y: float, color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]] = (255, 255, 255), font_size: float = 12, width: int = 0, align: str = 'left', font_name: Union[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*)

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** (*str*) – Initial text to display. Can be an empty string

- **start_x** (*float*) – x position to align the text's anchor point with
- **start_y** (*float*) – y position to align the text's anchor point with
- **color** (*Color*) – Color of the text as a tuple or list of 3 (RGB) or 4 (RGBA) integers
- **font_size** (*float*) – Size of the text in points
- **width** (*float*) – A width limit in pixels
- **align** (*str*) – 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** (*bool*) – Whether to draw the text as bold
- **italic** (*bool*) – Whether to draw the text as italic
- **anchor_x** (*str*) – How to calculate the anchor point's x coordinate. Options: "left", "center", or "right"
- **anchor_y** (*str*) – How to calculate the anchor point's y coordinate. Options: "top", "bottom", "center", or "baseline".
- **multiline** (*bool*) – Requires width to be set; enables word wrap rather than clipping
- **rotation** (*float*) – 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. 4: 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:

property anchor_x: str

Get or set the horizontal anchor.

Options: "left", "center", or "right"

property anchor_y: str

Get or set the vertical anchor.

Options : "top", "bottom", "center", or "baseline"

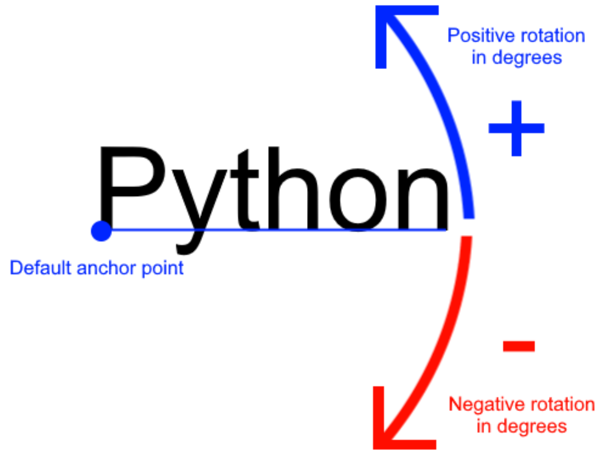


Fig. 5: Rotation around the default anchor (`anchor_y="baseline"` and `anchor_x="left"`)

property bold: bool

Get or set bold state of this label

property bottom: int

Pixel location of the bottom content border.

property color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]]

Get or set the text color for this label

property content_height: int

Get the pixel height of the text content.

property content_size: Tuple[int, int]

Get the pixel width and height of the text contents.

property content_width: int

Get the pixel width of the text contents

draw() → None

Draw this label to the screen at its current `x` and `y` position.

draw_debug(*anchor_color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]] = (255, 0, 0), background_color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]] = (0, 0, 139), outline_color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]] = (255, 255, 255)*) → None

Draw test with debug geometry showing the content area, outline and the anchor point.

Parameters

- **anchor_color** (*Color*) – Color of the anchor point
- **background_color** (*Color*) – Color the content background
- **outline_color** (*Color*) – Color of the content outline

property font_name: Union[str, Tuple[str, ...]]

Get or set the font name(s) for this label

property font_size: float

Get or set the font size of the label

property height: int

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*

property italic: bool

Get or set the italic state of this label

property left: int

Pixel location of the left content border.

property multiline: bool

Get or set the multiline flag of this label.

property position: Union[Tuple[float, float], List[float]]

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.

property right: int

Pixel location of the right content border.

property size

Get the size of this label

property text: str

Get or set the current text string to display.

THe value assigned will be converted to a string.

This is an alias for *value*

property top: int

Pixel location of the top content border.

property value: str

Get or set the current text string to display.

THe value assigned will be converted to a string.

property width: int

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*

property x: float

Get or set the x position of the label

property y: float

Get or set the y position of the label

7.10.2 arcade.draw_text

`arcade.draw_text(text: Any, start_x: float, start_y: float, color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]] = (255, 255, 255), font_size: float = 12, width: int = 0, align: str = 'left', font_name: Union[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)`

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** (*Any*) – Text to display. The object passed in will be converted to a string
- **start_x** (*float*) – x position to align the text's anchor point with
- **start_y** (*float*) – y position to align the text's anchor point with
- **color** (*Color*) – Color of the text as a tuple or list of 3 (RGB) or 4 (RGBA) integers
- **font_size** (*float*) – Size of the text in points
- **width** (*float*) – A width limit in pixels
- **align** (*str*) – 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** (*bool*) – Whether to draw the text as bold
- **italic** (*bool*) – Whether to draw the text as italic
- **anchor_x** (*str*) – How to calculate the anchor point's x coordinate
- **anchor_y** (*str*) – How to calculate the anchor point's y coordinate
- **multiline** (*bool*) – Requires width to be set; enables word wrap rather than clipping
- **rotation** (*float*) – 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. 6: 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 at 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 at 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:

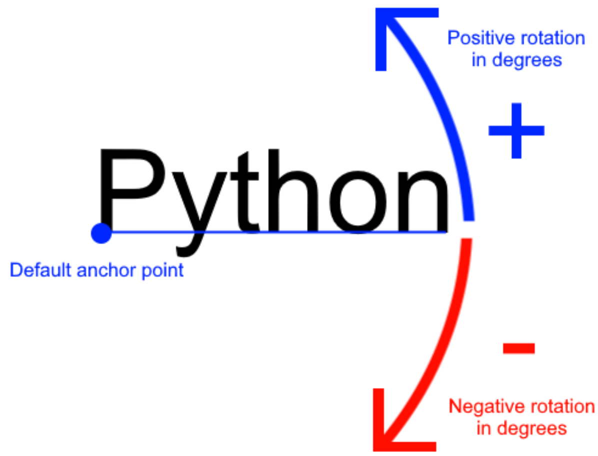


Fig. 7: Rotation around the default anchor point (`anchor_y="baseline"` and `anchor_x="left"`)

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 of the steps above can be repeated each time a call is made rather than fully cached as with the class.

7.10.3 arcade.load_font

`arcade.load_font`(*path: Union[str, pathlib.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 actually 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 **font_name** –

Raises `FileNotFoundError` – if the font specified wasn't found

Returns

7.11 Tiled Map Reader

7.11.1 arcade.tilemap.TileMap

class arcade.tilemap.**TileMap**(*map_file: Union[str, pathlib.Path] = '', scaling: float = 1.0, layer_options: Optional[Dict[str, Dict[str, Any]]] = None, use_spatial_hash: Optional[bool] = None, hit_box_algorithm: str = 'Simple', hit_box_detail: float = 4.5, tiled_map: Optional[pytiled_parser.tiled_map.TiledMap] = None, offset: pyglet.math.Vec2 = Vec2(0, 0)*)

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** (*float*) – Global scaling to apply to all Sprites.
- **layer_options** (*Dict[str, Dict[str, Any]]*) – Extra parameters for each layer.
- **use_spatial_hash** (*Optional[bool]*) – 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** (*str*) – One of 'None', 'Simple' or 'Detailed'.
- **hit_box_detail** (*float*) – Float, defaults to 4.5. Used with 'Detailed' to hit box.
- **tiled_map** (*pytiled_parser.TiledMap*) – 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** (*pyglet.math.Vec2*) – 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.

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 - A string for the hit box algorithm to use for the Sprite's in this layer. hit_box_detail - A float specifying the level of detail for each Sprite's hitbox 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

For example:

code-block:

```
layer_options = {
    "Platforms": {
        "use_spatial_hash": True,
```

(continues on next page)

(continued from previous page)

```
        "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.

Attributes:

tiled_map 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 The width of the map in tiles. This is the number of tiles, not pixels.

height The height of the map in tiles. This is the number of tiles, not pixels.

tile_width The width in pixels of each tile.

tile_height The height in pixels of each tile.

background_color The background color of the map.

scaling A global scaling value to be applied to all Sprites in the map.

sprite_lists A dictionary mapping SpriteLists to their layer names. This is used for all tile layers of the map.

object_lists A dictionary mapping TiledObjects to their layer names. This is used for all object layers of the map.

offset A tuple containing the X and Y position offset values.

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** (*float*) – The X Coordinate to convert
- **y** (*float*) – The Y Coordinate to convert

7.11.2 arcade.tilemap.load_tilemap

`arcade.tilemap.load_tilemap(map_file: Union[str, pathlib.Path], scaling: float = 1.0, layer_options: Optional[Dict[str, Dict[str, Any]]] = None, use_spatial_hash: Optional[bool] = None, hit_box_algorithm: str = 'Simple', hit_box_detail: float = 4.5, offset: pyglet.math.Vec2 = Vec2(0, 0)) → arcade.tilemap.tilemap.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** (*float*) – The global scaling to apply to all Sprite's within the map.
- **use_spatial_hash** (*Optional[bool]*) – 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** (*str*) – One of 'None', 'Simple' or 'Detailed'.
- **hit_box_detail** (*float*) – Float, defaults to 4.5. Used with 'Detailed' to hit box.
- **layer_options** (*Dict[str, Dict[str, Any]]*) – Layer specific options for the map.
- **offset** (*pyglet.math.Vec2*) – 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.

7.11.3 arcade.tilemap.read_tmx

`arcade.tilemap.read_tmx(map_file: Union[str, pathlib.Path]) → pytiled_parser.tiled_map.TiledMap`

Deprecated function to raise a warning that it has been removed.

Exists to provide info for outdated code bases.

7.12 Texture Management

7.12.1 arcade.Texture

`class arcade.Texture(name: str, image: Optional[PIL.Image.Image] = None, hit_box_algorithm: Optional[str] = 'Simple', hit_box_detail: float = 4.5)`

Class that represents a texture. Usually created by the `load_texture` or `load_textures` commands.

Parameters

- **name** (*str*) – Name of texture. Used for caching, so must be unique for each texture.
- **image** (*PIL.Image.Image*) – Image to use as a texture.
- **hit_box_algorithm** (*str*) – One of None, 'None', 'Simple' or 'Detailed'. Defaults to 'Simple'. Use 'Simple' for the `PhysicsEngineSimple`, `PhysicsEnginePlatformer` and 'Detailed' for the `PymunkPhysicsEngine`.
- **hit_box_detail** (*float*) – Float, defaults to 4.5. Used with 'Detailed' to hit box

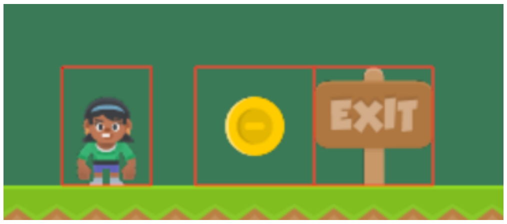


Fig. 8: hit_box_algorithm = “None”

Fig. 9: hit_box_algorithm = “Simple”

Fig. 10: hit_box_algorithm = “Detailed”

Attributes:

name Unique name of the texture. Used by load_textures for caching. If you are manually creating a texture, you can just set this to whatever.

image A `PIL.Image.Image` object.

width Width of the texture in pixels.

height Height of the texture in pixels.

size Tuple containing (width, height)

hit_box_points The computed hit box of the texture

classmethod create_empty(*name: str*, *size: Tuple[int, int]*) → *arcade.texture.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** (*str*) – The unique name for this texture
- **size** (*Tuple[int, int]*) – 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: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]]*) → *arcade.texture.Texture*

Create a texture completely filled with the passed color.

The hit box of the returned Texture will be set to a rectangle with the dimensions in `size` because all pixels are filled with the same color.

Parameters

- **name** (*str*) – The unique name for this texture
- **size** (*Tuple[int, int]*) – The xy size of the internal image
- **color** (*Color*) – the color to fill the texture with

This function has multiple uses, including:

- A helper for pre-blending backgrounds into terrain tiles
- Fillers to stand in for state-specific textures
- Quick filler assets for various proofs of concept

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 filled texture with the same background color, 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 filled Texture instance you would like to create. This can be especially helpful if you are creating multiple large Textures.

draw_scaled(*center_x: float*, *center_y: float*, *scale: float = 1.0*, *angle: float = 0*, *alpha: int = 255*)

Draw the texture.

Parameters

- **center_x** (*float*) – X location of where to draw the texture.
- **center_y** (*float*) – Y location of where to draw the texture.
- **scale** (*float*) – Scale to draw rectangle. Defaults to 1.
- **angle** (*float*) – Angle to rotate the texture by.
- **alpha** (*int*) – The transparency of the texture *(0-255)*.

draw_sized(*center_x: float*, *center_y: float*, *width: float*, *height: float*, *angle: float = 0*, *alpha: int = 255*)

Draw a texture with a specific width and height.

property height: int

Height of the texture in pixels.

property size: Tuple[int, int]

Width and height as a tuple

property width: int

Width of the texture in pixels.

7.12.2 arcade.cleanup_texture_cache

arcade.**cleanup_texture_cache**()

This cleans up the cache of textures. Useful when running unit tests so that the next test starts clean.

7.12.3 arcade.load_spritesheet

arcade.**load_spritesheet**(*file_name: Union[str, pathlib.Path]*, *sprite_width: int*, *sprite_height: int*, *columns: int*, *count: int*, *margin: int = 0*, *hit_box_algorithm: Optional[str] = 'Simple'*, *hit_box_detail: float = 4.5*) → List[*arcade.texture.Texture*]

Parameters

- **file_name** (*str*) – Name of the file to that holds the texture.
- **sprite_width** (*int*) – Width of the sprites in pixels
- **sprite_height** (*int*) – Height of the sprites in pixels
- **columns** (*int*) – Number of tiles wide the image is.
- **count** (*int*) – Number of tiles in the image.
- **margin** (*int*) – Margin between images
- **hit_box_algorithm** (*str*) – One of None, 'None', 'Simple' (default) or 'Detailed'.
- **hit_box_detail** (*float*) – Float, defaults to 4.5. Used with 'Detailed' to hit box

Returns List List of `Texture` objects.

7.12.4 arcade.load_texture

arcade.**load_texture**(*file_name: Union[str, pathlib.Path]*, *x: float = 0*, *y: float = 0*, *width: float = 0*, *height: float = 0*, *flipped_horizontally: bool = False*, *flipped_vertically: bool = False*, *flipped_diagonally: bool = False*, *can_cache: bool = True*, *mirrored: Optional[bool] = None*, *hit_box_algorithm: Optional[str] = 'Simple'*, *hit_box_detail: float = 4.5*) → *arcade.texture.Texture*

Load an image from disk and create a texture.

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** (*str*) – Name of the file to that holds the texture.
- **x** (*float*) – X position of the crop area of the texture.
- **y** (*float*) – Y position of the crop area of the texture.
- **width** (*float*) – Width of the crop area of the texture.
- **height** (*float*) – Height of the crop area of the texture.
- **flipped_horizontally** (*bool*) – Mirror the sprite image. Flip left/right across vertical axis.
- **flipped_vertically** (*bool*) – Flip the image up/down across the horizontal axis.
- **flipped_diagonally** (*bool*) – Transpose the image, flip it across the diagonal.
- **can_cache** (*bool*) – If a texture has already been loaded, load_texture will return the same texture in order to save time. Sometimes this is not desirable, as resizing a cached texture

will cause all other textures to resize with it. Setting can_cache to false will prevent this issue at the experience of additional resources.

- **mirrored** (*bool*) – Deprecated.
- **hit_box_algorithm** (*str*) – One of None, 'None', 'Simple' or 'Detailed'. Defaults to 'Simple'. Use 'Simple' for the *PhysicsEngineSimple*, *PhysicsEnginePlatformer* and 'Detailed' for the *PymunkPhysicsEngine*.

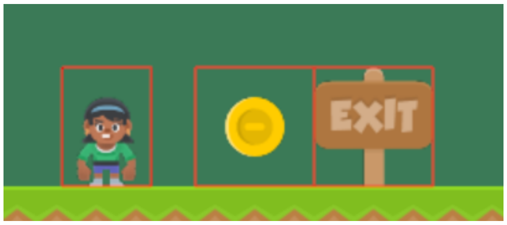


Fig. 11: hit_box_algorithm = "None"

Fig. 12: hit_box_algorithm = "Simple"

Fig. 13: hit_box_algorithm = "Detailed"

- **hit_box_detail** (*float*) – Float, defaults to 4.5. Used with 'Detailed' to hit box

Returns New *Texture* object.

Raises ValueError

7.12.5 arcade.load_texture_pair

`arcade.load_texture_pair`(*filename*, *hit_box_algorithm: str = 'Simple'*)

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.

7.12.6 arcade.load_textures

`arcade.load_textures`(*file_name: Union[str, pathlib.Path]*, *image_location_list: Union[Tuple[Union[Tuple[float, float, float, float], List[float]], ...], List[Union[Tuple[float, float, float, float], List[float]]]]*, *mirrored: bool = False*, *flipped: bool = False*, *hit_box_algorithm: Optional[str] = 'Simple'*, *hit_box_detail: float = 4.5*) → List[*arcade.texture.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** (*str*) – Name of the file.
- **image_location_list** (*List*) – List of image sub-locations. Each rectangle should be a *List* of four floats: *[x, y, width, height]*.
- **mirrored** (*bool*) – If set to *True*, the image is mirrored left to right.
- **flipped** (*bool*) – If set to *True*, the image is flipped upside down.
- **hit_box_algorithm** (*str*) – One of None, 'None', 'Simple' (default) or 'Detailed'.
- **hit_box_detail** (*float*) – Float, defaults to 4.5. Used with 'Detailed' to hit box

Returns List of `Texture`'s.

Raises ValueError

7.12.7 arcade.make_circle_texture

`arcade.make_circle_texture`(*diameter: int*, *color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]]*, *name: Optional[str] = None*) → *arcade.texture.Texture*

Return a Texture of a circle with the given diameter and color.

Parameters

- **diameter** (*int*) – Diameter of the circle and dimensions of the square `Texture` returned.
- **color** (*Color*) – Color of the circle.
- **name** (*str*) – Custom or pre-chosen name for this texture

Returns New `Texture` object.

7.12.8 arcade.make_soft_circle_texture

arcade.**make_soft_circle_texture**(*diameter: int*, *color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]]*, *center_alpha: int = 255*, *outer_alpha: int = 0*, *name: Optional[str] = None*) → *arcade.texture.Texture*

Return a `Texture` of a circle with the given diameter and color, fading out at its edges.

Parameters

- **diameter** (*int*) – Diameter of the circle and dimensions of the square `Texture` returned.
- **color** (*Color*) – Color of the circle.
- **center_alpha** (*int*) – Alpha value of the circle at its center.
- **outer_alpha** (*int*) – Alpha value of the circle at its edges.
- **name** (*str*) – Custom or pre-chosen name for this texture

Returns New `Texture` object.

Return type *arcade.Texture*

7.12.9 arcade.make_soft_square_texture

arcade.**make_soft_square_texture**(*size: int*, *color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]]*, *center_alpha: int = 255*, *outer_alpha: int = 0*, *name: Optional[str] = None*) → *arcade.texture.Texture*

Return a `Texture` of a square with the given diameter and color, fading out at its edges.

Parameters

- **size** (*int*) – Diameter of the square and dimensions of the square Texture returned.
- **color** (*Color*) – Color of the square.
- **center_alpha** (*int*) – Alpha value of the square at its center.
- **outer_alpha** (*int*) – Alpha value of the square at its edges.
- **name** (*str*) – Custom or pre-chosen name for this texture

Returns New `Texture` object.

7.12.10 arcade.trim_image

arcade.**trim_image**(*image: PIL.Image.Image*) → PIL.Image.Image

Crops the extra whitespace out of an image.

Returns New `PIL.Image.Image` object.

7.13 Texture Atlas

7.13.1 arcade.AtlasRegion

class arcade.**AtlasRegion**(*atlas: TextureAtlas*, *texture: Texture*, *x: int*, *y: int*, *width: int*, *height: int*)

Stores information about where a texture is located

Parameters

- **atlas** (*str*) – The atlas this region belongs to
- **texture** (*str*) – The arcade texture
- **x** (*int*) – The x position of the texture
- **y** (*int*) – The y position of the texture
- **width** (*int*) – The width of the texture in pixels
- **height** (*int*) – The height of the texture in pixels

verify_image_size()

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.

7.13.2 arcade.TextureAtlas

class arcade.**TextureAtlas**(*size: Tuple[int, int]*, ***, *border: int = 1*, *textures: Sequence[Texture] = None*, *auto_resize: bool = True*, *ctx: ArcadeContext = None*)

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.

Parameters

- **size** (*Tuple[int, int]*) – The width and height of the atlas in pixels
- **border** (*int*) – Currently no effect; Should always be 1 to avoid textures bleeding
- **textures** (*Sequence[arcade.Texture]*) – The texture for this atlas
- **auto_resize** (*bool*) – Automatically resize the atlas when full
- **ctx** (*Context*) – The context for this atlas (will use window context if left empty)

add(*texture: Texture*) → Tuple[int, *arcade.texture_atlas.AtlasRegion*]

Add a texture to the atlas.

Parameters **texture** (*Texture*) – The texture to add

Returns texture_id, AtlasRegion tuple

allocate(*texture: Texture*) → Tuple[int, int, int, *arcade.texture_atlas.AtlasRegion*]

Attempts to allocate space for a texture in the atlas. This doesn't write the texture to the atlas texture itself. It only allocates space.

Returns The x, y texture_id, TextureRegion

property auto_resize: bool

Get or set the auto resize flag for the atlas. If enabled the atlas will resize itself when full.

Return type bool

property border: int

The texture border in pixels

Return type int

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[Texture]*) – Sequence of textures
- **border** –

Returns An estimated minimum size as a (width, height) tuple

clear(*texture_ids: bool = True, texture: bool = True*) → None

Clear and reset the texture atlas. Note that also clearing "texture_ids" makes the atlas lose track of the old texture ids. This means the sprite list must be rebuild from scratch.

Parameters

- **texture_ids** (*bool*) – Clear the assigned texture ids
- **texture** (*bool*) – Clear the contents of the atlas texture itself

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** (*Sequence[Texture]*) – A sequence of textures (list, set, tuple, generator etc.)
- **border** (*int*) – The border for the atlas in pixels (space between each texture)

property fbo: *arcade.gl.framebuffer.Framebuffer*

The framebuffer object for this atlas

get_region_info(*name: str*) → *arcade.texture_atlas.AtlasRegion*

Get the region info for a texture

Returns The AtlasRegion for the given texture name

get_texture_id(*name: str*) → int

Get the uv slot for a texture name

Returns The texture id for the given texture name

has_texture(*texture: Texture*) → bool

Check if a texture is already in the atlas

property height: int

The height of the texture atlas in pixels

Return type int

property max_height: int

The maximum height of the atlas in pixels

Return type int

property max_size: Tuple[int, int]

The maximum size of the atlas in pixels (x, y)

Return type Tuple[int,int]

property max_width: int

The maximum width of the atlas in pixels

Return type int

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 remove the image from the underlying texture. To physically remove the data you need to `rebuild()`.

Parameters **texture** (`Texture`) – The texture to remove

render_into(*texture:* Texture, *projection: Tuple[float, float, float, float] = 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:

```
with atlas.render_into(texture):
    # Draw commands here

# Specify projection
with atlas.render_into(texture, projection=(0, 100, 0, 100))
    # Draw geometry
```

Parameters

- **texture** (`Texture`) – The texture area to render into
- **projection** (*Tuple[float, float, float, float]*) – 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** (*Tuple[int,int]*) – The new size

save(*path: str, 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.

Parameters

- **path** (*str*) – The path to save the atlas on disk
- **flip** (*bool*) – Flip the image horizontally
- **components** (*int*) – 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

Parameters

- **flip** (*bool*) – Flip the image horizontally
- **components** (*int*) – Number of components. (3 = RGB, 4 = RGBA)
- **draw_borders** (*bool*) – Draw region borders into image
- **color** – RGB color of the borders

property size: Tuple[int, int]

The width and height of the texture atlas in pixels

Return type Tuple[int,int]

property texture: GLTexture

The atlas texture

Return type *Texture*

to_image(*flip: bool = False, components: int = 4, draw_borders: bool = False, border_color: Tuple[int, int, int] = (255, 0, 0)*) → PIL.Image.Image

Convert the atlas to a Pillow image

Parameters

- **flip** (*bool*) – Flip the image horizontally
- **components** (*int*) – Number of components. (3 = RGB, 4 = RGBA)
- **draw_borders** (*bool*) – 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** (`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** (*int*) – The texture unit to bind the uv texture

property uv_texture: GLTexture

Texture coordinate texture.

Return type *Texture*

property width: int

The width of the texture atlas in pixels

Return type int

write_image(*image: PIL.Image.Image, x: int, y: int*) → None

Write a PIL image to the atlas in a specific region.

Parameters

- **image** (*PIL.Image.Image*) – The pillow image
- **x** (*int*) – The x position to write the texture
- **y** (*int*) – The y position to write the texture

write_texture(*texture: Texture, x: int, y: int*)

Writes an arcade texture to a subsection of the texture atlas

7.14 Performance Information

7.14.1 arcade.PerfGraph

class arcade.**PerfGraph**(*width, height, graph_data: str = 'FPS', update_rate: float = 0.1, background_color=(0, 0, 0), data_line_color=(255, 255, 255), axis_color=(155, 135, 12), grid_color=(155, 135, 12), font_color=(255, 255, 255), font_size=10*)

Create a graph showing performance statistics.

update_graph(*delta_time: float*)

Update the graph.

7.14.2 arcade.clear_timings

arcade.**clear_timings**()

Clear the dispatch event timing table created after `arcade.enable_timings()` is called.

7.14.3 arcade.disable_timings

arcade.**disable_timings**()

Turn off the collection of timing information started by `arcade.enable_timings()`.

7.14.4 arcade.enable_timings

arcade.**enable_timings**(*max_history: int = 100*)

Enable the saving of performance information.

7.14.5 arcade.get_fps

arcade.**get_fps**(*frame_count: int = 60*) → float

Get the current FPS. `arcade.enable_timings()` must be called before getting the FPS.

Parameters **frame_count** (*int*) – How many frames to look at to get FPS. So 30, would give you average FPS over the last 30 frames.

7.14.6 arcade.get_timings

arcade.**get_timings**() → Dict

Get a table with the dispatch event timings.

7.14.7 arcade.print_timings

arcade.**print_timings**()

This prints to stdout a table of the most recent dispatched events and their average time.

The table looks something like:

```
Event          Count Average Time
-------------- ----- ------------
update            60       0.0553
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
```

7.14.8 arcade.timings_enabled

`arcade.timings_enabled()`

Return true if timings are enabled, false otherwise. See `arcade.enable_timings()`.

7.15 Physics Engines

7.15.1 arcade.PymunkException

`class arcade.PymunkException`

7.15.2 arcade.PymunkPhysicsEngine

`class arcade.PymunkPhysicsEngine`(*gravity=(0, 0)*, *damping: float = 1.0*, *maximum_incline_on_ground: float = 0.708*)

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: Optional[Callable] = None*, *pre_handler: Optional[Callable] = None*, *post_handler: Optional[Callable] = None*, *separate_handler: Optional[Callable] = None*)

Add code to handle collisions between objects.

`add_sprite`(*sprite: arcade.sprite.Sprite*, *mass: float = 1*, *friction: float = 0.2*, *elasticity: Optional[float] = None*, *moment_of_inertia: Optional[float] = None*, *body_type: int = 0*, *damping: Optional[float] = None*, *gravity: Optional[Union[pymunk.vec2d.Vec2d, Tuple[float, float], pyglet.math.Vec2]] = None*, *max_velocity: Optional[int] = None*, *max_horizontal_velocity: Optional[int] = None*, *max_vertical_velocity: Optional[int] = None*, *radius: float = 0*, *collision_type: Optional[str] = 'default'*, *moment_of_intertia: Optional[float] = None*, *moment: Optional[float] = None*)

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
- **max_vertical_velocity** – maximum velocity on the y axis
- **radius** –
- **collision_type** –
- **moment_of_intertia** – Deprecated alias of moment_of_inertia compatible with a typo introduced in 2.6.2
- **moment** – Deprecated alias of moment_of_inertia compatible with versions <= 2.6.1

add_sprite_list(*sprite_list*, *mass: float = 1*, *friction: float = 0.2*, *elasticity: Optional[float] = None*, *moment_of_intertia: Optional[float] = None*, *body_type: int = 0*, *damping: Optional[float] = None*, *collision_type: Optional[str] = None*)

Add all sprites in a sprite list to the physics engine.

apply_force(*sprite: arcade.sprite.Sprite*, *force: Tuple[float, float]*)

Apply force to a Sprite.

apply_impulse(*sprite: arcade.sprite.Sprite*, *impulse: Tuple[float, float]*)

Apply an impulse force on a sprite

apply_opposite_running_force(*sprite: arcade.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: arcade.sprite.Sprite*)

See if the player is on the ground. Used to see if we can jump.

get_physics_object(*sprite: arcade.sprite.Sprite*) → *arcade.pymunk_physics_engine.PymunkPhysicsObject*

Get the shape/body for a sprite.

get_sprite_for_shape(*shape: Optional[pymunk.shapes.Shape]*) → Optional[*arcade.sprite.Sprite*]

Given a shape, what sprite is associated with it?

get_sprites_from_arbiter(*arbiter: pymunk.arbiter.Arbiter*) → Tuple[Optional[*arcade.sprite.Sprite*], Optional[*arcade.sprite.Sprite*]]

Given a collision arbiter, return the sprites associated with the collision.

is_on_ground(*sprite: arcade.sprite.Sprite*) → bool

Return true of sprite is on top of something.

remove_sprite(*sprite: arcade.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: arcade.sprite.Sprite*, *friction: float*)

Apply force to a Sprite.

set_horizontal_velocity(*sprite: arcade.sprite.Sprite*, *velocity: float*)

Set a sprite's velocity

set_position(*sprite: arcade.sprite.Sprite*, *position: Union[pymunk.vec2d.Vec2d, Tuple[float, float]]*)

Apply an impulse force on a sprite

set_velocity(*sprite: arcade.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** (*float*) – Time to move the simulation forward. Keep this value constant, do not use varying values for each step.
- **resync_sprites** (*bool*) – 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.

7.15.3 arcade.PymunkPhysicsObject

class arcade.**PymunkPhysicsObject**(*body: Optional[pymunk.body.Body] = None*, *shape: Optional[pymunk.shapes.Shape] = None*)

Object that holds pymunk body/shape for a sprite.

7.15.4 arcade.PhysicsEnginePlatformer

class arcade.**PhysicsEnginePlatformer**(*player_sprite: arcade.sprite.Sprite*, *platforms: Optional[Union[arcade.sprite_list.sprite_list.SpriteList, Iterable[arcade.sprite_list.sprite_list.SpriteList]]] = None*, *gravity_constant: float = 0.5*, *ladders: Optional[Union[arcade.sprite_list.sprite_list.SpriteList, Iterable[arcade.sprite_list.sprite_list.SpriteList]]] = None*, *walls: Optional[Union[arcade.sprite_list.sprite_list.SpriteList, Iterable[arcade.sprite_list.sprite_list.SpriteList]]] = None*)

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** (`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** (*float*) – 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

Return type bool

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** (*int*) –

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.

7.15.5 arcade.PhysicsEngineSimple

class arcade.**PhysicsEngineSimple**(*player_sprite:* arcade.sprite.Sprite, *walls: Union*[arcade.sprite_list.sprite_list.SpriteList, *Iterable*[arcade.sprite_list.sprite_list.SpriteList]])

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** (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.

7.16 Misc Utility Functions

7.16.1 arcade.configure_logging

arcade.**configure_logging**(*level: Optional[int] = None*)

Set up basic logging. :param int level: The log level. Defaults to DEBUG.

7.16.2 arcade.generate_uuid_from_kwargs

arcade.**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.

7.16.3 arcade.lerp

arcade.**lerp**(*v1: float*, *v2: float*, *u: float*) → float

linearly interpolate between two values

7.16.4 arcade.lerp_vec

arcade.**lerp_vec**(*v1: Union[Tuple[float, float], List[float]]*, *v2: Union[Tuple[float, float], List[float]]*, *u: float*) → Union[Tuple[float, float], List[float]]

7.16.5 arcade.rand_angle_360_deg

arcade.**rand_angle_360_deg**()

Return a random angle in degrees.

7.16.6 arcade.rand_angle_spread_deg

`arcade.`**`rand_angle_spread_deg`**(*angle: float*, *half_angle_spread: float*) → float

7.16.7 arcade.rand_in_circle

`arcade.`**`rand_in_circle`**(*center: Union[Tuple[float, float], List[float]]*, *radius: 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

7.16.8 arcade.rand_in_rect

`arcade.`**`rand_in_rect`**(*bottom_left: Union[Tuple[float, float], List[float]]*, *width: float*, *height: float*) → Union[Tuple[float, float], List[float]]

7.16.9 arcade.rand_on_circle

`arcade.`**`rand_on_circle`**(*center: Union[Tuple[float, float], List[float]]*, *radius: float*) → Union[Tuple[float, float], List[float]]

Note: by passing a random value in for float, you can achieve what rand_in_circle() does

7.16.10 arcade.rand_on_line

`arcade.`**`rand_on_line`**(*pos1: Union[Tuple[float, float], List[float]]*, *pos2: Union[Tuple[float, float], List[float]]*) → Union[Tuple[float, float], List[float]]

Given two points defining a line, return a random point on that line.

7.16.11 arcade.rand_vec_magnitude

`arcade.`**`rand_vec_magnitude`**(*angle: float*, *lo_magnitude: float*, *hi_magnitude: float*) → Union[Tuple[float, float], List[float]]

7.16.12 arcade.rand_vec_spread_deg

`arcade.`**`rand_vec_spread_deg`**(*angle: float*, *half_angle_spread: float*, *length: float*) → Union[Tuple[float, float], List[float]]

7.17 Geometry Support

7.17.1 arcade.calculate_hit_box_points_detailed

`arcade.calculate_hit_box_points_detailed(image: PIL.Image.Image, hit_box_detail: float = 4.5) → Union[List[Union[Tuple[float, float], List[float]]], Tuple[Union[Tuple[float, float], List[float]], ...]]`

Given an RGBA image, this returns points that make up a hit box around it. Attempts to trim out transparent pixels.

Parameters

- **image** (*Image*) – Image get hit box from.
- **hit_box_detail** (*int*) – How detailed to make the hit box. There's a trade-off in number of points vs. accuracy.

Returns List of points

7.17.2 arcade.calculate_hit_box_points_simple

`arcade.calculate_hit_box_points_simple(image: PIL.Image.Image) → Union[Tuple[Union[Tuple[float, float], List[float]]], List]`

Given an RGBA image, this returns points that make up a hit box around it. Attempts to trim out transparent pixels.

Parameters **image** (*Image*) –

Returns List of points

7.17.3 arcade.are_polygons_intersecting

`arcade.are_polygons_intersecting(poly_a: Sequence[Union[Tuple[float, float], List[float]]], poly_b: Sequence[Union[Tuple[float, float], List[float]]]) → bool`

Return True if two polygons intersect.

Parameters

- **poly_a** (*PointList*) – List of points that define the first polygon.
- **poly_b** (*PointList*) – List of points that define the second polygon.

Returns True or false depending if polygons intersect

Rtype bool

7.17.4 arcade.is_point_in_polygon

arcade.**is_point_in_polygon**(*x*, *y*, *polygon_point_list*)

Use ray-tracing to see if point is inside a polygon

Args: x: y: polygon_point_list:

Returns: bool

7.17.5 arcade.EasingData

class arcade.**EasingData**(*start_period: float*, *cur_period: float*, *end_period: float*, *start_value: float*, *end_value: float*, *ease_function: Callable*)

Data class for holding information about easing.

7.17.6 arcade.ease_angle

arcade.**ease_angle**(*start_angle*, *end_angle*, ***, *time=None*, *rate=None*, *ease_function=<function linear>*)

Set up easing for angles.

7.17.7 arcade.ease_angle_update

arcade.**ease_angle_update**(*easing_data:* arcade.easing.EasingData, *delta_time: float*) → Tuple

Update angle easing.

7.17.8 arcade.ease_in

arcade.**ease_in**(*percent: float*) → float

Function for quadratic ease-in easing.

7.17.9 arcade.ease_in_back

arcade.**ease_in_back**(*percent: float*) → float

Function for ease_in easing which moves back before moving forward.

7.17.10 arcade.ease_in_out

arcade.**ease_in_out**(*percent: float*) → float

Function for quadratic easing in and out.

7.17.11 arcade.ease_in_out_sin

arcade.**ease_in_out_sin**(*percent: float*) → float

Function for easing in and out using a sin wave

7.17.12 arcade.ease_in_sin

arcade.**ease_in_sin**(*percent: float*) → float

Function for ease_in easing using a sin wave

7.17.13 arcade.ease_out

arcade.**ease_out**(*percent: float*) → float

Function for quadratic ease-out easing.

7.17.14 arcade.ease_out_back

arcade.**ease_out_back**(*percent: float*) → float

Function for ease_out easing which moves back before moving forward.

7.17.15 arcade.ease_out_bounce

arcade.**ease_out_bounce**(*percent: float*) → float

Function for a bouncy ease-out easing.

7.17.16 arcade.ease_out_elastic

arcade.**ease_out_elastic**(*percent: float*) → float

Function for elastic ease-out easing.

7.17.17 arcade.ease_out_sin

arcade.**ease_out_sin**(*percent: float*) → float

Function for ease_out easing using a sin wave

7.17.18 arcade.ease_position

arcade.**ease_position**(*start_position*, *end_position*, ***, *time=None*, *rate=None*, *ease_function=<function linear>*)

Get an easing position

7.17.19 arcade.ease_update

`arcade.ease_update`(*easing_data: arcade.easing.EasingData*, *delta_time: float*) → Tuple

Update easing between two values/

7.17.20 arcade.ease_value

`arcade.ease_value`(*start_value*, *end_value*, ***, *time=None*, *rate=None*, *ease_function=<function linear>*)

Get an easing value

7.17.21 arcade.easing

`arcade.easing`(*percent: float*, *easing_data: arcade.easing.EasingData*) → float

Function for calculating return value for easing, given percent and easing data.

7.17.22 arcade.linear

`arcade.linear`(*percent: float*) → float

Function for linear easing.

7.17.23 arcade.smoothstep

`arcade.smoothstep`(*percent: float*) → float

Function for smoothstep easing.

7.17.24 arcade.earclip

`arcade.earclip`(*polygon: Sequence[Union[Tuple[float, float], List[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

7.17.25 arcade.clamp

`arcade.clamp`(*a*, *low*, *high*)

Clamp a number between a range.

7.17.26 arcade.get_angle_degrees

arcade.**get_angle_degrees**(*x1: float*, *y1: float*, *x2: float*, *y2: float*) → float

Get the angle in degrees between two points.

Parameters

- **x1** (*float*) – x coordinate of the first point
- **y1** (*float*) – y coordinate of the first point
- **x2** (*float*) – x coordinate of the second point
- **y2** (*float*) – y coordinate of the second point

7.17.27 arcade.get_angle_radians

arcade.**get_angle_radians**(*x1: float*, *y1: float*, *x2: float*, *y2: float*) → float

Get the angle in radians between two points.

Parameters

- **x1** (*float*) – x coordinate of the first point
- **y1** (*float*) – y coordinate of the first point
- **x2** (*float*) – x coordinate of the second point
- **y2** (*float*) – y coordinate of the second point

7.17.28 arcade.get_distance

arcade.**get_distance**(*x1: float*, *y1: float*, *x2: float*, *y2: float*)

Get the distance between two points.

7.17.29 arcade.rotate_point

arcade.**rotate_point**(*x: float*, *y: float*, *cx: float*, *cy: float*, *angle_degrees: float*) → List[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

Return type (float, float)

7.18 Game Controller Support

7.18.1 arcade.get_game_controllers

`arcade.get_game_controllers()` → List[pyglet.input.base.Joystick]

Get a list of all the game controllers

Returns List of game controllers

7.18.2 arcade.get_joysticks

`arcade.get_joysticks()` → List[pyglet.input.base.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

7.19 Window and View

7.19.1 arcade.close_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).

7.19.2 arcade.create_orthogonal_projection

`arcade.create_orthogonal_projection`(*left: float*, *right: float*, *bottom: float*, *top: float*, *near: float = 1*, *far: float = - 1*) → pyglet.math.Mat4

Creates an orthogonal projection matrix. Used internally with the OpenGL shaders. It creates the same matrix as the deprecated/removed `glOrtho` OpenGL function.

Parameters

- **left** (*float*) – The left of the near plane relative to the plane's center.
- **right** (*float*) – The right of the near plane relative to the plane's center.
- **top** (*float*) – The top of the near plane relative to the plane's center.
- **bottom** (*float*) – The bottom of the near plane relative to the plane's center.
- **near** (*float*) – The distance of the near plane from the camera's origin. It is recommended that the near plane is set to 1.0 or above to avoid rendering issues at close range.
- **far** (*float*) – The distance of the far plane from the camera's origin.

Returns A projection matrix representing the specified orthogonal perspective.

Return type pyglet.math.Mat4

See also:

https://www.khronos.org/registry/OpenGL-Refpages/gl2.1/xhtml/glOrtho.xml

7.19.3 arcade.exit

`arcade.exit()`

Exits the application.

7.19.4 arcade.finish_render

`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`.

7.19.5 arcade.get_display_size

`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** (*int*) – The screen number

Returns Tuple containing the width and height of the screen

Return type tuple

7.19.6 arcade.get_projection

`arcade.get_projection() → pyglet.math.Mat4`

Returns the current projection matrix used by sprites and shapes in arcade.

This is a shortcut for ``window.ctx.projection_2d_matrix`.

Returns Projection matrix

Return type Mat4

7.19.7 arcade.get_scaling_factor

`arcade.get_scaling_factor(window: Window = None) → float`

Gets the scaling factor of the given Window. This is the ratio between the window and framebuffer size. If no window is supplied the currently active window will be used.

Parameters **window** (*Window*) – Handle to window we want to get scaling factor of.

Returns Scaling factor. E.g., 2.0 would indicate the framebuffer width and height being 2.0 times the window width and height. This means one "window pixel" is actual a 2 x 2 square of pixels in the framebuffer.

Return type float

7.19.8 arcade.get_viewport

`arcade.get_viewport()` → Tuple[float, float, float, float]

Get the current viewport settings.

Returns Tuple of floats, with `(left, right, bottom, top)`

7.19.9 arcade.get_window

`arcade.get_window()` → *Window*

Return a handle to the current window.

Returns Handle to the current window.

7.19.10 arcade.pause

`arcade.pause`(*seconds: numbers.Number*) → 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** (*float*) – Time interval to pause in seconds.

7.19.11 arcade.run

`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`.

7.19.12 arcade.schedule

`arcade.schedule`(*function_pointer: Callable*, *interval: numbers.Number*)

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:

```
def some_action(delta_time):
    print(delta_time)

# Call the function every second
arcade.schedule(some_action, 1)
# Unschedule
```

Parameters

- **function_pointer** (*Callable*) – Pointer to the function to be called.
- **interval** (*Number*) – Interval to call the function (float or integer)

7.19.13 arcade.set_background_color

arcade.**set_background_color**(*color: Union[Tuple[int, int, int], List[int], 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:

```
# 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 **color** (*Color*) – List of 3 or 4 values in RGB/RGBA format.

7.19.14 arcade.set_viewport

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 over-ride the `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** (*Number*) – Left-most (smallest) x value.
- **right** (*Number*) – Right-most (largest) x value.
- **bottom** (*Number*) – Bottom (smallest) y value.
- **top** (*Number*) – Top (largest) y value.

7.19.15 arcade.set_window

arcade.**set_window**(*window: Window*) → None

Set a handle to the current window.

Parameters **window** (Window) – Handle to the current window.

7.19.16 arcade.start_render

arcade.**start_render**() → None

Clears the window.

More practical alternatives to this function is *arcade.Window.clear()* or *arcade.View.clear()*.

7.19.17 arcade.unschedule

arcade.**unschedule**(*function_pointer: Callable*)

Unschedule a function being automatically called.

Example:

```
def some_action(delta_time):
    print(delta_time)

arcade.schedule(some_action, 1)
arcade.unschedule(some_action)
```

Parameters **function_pointer** (*Callable*) – Pointer to the function to be unscheduled.

7.19.18 arcade.NoOpenGLException

class arcade.**NoOpenGLException**

Exception when we can't get an OpenGL 3.3+ context

7.19.19 arcade.View

class arcade.**View**(*window: Optional[arcade.application.Window] = None*)

Support different views/screens in a window.

add_section(*section, at_index: Optional[int] = None*) → None

Adds a section to the view Section Manager. :param section: the section to add to this section manager :param at_index: inserts the section at that index. If None at the end

clear(*color: Optional[Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]]] = None, normalized: bool = False, viewport: Optional[Tuple[int, int, int, int]] = None*)

Clears the View's Window with the configured background color set through `arcade.Window.background_color`.

Parameters

- **color** (*Color*) – Optional color overriding the current background color
- **normalized** (*bool*) – If the color format is normalized (0.0 -> 1.0) or byte values
- **viewport** (*Tuple[int, int, int, int]*) – The viewport range to clear

property has_sections: bool

Return if the View has sections

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** (*int*) – Key that was hit
- **modifiers** (*int*) – Bitwise 'and' of all modifiers (shift, ctrl, num lock) pressed during this event. See *Modifiers*.

on_key_release(*_symbol: int, _modifiers: int*)

Override this function to add key release functionality.

Parameters

- **_symbol** (*int*) – Key that was hit
- **_modifiers** (*int*) – Bitwise 'and' of all modifiers (shift, ctrl, num lock) pressed 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** (*int*) – x position of mouse
- **y** (*int*) – y position of mouse
- **dx** (*int*) – Change in x since the last time this method was called
- **dy** (*int*) – Change in y since the last time this method was called
- **_buttons** (*int*) – Which button is pressed
- **_modifiers** (*int*) – Bitwise 'and' of all modifiers (shift, ctrl, num lock) pressed 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** (*int*) – x position of mouse
- **y** (*int*) – 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** (*int*) – x position of mouse
- **y** (*int*) – y position of mouse

on_mouse_motion(*x: int*, *y: int*, *dx: int*, *dy: int*)

Override this function to add mouse functionality.

Parameters

- **x** (*int*) – x position of mouse
- **y** (*int*) – y position of mouse
- **dx** (*int*) – Change in x since the last time this method was called
- **dy** (*int*) – 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** (*int*) – x position of the mouse
- **y** (*int*) – y position of the mouse
- **button** (*int*) – What button was hit. One of: arcade.MOUSE_BUTTON_LEFT, arcade.MOUSE_BUTTON_RIGHT, arcade.MOUSE_BUTTON_MIDDLE
- **modifiers** (*int*) – Bitwise 'and' of all modifiers (shift, ctrl, num lock) pressed 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** (*int*) – x position of mouse
- **y** (*int*) – y position of mouse
- **button** (*int*) – What button was hit. One of: arcade.MOUSE_BUTTON_LEFT, arcade.MOUSE_BUTTON_RIGHT, arcade.MOUSE_BUTTON_MIDDLE
- **modifiers** (*int*) – Bitwise 'and' of all modifiers (shift, ctrl, num lock) pressed 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** (*int*) – x position of mouse
- **y** (*int*) – y position of mouse
- **scroll_x** (*int*) – ammout of x pixels scrolled since last call
- **scroll_y** (*int*) – 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

update(*delta_time: float*)

To be overridden

7.19.20 arcade.Window

class arcade.Window(*width: int = 800, height: int = 600, title: Optional[str] = 'Arcade Window', fullscreen: bool = False, resizable: bool = False, update_rate: Optional[float] = 0.016666666666666666, antialiasing: bool = True, gl_version: Tuple[int, int] = (3, 3), screen: Optional[pyglet.canvas.base.Screen] = None, style: Optional[str] = None, visible: bool = True, vsync: bool = False, gc_mode: str = 'context_gc', center_window: bool = False, samples: int = 4, enable_polling: bool = True*)

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** (*int*) – Window width
- **height** (*int*) – Window height
- **title** (*str*) – Title (appears in title bar)
- **fullscreen** (*bool*) – Should this be full screen?
- **resizable** (*bool*) – Can the user resize the window?
- **update_rate** (*float*) – How frequently to update the window.
- **antialiasing** (*bool*) – Should OpenGL's anti-aliasing be enabled?
- **gl_version** (*Tuple[int,int]*) – What OpenGL version to request. This is `(3, 3)` by default and can be overridden when using more advanced OpenGL features.
- **visible** (*bool*) – Should the window be visible immediately
- **vsync** (*bool*) – Wait for vertical screen refresh before swapping buffer This can make animations and movement look smoother.
- **gc_mode** (*bool*) – Decides how OpenGL objects should be garbage collected ("context_gc" (default) or "auto")
- **center_window** (*bool*) – If true, will center the window.
- **samples** (*bool*) – Number of samples used in antialiasing (default 4). Usually this is 2, 4, 8 or 16.
- **enable_polling** (*bool*) – Enabled input polling capability. This makes the `keyboard` and `mouse` attributes available for use.

activate()

Activate this window.

property background_color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]]

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

# Specify RGB value directly (red)
window.background_color = 255, 0, 0
```

If the background color is an `RGB` value instead of `RGBA` we assume alpha value 255.

Type Color

center_window()

Center the window on the screen.

clear(*color: Optional[Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]]] = None, normalized: bool = False, viewport: Optional[Tuple[int, int, int, int]] = None*)

Clears the window with the configured background color set through *arcade.Window.background_color*.

Parameters

- **color** (*Color*) – Optional color overriding the current background color

- **normalized** (*bool*) – 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.

property ctx: *arcade.context.ArcadeContext*

The OpenGL context for this window.

Type *arcade.ArcadeContext*

property current_view: Optional[*arcade.application.View***]**

This property returns the current view being shown. To set a different view, call the `arcade.Window.show_view()` method.

Return type *arcade.View*

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.)

headless

bool: If this is a headless window

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*)

Override this function to add key press functionality.

Parameters

- **symbol** (*int*) – Key that was hit
- **modifiers** (*int*) – Bitwise 'and' of all modifiers (shift, ctrl, num lock) pressed during this event. See *Modifiers*.

on_key_release(*symbol: int*, *modifiers: int*)

Override this function to add key release functionality.

Parameters

- **symbol** (*int*) – Key that was hit
- **modifiers** (*int*) – Bitwise 'and' of all modifiers (shift, ctrl, num lock) pressed 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** (*int*) – x position of mouse
- **y** (*int*) – y position of mouse
- **dx** (*int*) – Change in x since the last time this method was called
- **dy** (*int*) – Change in y since the last time this method was called
- **buttons** (*int*) – Which button is pressed
- **modifiers** (*int*) – Bitwise 'and' of all modifiers (shift, ctrl, num lock) pressed 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** (*int*) –
- **y** (*int*) –

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** (*int*) –
- **y** (*int*) –

on_mouse_motion(*x: int*, *y: int*, *dx: int*, *dy: int*)

Override this function to add mouse functionality.

Parameters

- **x** (*int*) – x position of mouse

- **y** (*int*) – y position of mouse
- **dx** (*int*) – Change in x since the last time this method was called
- **dy** (*int*) – 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** (*int*) – x position of the mouse
- **y** (*int*) – y position of the mouse
- **button** (*int*) – What button was hit. One of: arcade.MOUSE_BUTTON_LEFT, arcade.MOUSE_BUTTON_RIGHT, arcade.MOUSE_BUTTON_MIDDLE
- **modifiers** (*int*) – Bitwise 'and' of all modifiers (shift, ctrl, num lock) pressed 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** (*int*) – x position of mouse
- **y** (*int*) – y position of mouse
- **button** (*int*) – What button was hit. One of: arcade.MOUSE_BUTTON_LEFT, arcade.MOUSE_BUTTON_RIGHT, arcade.MOUSE_BUTTON_MIDDLE
- **modifiers** (*int*) – Bitwise 'and' of all modifiers (shift, ctrl, num lock) pressed 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** (*int*) – x position of mouse
- **y** (*int*) – y position of mouse
- **scroll_x** (*int*) – ammout of x pixels scrolled since last call
- **scroll_y** (*int*) – ammout of y pixels scrolled since last call

on_resize(*width: float*, *height: float*)

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:

```
def on_resize(self, width: int, height: int):
    super().on_resize(width, height)
    # Add extra resize logic here
```

Parameters

- **width** (*int*) – New width
- **height** (*int*) – New height

on_update(*delta_time: float*)

Move everything. Perform collision checks. Do all the game logic here.

Parameters **delta_time** (`float`) – Time interval since the last time the function was called.

run()

Shortcut for `arcade.run()`.

For example:

```
MyWindow().run()
```

set_caption(*caption*)

Set the caption for the window.

set_exclusive_keyboard(*exclusive=True*)

Capture all keyboard input.

set_exclusive_mouse(*exclusive=True*)

Capture the mouse.

set_fullscreen(*fullscreen: bool = True, screen: Optional[arcade.application.Window] = None, mode: Optional[pyglet.canvas.base.ScreenMode] = None, width: Optional[float] = None, height: Optional[float] = None*)

Set if we are full screen or not.

Parameters

- **fullscreen** (`bool`) –
- **screen** – Which screen should we display on? See `get_screens()`
- **mode** (`pyglet.canvas.ScreenMode`) – 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** (`int`) –
- **height** (`int`) –

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** (`int`) – width in pixels.
- **height** (`int`) – 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** (`float`) – width in pixels.

- **height** (*float*) – height in pixels.

set_minimum_size(*width: int*, *height: int*)

Set smallest window size.

set_mouse_platform_visible(*platform_visible=None*)

This method is only exposed/overridden because it causes PyCharm to display a warning. This function is setting the platform specific mouse cursor visibility and would only be something an advanced user would care about.

See pyglet documentation for details.

set_mouse_visible(*visible: bool = True*)

If true, user can see the mouse cursor while it is over the window. Set false, the mouse is not visible. Default is true.

Parameters **visible** (*bool*) –

set_size(*width: int*, *height: int*)

Ignore the resizable flag and set the size

Parameters

- **width** (*int*) –
- **height** (*int*) –

set_update_rate(*rate: float*)

Set how often the screen should be updated. For example, self.set_update_rate(1 / 60) will set the update rate to 60 fps

Parameters **rate** (*float*) – 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** (*Number*) –
- **right** (*Number*) –
- **bottom** (*Number*) –
- **top** (*Number*) –

set_visible(*visible: bool = True*)

Set if the window is visible or not. Normally, a program's window is visible.

Parameters **visible** (*bool*) –

set_vsync(*vsync: bool*)

Set if we sync our draws to the monitors vertical sync rate.

show_view(*new_view: arcade.application.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*) – 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** (*int*) –

update(*delta_time: float*)

Move everything. For better consistency in naming, use `on_update` instead.

Parameters **delta_time** (*float*) – Time interval since the last time the function was called in seconds.

use()

Bind the window's framebuffer for rendering commands

7.19.21 arcade.get_screens

arcade.**get_screens**()

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.

Return type List

7.19.22 arcade.open_window

arcade.**open_window**(*width: int, height: int, window_title: Optional[str] = None, resizable: bool = False, antialiasing: bool = True*) → *arcade.application.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** (*Number*) – Width of the window.
- **height** (*Number*) – Height of the window.
- **window_title** (*str*) – Title of the window.
- **resizable** (*bool*) – Whether the window can be user-resizable.
- **antialiasing** (*bool*) – Smooth the graphics?

Returns Handle to window

Return type *Window*

7.19.23 arcade.Section

class arcade.**Section**(*left: int, bottom: int, width: int, height: int, *, name: Optional[str] = None, accept_keyboard_events: Union[bool, Iterable] = True, prevent_dispatch: Optional[Iterable] = None, prevent_dispatch_view: Optional[Iterable] = None, local_mouse_coordinates: bool = False, enabled: bool = True, modal: bool = False*)

A Section represents a rectangular portion of the viewport Events are dispatched to the section based on it's position on the screen.

property bottom: int

The bottom edge of this section

property enabled: bool

enables or disables this section

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

property height: int

The height of this section

property left: int

Left edge of this section

property modal: bool

Returns the modal state (Prevent the following sections from receiving input events and updating)

mouse_is_on_top(*x: int, y: int*) → bool

Check if the current mouse position is on top of this section

overlaps_with(*section*) → bool

Checks if this section overlaps with another section

property right: int

Right edge of this section

property section_manager: Optional[*arcade.sections.SectionManager*]

Returns the section manager

property top: int

Top edge of this section

property view

The view this section is set on

property width: int

The width of this section

property window

The view window

7.19.24 arcade.SectionManager

class arcade.**SectionManager**(*view*)

This manages the different Sections a View has. Actions such as dispatching the events to the correct Section, draw order, etc.

add_section(*section: arcade.sections.Section*, *at_index: Optional[int] = None*) → None

Adds a section to this Section Manager :param section: the section to add to this section manager :param at_index: inserts the section at that index. If None at the end

clear_sections()

Removes all sections

disable() → None

Disable all sections

disable_all_keyboard_events() → None

Removes the keyboard events handling from all sections

dispatch_keyboard_event(*event*, **args*, ***kwargs*) → Optional[bool]

Generic method to dispatch keyboard events to the correct sections

dispatch_mouse_event(*event: str*, *x: int*, *y: int*, **args*, ***kwargs*) → Optional[bool]

Generic method to dispatch mouse events to the correct Section

enable() → None

Enables all section

get_section(*x: int*, *y: int*) → Optional[*arcade.sections.Section*]

Returns the first section based on x,y position

get_section_by_name(*name: str*) → Optional[*arcade.sections.Section*]

Returns the first section with the given name

property has_sections: bool

Returns true if sections are available

on_draw()

Called on each event loop. First dispatch the view event, then the section ones. 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.

on_mouse_drag(*x: int*, *y: int*, **args*, ***kwargs*) → Optional[bool]

This method dispatches the on_mouse_drag and also calculates if on_mouse_enter/leave should be fired

on_mouse_motion(*x: int*, *y: int*, **args*, ***kwargs*) → Optional[bool]

This method dispatches the on_mouse_motion and also calculates if on_mouse_enter/leave should be fired

on_resize(*width: int*, *height: int*)

Called when the window is resized. First dispatch the view event, then the section ones.

on_update(*delta_time: float*)

Called on each event loop. First dispatch the view event, then the section ones.

remove_section(*section: arcade.sections.Section*) → None

Removes a section from this section manager

update(*delta_time: float*)

Called on each event loop. First dispatch the view event, then the section ones.

7.20 Sound

7.20.1 arcade.Sound

class arcade.**Sound**(*file_name: Union[str, pathlib.Path]*, *streaming: bool = False*)

This class represents a sound you can play.

get_length() → float

Get length of audio in seconds

get_stream_position(*player: pyglet.media.player.Player*) → float

Return where we are in the stream. This will reset back to zero when it is done playing.

Parameters **player** (*pyglet.media.Player*) – Player returned from *play_sound()*.

get_volume(*player: pyglet.media.player.Player*) → float

Get the current volume.

Parameters **player** (*pyglet.media.Player*) – Player returned from *play_sound()*.

Returns A float, 0 for volume off, 1 for full volume.

Return type float

is_complete(*player: pyglet.media.player.Player*) → bool

Return true if the sound is done playing.

is_playing(*player: pyglet.media.player.Player*) → bool

Return if the sound is currently playing or not

Parameters **player** (*pyglet.media.Player*) – Player returned from *play_sound()*.

Returns A boolean, `True` if the sound is playing.

Return type bool

play(*volume: float = 1.0*, *pan: float = 0.0*, *loop: bool = False*, *speed: float = 1.0*) → pyglet.media.player.Player

Play the sound.

Parameters

- **volume** (*float*) – Volume, from 0=quiet to 1=loud
- **pan** (*float*) – Pan, from -1=left to 0=centered to 1=right
- **loop** (*bool*) – Loop, false to play once, true to loop continuously
- **speed** (*float*) – Change the speed of the sound which also changes pitch, default 1.0

set_volume(*volume*, *player: pyglet.media.player.Player*) → None

Set the volume of a sound as it is playing.

Parameters

- **volume** (*float*) – Floating point volume. 0 is silent, 1 is full.
- **player** (*pyglet.media.Player*) – Player returned from `play_sound()`.

stop(*player: pyglet.media.player.Player*) → None

Stop a currently playing sound.

7.20.2 arcade.load_sound

arcade.**load_sound**(*path: Union[str, pathlib.Path]*, *streaming: bool = False*) → Optional[*arcade.sound.Sound*]

Load a sound.

Parameters

- **path** (*Path*) – Name of the sound file to load.
- **streaming** (*bool*) – 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.

Return type *Sound*

7.20.3 arcade.play_sound

arcade.**play_sound**(*sound:* arcade.sound.Sound, *volume: float = 1.0*, *pan: float = 0.0*, *looping: bool = False*, *speed: float = 1.0*) → pyglet.media.player.Player

Play a sound.

Parameters

- **sound** (Sound) – Sound loaded by `load_sound()`. Do NOT use a string here for the filename.
- **volume** (*float*) – Volume, from 0=quiet to 1=loud
- **pan** (*float*) – Pan, from -1=left to 0=centered to 1=right
- **looping** (*bool*) – Should we loop the sound over and over?
- **speed** (*float*) – Change the speed of the sound which also changes pitch, default 1.0

7.20.4 arcade.stop_sound

arcade.**stop_sound**(*player: pyglet.media.player.Player*)

Stop a sound that is currently playing.

Parameters **player** (*pyglet.media.Player*) – Player returned from `play_sound()`.

7.21 Pathfinding

7.21.1 arcade.AStarBarrierList

`class arcade.AStarBarrierList`(*moving_sprite: arcade.sprite.Sprite*, *blocking_sprites: arcade.sprite_list.sprite_list.SpriteList*, *grid_size: int*, *left: int*, *right: int*, *bottom: int*, *top: int*)

Class that manages a list of barriers that can be encountered during A* path finding.

Parameters

- **moving_sprite** (`Sprite`) – Sprite that will be moving
- **blocking_sprites** (`SpriteList`) – Sprites that can block movement
- **grid_size** (*int*) – Size of the grid, in pixels
- **left** (*int*) – Left border of playing field
- **right** (*int*) – Right border of playing field
- **bottom** (*int*) – Bottom of playing field
- **top** (*int*) – Top of playing field

`recalculate()`

Recalculate blocking sprites.

7.21.2 arcade.astar_calculate_path

`arcade.astar_calculate_path`(*start_point: Union[Tuple[float, float], List[float]]*, *end_point: Union[Tuple[float, float], List[float]]*, *astar_barrier_list: arcade.paths.AStarBarrierList*, *diagonal_movement=True*)

Parameters

- **start_point** (*Point*) –
- **end_point** (*Point*) –
- **astar_barrier_list** (`AStarBarrierList`) –
- **diagonal_movement** (*bool*) –

Returns: List

7.21.3 arcade.has_line_of_sight

`arcade.has_line_of_sight`(*point_1: Union[Tuple[float, float], List[float]]*, *point_2: Union[Tuple[float, float], List[float]]*, *walls: arcade.sprite_list.sprite_list.SpriteList*, *max_distance: int = - 1*, *check_resolution: int = 2*)

Determine if we have line of sight between two points. Try to make sure that spatial hashing is enabled on the wall SpriteList or this will be very slow.

Parameters

- **point_1** (*Point*) – Start position
- **point_2** (*Point*) – End position position

- **walls** (SpriteList) – List of all blocking sprites
- **max_distance** (*int*) – Max distance point 1 can see
- **check_resolution** (*int*) – Check every x pixels for a sprite. Trade-off between accuracy and speed.

7.22 Particles

7.22.1 arcade.EternalParticle

class arcade.**EternalParticle**(*filename_or_texture: Union[str, arcade.texture.Texture], change_xy: Union[Tuple[float, float], List[float]], center_xy: Union[Tuple[float, float], List[float]] = (0.0, 0.0), angle: float = 0, change_angle: float = 0, scale: float = 1.0, alpha: int = 255, mutation_callback=None*)

Particle that has no end to its life

can_reap()

Determine if Particle can be deleted

7.22.2 arcade.FadeParticle

class arcade.**FadeParticle**(*filename_or_texture: Union[str, arcade.texture.Texture], change_xy: Union[Tuple[float, float], List[float]], lifetime: float, center_xy: Union[Tuple[float, float], List[float]] = (0.0, 0.0), angle: float = 0, change_angle: float = 0, scale: float = 1.0, start_alpha: int = 255, end_alpha: int = 0, mutation_callback=None*)

Particle that animates its alpha between two values during its lifetime

update()

Advance the Particle's simulation

7.22.3 arcade.LifetimeParticle

class arcade.**LifetimeParticle**(*filename_or_texture: Union[str, arcade.texture.Texture], change_xy: Union[Tuple[float, float], List[float]], lifetime: float, center_xy: Union[Tuple[float, float], List[float]] = (0.0, 0.0), angle: float = 0, change_angle: float = 0, scale: float = 1.0, alpha: int = 255, mutation_callback=None*)

Particle that lives for a given amount of time and is then deleted

can_reap()

Determine if Particle can be deleted

update()

Advance the Particle's simulation

7.22.4 arcade.Particle

class arcade.**Particle**(*filename_or_texture: Union[str, arcade.texture.Texture], change_xy: Union[Tuple[float, float], List[float]], center_xy: Union[Tuple[float, float], List[float]] = (0.0, 0.0), angle: float = 0.0, change_angle: float = 0.0, scale: float = 1.0, alpha: int = 255, mutation_callback=None*)

Sprite that is emitted from an Emitter

can_reap()

Determine if Particle can be deleted

update()

Advance the Particle's simulation

7.22.5 arcade.EmitBurst

class arcade.**EmitBurst**(*count: int*)

Used to configure an Emitter to emit particles in one burst

7.22.6 arcade.EmitController

class arcade.**EmitController**

Base class for how a client configure the rate at which an Emitter emits Particles

Subclasses allow the client to control the rate and duration of emitting

7.22.7 arcade.EmitInterval

class arcade.**EmitInterval**(*emit_interval: float*)

Base class used to configure an Emitter to have a constant rate of emitting. Will emit indefinitely.

7.22.8 arcade.EmitMaintainCount

class arcade.**EmitMaintainCount**(*particle_count: int*)

Used to configure an Emitter so it emits particles so that the given count is always maintained

7.22.9 arcade.Emitter

class arcade.**Emitter**(*center_xy: Union[Tuple[float, float], List[float]], emit_controller: arcade.emitter.EmitController, particle_factory: Callable[[arcade.emitter.Emitter], arcade.particle.Particle], change_xy: Union[Tuple[float, float], List[float]] = (0.0, 0.0), emit_done_cb: Optional[Callable[[arcade.emitter.Emitter], None]] = None, reap_cb: Optional[Callable[[], None]] = None*)

Emits and manages Particles over their lifetime. The foundational class in a particle system.

can_reap() → bool

Determine if Emitter can be deleted

get_pos() → Union[Tuple[float, float], List[float]]

Get position of emitter

7.22.10 arcade.EmitterIntervalWithCount

`class arcade.EmitterIntervalWithCount(emit_interval: float, particle_count: int)`

Configure an Emitter to emit particles with given interval, ending after emitting given number of particles

7.22.11 arcade.EmitterIntervalWithTime

`class arcade.EmitterIntervalWithTime(emit_interval: float, lifetime: float)`

Configure an Emitter to emit particles with given interval, ending after given number of seconds

7.22.12 arcade.make_burst_emitter

`arcade.make_burst_emitter(center_xy: Union[Tuple[float, float], List[float]], filenames_and_textures: Sequence[Union[str, arcade.texture.Texture]], particle_count: int, particle_speed: float, particle_lifetime_min: float, particle_lifetime_max: float, particle_scale: float = 1.0, fade_particles: bool = True)`

Returns an emitter that emits all of its particles at once

7.22.13 arcade.make_interval_emitter

`arcade.make_interval_emitter(center_xy: Union[Tuple[float, float], List[float]], filenames_and_textures: Sequence[Union[str, arcade.texture.Texture]], emit_interval: float, emit_duration: float, particle_speed: float, particle_lifetime_min: float, particle_lifetime_max: float, particle_scale: float = 1.0, fade_particles: bool = True)`

Returns an emitter that emits its particles at a constant rate for a given amount of time

7.23 Arcade Version Number

7.24 Isometric Map Support (incomplete)

7.24.1 arcade.create_isometric_grid_lines

`arcade.create_isometric_grid_lines(width: int, height: int, tile_width: int, tile_height: int, color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]], line_width: int) → arcade.buffered_draw_commands.ShapeElementList`

7.24.2 arcade.isometric_grid_to_screen

arcade.**isometric_grid_to_screen**(*tile_x: int*, *tile_y: int*, *width: int*, *height: int*, *tile_width: int*, *tile_height: int*) → Tuple[int, int]

7.24.3 arcade.screen_to_isometric_grid

arcade.**screen_to_isometric_grid**(*screen_x: int*, *screen_y: int*, *width: int*, *height: int*, *tile_width: int*, *tile_height: int*) → Tuple[int, int]

7.25 OpenGL Context

7.25.1 arcade.ArcadeContext

class arcade.**ArcadeContext**(*window: pyglet.window.BaseWindow*, *gc_mode: str = 'context_gc'*)

Bases: `arcade.gl.context.Context`

An OpenGL context implementation for Arcade with added custom features. This context is normally accessed thought `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** (`pyglet.window.Window`) – The pyglet window
- **gc_mode** (`str`) – 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.

classmethod activate(*ctx: arcade.gl.context.Context*)

Mark a context as the currently active one.

> **Warning:** Never call this unless you know exactly what you are doing.

property blend_func: Tuple[int, int]

Get or set the blend function. This is tuple specifying how the red, green, blue, and alpha blending factors are computed for the source and destination pixel.

Supported blend functions are:

```
ZERO
ONE
SRC_COLOR
ONE_MINUS_SRC_COLOR
DST_COLOR
ONE_MINUS_DST_COLOR
SRC_ALPHA
```

(continues on next page)

(continued from previous page)

```
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)

buffer(**, data: Optional[Any] = None, reserve: int = 0, usage: str = 'static'*) → *arcade.gl.buffer.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 `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 means 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** (*Any*) – The buffer data, This can be `bytes` or an object supporting the buffer protocol.
- **reserve** (*int*) – The number of bytes reserve

- **usage** (*str*) – Buffer usage. 'static', 'dynamic' or 'stream'

Return type *Buffer*

compute_shader(*, *source: str*) → *arcade.gl.compute_shader.ComputeShader*

Create a compute shader.

Parameters **source** (*str*) – The glsl source

copy_framebuffer(*src:* arcade.gl.framebuffer.Framebuffer, *dst:* arcade.gl.framebuffer.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** (Framebuffer) – The framebuffer to copy from
- **dst** (Framebuffer) – The framebuffer we copy to

property default_atlas: *arcade.texture_atlas.TextureAtlas*

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*

depth_texture(*size: Tuple[int, int]*, *, *data=None*) → *arcade.gl.texture.Texture*

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** (*Any*) – The texture data (optional). Can be bytes or an 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
```

property error: Optional[str]

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

property fbo: *arcade.gl.framebuffer.Framebuffer*

Get the currently active framebuffer. This property is read-only

Type *arcade.gl.Framebuffer*

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: Optional[Union[*arcade.gl.texture.Texture, *List[*arcade.gl.texture.Texture*]]] = None*, *depth_attachment: Optional[*arcade.gl.texture.Texture*] = None*) → *arcade.gl.framebuffer.Framebuffer*

Create a Framebuffer.

Parameters

- **color_attachments** (*List[*`arcade.gl.Texture`*]*) – List of textures we want to render into
- **depth_attachment** (`arcade.gl.Texture`) – Depth texture

Return type *Framebuffer*

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

Return type int

property gc_mode: str

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"
```

geometry(*content: Optional[Sequence[*arcade.gl.types.BufferDescription*]] = None*, *index_buffer: Optional[*arcade.gl.buffer.Buffer*] = None*, *mode: Optional[int] = None*, *index_element_size: int = 4*)

Create a Geomtry instance. This is Arcade's version of a vertex array adding a lot of convenice 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*) – List of *BufferDescription* (optional)
- **index_buffer** (Buffer) – Index/element buffer (optional)
- **mode** (*int*) – The default draw mode (optional)
- **mode** – The default draw mode (optional)
- **index_element_size** (*int*) – 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)

property gl_version: Tuple[int, int]

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

property info: *arcade.gl.context.Limits*

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
```

is_enabled(*flag*) → bool

Check if a context flag is enabled

Type bool

property limits: *arcade.gl.context.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
```

load_compute_shader(*path: Union[str, pathlib.Path]*) → *arcade.gl.compute_shader.ComputeShader*

Loads a compute shader from file. This methods supports resource handles.

Example:

```
ctx.load_compute_shader(":shader:compute/do_work.glsl")
```

Parameters **path** (*Union[str,pathlib.Path]*) – Path to texture

load_program(**, vertex_shader: Union[str, pathlib.Path], fragment_shader: Optional[Union[str, pathlib.Path]] = None, geometry_shader: Optional[Union[str, pathlib.Path]] = None, tess_control_shader: Optional[Union[str, pathlib.Path]] = None, tess_evaluation_shader: Optional[Union[str, pathlib.Path]] = None, defines: Optional[dict] = None*) → *arcade.gl.program.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 `:resources:` prefix. It's recommended to use absolute paths, but not required.

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** (*Union[str,pathlib.Path]*) – path to vertex shader
- **fragment_shader** (*Union[str,pathlib.Path]*) – path to fragment shader (optional)
- **geometry_shader** (*Union[str,pathlib.Path]*) – path to geometry shader (optional)
- **defines** (*dict*) – Substitute `#define` values in the source
- **tess_control_shader** (*Union[str,pathlib.Path]*) – Tessellation Control Shader
- **tess_evaluation_shader** (*Union[str,pathlib.Path]*) – Tessellation Evaluation Shader

load_texture(*path: Union[str, pathlib.Path], *, flip: bool = True, build_mipmaps: bool = False*) → *arcade.gl.texture.Texture*

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** (*Union[str,pathlib.Path]*) – Path to texture
- **flip** (*bool*) – Flips the image upside down
- **build_mipmaps** (*bool*) – Build mipmaps for the texture

objects: Deque[Any]

Collected objects to gc when gc_mode is "context_gc". This can be used during debugging.

property patch_vertices: int

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

property point_size: float

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.

---

property primitive_restart_index: int

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.

program(**, vertex_shader: str, fragment_shader: Optional[str] = None, geometry_shader: Optional[str] = None, tess_control_shader: Optional[str] = None, tess_evaluation_shader: Optional[str] = None, defines: Optional[Dict[str, str]] = None, varyings: Optional[Sequence[str]] = None, varyings_capture_mode: str = 'interleaved'*) → *arcade.gl.program.Program*

Create a *Program* given the vertex, fragment and geometry shader.

Parameters

- **vertex_shader** (*str*) – vertex shader source
- **fragment_shader** (*str*) – fragment shader source (optional)
- **geometry_shader** (*str*) – geometry shader source (optional)
- **tess_control_shader** (*str*) – tessellation control shader source (optional)
- **tess_evaluation_shader** (*str*) – tessellation evaluation shader source (optional)
- **defines** (*dict*) – Substitute #defines values in the source (optional)
- **varyings** (*Optional[Sequence[str]]*) – 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** (*str*) – 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.

Return type *Program*

property projection_2d: Tuple[float, float, float, float]

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)`

Type Tuple[float, float, float, float]

property projection_2d_matrix: pyglet.math.Mat4

Get the current projection matrix. This 4x4 float32 matrix is calculated when setting *projection_2d*.

Type pyglet.math.Mat4

pyglet_rendering()

Context manager for pyglet rendering. Since arcade and pyglet needs slightly different states we needs some initialization and cleanup.

Examples:

```
with window.ctx.pyglet_rendering():
    # Draw with pyglet here
```

query(*, *samples=True*, *time=True*, *primitives=True*)

Create a query object for measuring rendering calls in opengl.

Parameters

- **samples** (*bool*) – Collect written samples
- **time** (*bool*) – Measure rendering duration
- **primitives** (*bool*) – Collect the number of primitives emitted

Return type *Query*

reset() → None

Reset context flags and other states. This is mostly used in unit testing.

property scissor: Optional[Tuple[int, int, int, int]]

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)

property screen: *arcade.gl.framebuffer.Framebuffer*

The framebuffer for the window.

Type `Framebuffer`

property stats: *arcade.gl.context.ContextStats*

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
```

texture(*size: Tuple[int, int], *, components: int = 4, dtype: str = 'f1', data: Optional[Any] = None, wrap_x: Optional[ctypes.c_uint] = None, wrap_y: Optional[ctypes.c_uint] = None, filter: Optional[Tuple[ctypes.c_uint, ctypes.c_uint]] = None, samples: int = 0*) → *arcade.gl.texture.Texture*

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** (*int*) – Number of components (1: R, 2: RG, 3: RGB, 4: RGBA)
- **dtype** (*str*) – The data type of each component: f1, f2, f4 / i1, i2, i4 / u1, u2, u4
- **data** (*Any*) – The texture data (optional). Can be bytes or an object supporting the buffer protocol.
- **wrap_x** (*GLenum*) – How the texture wraps in x direction
- **wrap_y** (*GLenum*) – How the texture wraps in y direction
- **filter** (*Tuple[GLenum,GLenum]*) – Minification and magnification filter
- **samples** (*int*) – Creates a multisampled texture for values > 0

property viewport: Tuple[int, int, int, int]

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)

property window: pyglet.window.BaseWindow

The window this context belongs to.

Type `pyglet.Window`

7.26 Arcade OpenGL API

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).

7.26.1 Context

Context

class `arcade.gl.Context`(*window: pyglet.window.BaseWindow*, *gc_mode: str = 'context_gc'*)

Bases: `object`

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: Optional[arcade.gl.context.Context] = None

The active context

NEAREST = 9728

Texture interpolation: Nearest pixel

LINEAR = 9729

Texture interpolation: Linear interpolate

NEAREST_MIPMAP_NEAREST = 9984

Texture interpolation: Minification filter for mipmaps

LINEAR_MIPMAP_NEAREST = 9985

Texture interpolation: Minification filter for mipmaps

NEAREST_MIPMAP_LINEAR = 9986

Texture interpolation: Minification filter for mipmaps

LINEAR_MIPMAP_LINEAR = 9987

Texture interpolation: Minification filter for mipmaps

REPEAT = 10497

Texture wrap mode: Repeat

`CLAMP_TO_EDGE = 33071`

`CLAMP_TO_BORDER = 33069`

`MIRRORED_REPEAT = 33648`

`BLEND = 3042`
: Context flag: Blending

`DEPTH_TEST = 2929`
: Context flag: Depth testing

`CULL_FACE = 2884`
: Context flag: Face culling

`PROGRAM_POINT_SIZE = 34370`
: Context flag: 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.

`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
Blend equations: source - destination

FUNC_REVERSE_SUBTRACT = 32779
Blend equations: destination - source

MIN = 32775
Blend equations: Minimum of source and destination

MAX = 32776
Blend equations: Maximum of source and destination

BLEND_DEFAULT = (770, 771)
Blend mode shortcut for default blend mode: `SRC_ALPHA, ONE_MINUS_SRC_ALPHA`

BLEND_ADDITIVE = (1, 1)
Blend mode shortcut for additive blending: `ONE, ONE`

BLEND_PREMULTIPLIED_ALPHA = (770, 1)
Blend mode shortcut for premultipled alpha: `SRC_ALPHA, ONE`

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)

objects: Deque[Any]

Collected objects to gc when gc_mode is "context_gc". This can be used during debugging.

property info: *arcade.gl.context.Limits*

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
```

property limits: *arcade.gl.context.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
```

property stats: *arcade.gl.context.ContextStats*

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
```

property window: pyglet.window.BaseWindow

The window this context belongs to.

Type pyglet.Window

property screen: *arcade.gl.framebuffer.Framebuffer*

The framebuffer for the window.

Type Framebuffer

property fbo: *arcade.gl.framebuffer.Framebuffer*

Get the currently active framebuffer. This property is read-only

Type *arcade.gl.Framebuffer*

property gl_version: Tuple[int, int]

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

Return type int

property gc_mode: str

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"
```

property error: Optional[str]

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:* arcade.gl.context.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

property viewport: Tuple[int, int, int, int]

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)

property scissor: Optional[Tuple[int, int, int, int]]

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)

property blend_func: Tuple[int, int]

Get or set the blend function. This is tuple specifying how the red, green, blue, and alpha blending factors are computed for the source and destination pixel.

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)

property patch_vertices: int

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

property point_size: float

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.

---

property primitive_restart_index: int

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:* arcade.gl.framebuffer.Framebuffer, *dst:* arcade.gl.framebuffer.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** (`Framebuffer`) – The framebuffer to copy from
- **dst** (`Framebuffer`) – The framebuffer we copy to

buffer(**, data: Optional[Any] = None, reserve: int = 0, usage: str = 'static'*) → *arcade.gl.buffer.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 `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 means 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** (*Any*) – The buffer data, This can be `bytes` or an object supporting the buffer protocol.
- **reserve** (*int*) – The number of bytes reserve
- **usage** (*str*) – Buffer usage. 'static', 'dynamic' or 'stream'

Return type *Buffer*

framebuffer(**, color_attachments: Optional[Union[arcade.gl.texture.Texture, List[arcade.gl.texture.Texture]]] = None, depth_attachment: Optional[arcade.gl.texture.Texture] = None*) → *arcade.gl.framebuffer.Framebuffer*

Create a Framebuffer.

Parameters

- **color_attachments** (*List[arcade.gl.Texture]*) – List of textures we want to render into
- **depth_attachment** (`arcade.gl.Texture`) – Depth texture

Return type *Framebuffer*

texture(*size: Tuple[int, int], *, components: int = 4, dtype: str = 'f1', data: Optional[Any] = None, wrap_x: Optional[ctypes.c_uint] = None, wrap_y: Optional[ctypes.c_uint] = None, filter: Optional[Tuple[ctypes.c_uint, ctypes.c_uint]] = None, samples: int = 0*) → *arcade.gl.texture.Texture*

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** (*int*) – Number of components (1: R, 2: RG, 3: RGB, 4: RGBA)
- **dtype** (*str*) – The data type of each component: f1, f2, f4 / i1, i2, i4 / u1, u2, u4
- **data** (*Any*) – The texture data (optional). Can be bytes or an object supporting the buffer protocol.
- **wrap_x** (*GLenum*) – How the texture wraps in x direction
- **wrap_y** (*GLenum*) – How the texture wraps in y direction
- **filter** (*Tuple[GLenum,GLenum]*) – Minification and magnification filter
- **samples** (*int*) – Creates a multisampled texture for values > 0

depth_texture(*size: Tuple[int, int]*, ***, *data=None*) → *arcade.gl.texture.Texture*

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** (*Any*) – The texture data (optional). Can be bytes or an object supporting the buffer protocol.

geometry(*content: Optional[Sequence[arcade.gl.types.BufferDescription]] = None*, *index_buffer: Optional[arcade.gl.buffer.Buffer] = None*, *mode: Optional[int] = None*, *index_element_size: int = 4*)

Create a Geomtry instance. This is Arcade's version of a vertex array adding a lot of convenice 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)

```

(continues on next page)

(continued from previous page)

```
# 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*) – List of *BufferDescription* (optional)
- **index_buffer** (Buffer) – Index/element buffer (optional)
- **mode** (*int*) – The default draw mode (optional)
- **mode** – The default draw mode (optional)
- **index_element_size** (*int*) – 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: Optional[str] = None, geometry_shader: Optional[str] = None, tess_control_shader: Optional[str] = None, tess_evaluation_shader: Optional[str] = None, defines: Optional[Dict[str, str]] = None, varyings: Optional[Sequence[str]] = None, varyings_capture_mode: str = 'interleaved'*) → *arcade.gl.program.Program*

Create a *Program* given the vertex, fragment and geometry shader.

Parameters

- **vertex_shader** (*str*) – vertex shader source

- **fragment_shader** (*str*) – fragment shader source (optional)
- **geometry_shader** (*str*) – geometry shader source (optional)
- **tess_control_shader** (*str*) – tessellation control shader source (optional)
- **tess_evaluation_shader** (*str*) – tessellation evaluation shader source (optional)
- **defines** (*dict*) – Substitute #defines values in the source (optional)
- **varyings** (*Optional[Sequence[str]]*) – 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** (*str*) – 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.

Return type *Program*

query(**, samples=True, time=True, primitives=True*)

Create a query object for measuring rendering calls in opengl.

Parameters

- **samples** (*bool*) – Collect written samples
- **time** (*bool*) – Measure rendering duration
- **primitives** (*bool*) – Collect the number of primitives emitted

Return type *Query*

compute_shader(**, source: str*) → *arcade.gl.compute_shader.ComputeShader*

Create a compute shader.

Parameters **source** (*str*) – The glsl source

ContextStats

class arcade.gl.context.**ContextStats**(*warn_threshold=100*)

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** (*str*) – The attribute name / counter to increment.

decr(*key*)
Decrement a counter.

Parameters **key** (*str*) – The attribute name / counter to decrement.

Limits

class arcade.gl.context.**Limits**(*ctx*)
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_DUAL_SOURCE_DRAW_BUFFERS
Maximum number of active draw buffers when using dual-source blending

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_RECTANGLE_TEXTURE_SIZE
A rough estimate of the largest rectangular texture that the GL can handle

MAX_RENDERBUFFER_SIZE
Maximum supported size for renderbuffers

MAX_SAMPLE_MASK_WORDS
Maximum number of sample mask words

MAX_TEXTURE_BUFFER_SIZE
Maximum number of texels allowed in the texel array of a texture buffer object

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
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: ctypes.c_uint*, *length: int*)
Get an enum as an int tuple

get(*enum: ctypes.c_uint*) → int
Get an integer limit

get_float(*enum: ctypes.c_uint*) → float
Get a float limit

get_str(*enum: ctypes.c_uint*) → str
Get a string limit

7.26.2 Texture

class arcade.gl.**Texture**(*ctx: Context*, *size: Tuple[int, int]*, ***, *components: int = 4*, *dtype: str = 'f1'*, *data: Any = None*, *filter: Tuple[ctypes.c_uint, ctypes.c_uint] = None*, *wrap_x: ctypes.c_uint = None*, *wrap_y: ctypes.c_uint = None*, *target=3553*, *depth=False*, *samples: int = 0*)

Bases: `object`

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.

NOTE: Currently does not support multisample textures even though `_samples` is set.

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** (`Context`) – The context the object belongs to
- **size** (*Tuple[int, int]*) – The size of the texture
- **components** (*int*) – The number of components (1: R, 2: RG, 3: RGB, 4: RGBA)
- **dtype** (*str*) – The data type of each component: f1, f2, f4 / i1, i2, i4 / u1, u2, u4

- **data** (*Any*) – The texture data (optional). Can be bytes or any object supporting the buffer protocol.
- **data** – The byte data of the texture. bytes or anything supporting the buffer protocol.
- **filter** (*Tuple[gl.GLuint,gl.GLuint]*) – The minification/magnification filter of the texture
- **wrap_x** (*gl.GLuint*) – Wrap mode x
- **wrap_y** (*gl.GLuint*) – Wrap mode y
- **target** (*int*) – The texture type (Ignored. Legacy)
- **depth** (*bool*) – creates a depth texture if *True*
- **samples** (*int*) – Creates a multisampled texture for values > 0. This value will be clamped between 0 and the max sample capability reported by the drivers.

resize(*size: Tuple[int, int]*)

Resize the texture. This will re-allocate the internal memory and all pixel data will be lost.

property ctx: *Context*

The context this texture belongs to

Type *Context*

property glo: ctypes.c_uint

The OpenGL texture id

Type GLuint

property width: int

The width of the texture in pixels

Type int

property height: int

The height of the texture in pixels

Type int

property dtype: str

The data type of each component

Type str

property size: Tuple[int, int]

The size of the texture as a tuple

Type tuple (width, height)

property samples: int

Number of samples if multisampling is enabled (read only)

Type int

property byte_size: int

The byte size of the texture.

Type int

property components: int

Number of components in the texture

Type int

property depth: bool

If this is a depth texture.

Type bool

property swizzle: str

str: 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'
```

property filter: Tuple[int, int]

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)

property wrap_x: int

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

property wrap_y: int

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

property anisotropy: float

Get or set the anisotropy for this texture.

property compare_func: Optional[str]

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*) → bytearray

Read the contents of the texture.

Parameters

- **level** (*int*) – The texture level to read
- **alignment** (*int*) – Alignment of the start of each row in memory in number of bytes. Possible values: 1,2,4

Return type bytearray

write(*data: Union[bytes, arcade.gl.buffer.Buffer, array.array]*, *level: int = 0*, *viewport=None*) → None

Write byte data to the texture. This can be bytes or a `Buffer`.

Parameters

- **data** (*Union[bytes,Buffer]*) – bytes or a Buffer with data to write
- **level** (*int*) – The texture level to write
- **viewport** (*tuple*) – The are 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. Leaveing the default arguments will usually does the job. Building mipmaps will create several smaller versions of the texture (256 x 256, 128 x 128, 64 x 64, 32 x 32 etc) helping OpenGL in rendering a nicer version of texture when it's rendered to the screen in smaller version.

Note that mipmaps will only be used if the texture filter is configured with a mipmap-type minification:

```
# Set up linear interpolating minification filter
texture.filter = ctx.LINEAR_MIPMAP_LINEAR, ctx.LINEAR
```

Parameters

- **base** (*int*) – Level the mipmaps start at (usually 0)
- **max_level** (*int*) – The maximum 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: ctypes.c_uint*)

Destroy the texture. This is called automatically when the object is garbage collected.

Parameters

- **ctx** (`arcade.gl.Context`) – OpenGL Context
- **glo** (*gl.GLuint*) – The OpenGL texture id

use(*unit: int = 0*) → None

Bind the texture to a channel,

Parameters **unit** (*int*) – 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** (*int*) – The image unit
- **read** (*bool*) – The compute shader intends to read from this image
- **write** (*bool*) – The compute shader intends to write to this image
- **level** (*int*) –

7.26.3 Buffer

class arcade.gl.**Buffer**(*ctx: Context, data: Optional[Any] = None, reserve: int = 0, usage: str = 'static'*)

Bases: `object`

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.

Buffer objects should be created using `arcade.gl.Context.buffer()`

Parameters

- **ctx** (`Context`) – The context this buffer belongs to
- **data** (*Any*) – The data this buffer should contain. It can be bytes or any object supporting the buffer protocol.
- **reserve** (*int*) – Create a buffer of a specific byte size
- **usage** (*str*) – A hit of this buffer is `static` or `dynamic` (can mostly be ignored)

property size: int

The byte size of the buffer.

Type int

property ctx: Context

The context this resource belongs to.

Type `arcade.gl.Context`

property glo: ctypes.c_uint

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: ctypes.c_uint*)

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** (*int*) – The bytes to read. -1 means the entire buffer (default)
- **offset** (*int*) – Byte read offset

Return type bytes

write(*data: Any*, *offset: int = 0*)

Write byte data to the buffer.

Parameters

- **data** (*bytes*) – The byte data to write. This can be bytes or any object supporting the buffer protocol.
- **offset** (*int*) – The byte offset

copy_from_buffer(*source: arcade.gl.buffer.Buffer*, *size=- 1*, *offset=0*, *source_offset=0*)

Copy data into this buffer from another buffer

Parameters

- **source** (*Buffer*) – The buffer to copy from
- **size** (*int*) – The amount of bytes to copy
- **offset** (*int*) – The byte offset to write the data in this buffer
- **source_offset** (*int*) – 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** (*int*) – New size of buffer. -1 will retain the current size.
- **double** (*bool*) – 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** (*int*) – The binding location
- **offset** (*int*) – byte offset
- **size** (*int*) – 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** (*int*) – The binding location
- **offset** (*int*) – Byte offset in the buffer
- **size** (*int*) – The size in bytes. The entire buffer will be mapped by default.

7.26.4 BufferDescription

class arcade.gl.**BufferDescription**(*buffer: arcade.gl.buffer.Buffer*, *formats: str*, *attributes: Iterable[str]*, *normalized: Optional[Iterable[str]] = None*, *instanced: bool = False*)

Bases: `object`

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␣
↪(exact name)
BufferDescription(
    my_buffer,
    '2f 2f',
    ['in_pos', 'in_uv'],
)
```

Parameters

- **buffer** (Buffer) – The buffer to describe
- **formats** (*str*) – The format of each attribute
- **attributes** (*list*) – List of attributes names (strings)
- **normalized** (*list*) – list of attribute names that should be normalized
- **instanced** (*bool*) – `True` if this is per instance data

buffer: *arcade.gl.buffer.Buffer*

The *Buffer* this description object describes

attributes

List of string attributes

normalized

List of normalized attributes

instanced: bool

Instanced flag (bool)

formats: List[arcade.gl.types.AttribFormat]

Formats of each attribute

stride: int

The byte stride of the buffer

num_vertices: int

Number of vertices in the buffer

7.26.5 Geometry

Geometry Methods

arcade.gl.geometry.**quad_2d_fs**() → *arcade.gl.vertex_array.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)*) → *arcade.gl.vertex_array.Geometry*

Creates 2D quad Geometry using 2 triangle strip with texture coordinates.

Parameters

- **size** (*tuple*) – width and height
- **pos** (*float*) – Center position x and y

Return type A `Geometry` instance.

arcade.gl.geometry.**screen_rectangle**(*bottom_left_x: float, bottom_left_y: float, width: float, height: float*) → *arcade.gl.vertex_array.Geometry*

Creates screen rectangle using 2 triangle strip with texture coordinates.

Parameters

- **bottom_left_x** (*float*) – Bottom left x position
- **bottom_left_y** (*float*) – Bottom left y position
- **width** (*float*) – Width of the rectangle
- **height** (*float*) – 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)*) → *arcade.gl.vertex_array.Geometry*

Creates a cube with normals and texture coordinates.

Parameters

- **size** (*tuple*) – size of the cube as a 3-component tuple
- **center** (*tuple*) – center of the cube as a 3-component tuple

Return type *arcade.gl.Geometry*

Returns A cube

Geometry

class arcade.gl.**Geometry**(*ctx: Context*, *content: Optional[Sequence[arcade.gl.types.BufferDescription]]*, *index_buffer: arcade.gl.buffer.Buffer = None*, *mode=None*, *index_element_size: int = 4*)

Bases: `object`

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** (*Contex*) – The context this object belongs to
- **content** (*list*) – List of BufferDescriptions
- **index_buffer** (Buffer) – Index/element buffer
- **mode** (*int*) – The default draw mode

property ctx: Context

The context this geometry belongs to.

Type *Geometry*

property index_buffer: Optional[arcade.gl.buffer.Buffer]

Index/element buffer if supplied at creation.

Type *Buffer*

property num_vertices: int

Get or set the number of vertices. Be careful when modifying this properly and be absolutely sure what you are doing.

Type int

instance(*program: arcade.gl.program.Program*) → *arcade.gl.vertex_array.VertexArray*

Get the `arcade.gl.VertexArray` compatible with this program

render(*program: arcade.gl.program.Program*, ***, *mode: Optional[ctypes.c_uint] = None*, *first: int = 0*, *vertices: Optional[int] = 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** (Program) – The Program to render with
- **mode** (*gl.GLenum*) – Override what primitive mode should be used
- **first** (*int*) – Offset start vertex
- **vertices** (*int*) – Override the number of vertices to render
- **instances** (*int*) – Number of instances to render

render_indirect(*program: arcade.gl.program.Program*, *buffer: arcade.gl.buffer.Buffer*, *, *mode: Optional[ctypes.c_uint] = 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** (`Program`) – The program to execute
- **buffer** (`Buffer`) – The buffer containing one or multiple draw parameters
- **mode** (*GLuint*) – Primitive type to render. TRIANGLES, LINES etc.
- **count** (*int*) – The number if indirect draw calls to run. If omitted all draw commands in the buffer will be executed.
- **first** (*int*) – The first indirect draw call to start on
- **stride** (*int*) – The byte stride of the draw command buffer. Keep the default (0) if the buffer is tightly packed.

transform(*program: arcade.gl.program.Program*, *buffer: Union[arcade.gl.buffer.Buffer, List[arcade.gl.buffer.Buffer]]*, *, *first: int = 0*, *vertices: Optional[int] = 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** (`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** (*int*) – Offset start vertex
- **vertices** (*int*) – Number of vertices to render
- **instances** (*int*) – Number of instances to render
- **buffer_offset** (*int*) – 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: arcade.gl.program.Program*, *content: Sequence[arcade.gl.types.BufferDescription]*, *index_buffer: arcade.gl.buffer.Buffer = None*, *index_element_size: int = 4*)

Bases: `object`

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.

property ctx: *Context*

The Context this object belongs to

Type *arcade.gl.Context*

property program: *arcade.gl.program.Program*

The assigned program

Type *arcade.gl.Program*

property ibo: Optional[*arcade.gl.buffer.Buffer*]

Element/index buffer

Type *arcade.gl.Buffer*

property num_vertices: int

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: ctypes.c_uint*)

Delete this object. This is automatically called when this object is garbage collected.

render(*mode: ctypes.c_uint*, *first: int = 0*, *vertices: int = 0*, *instances: int = 1*)

Render the VertexArray to the currently active framebuffer.

Parameters

- **mode** (*GLuint*) – Primitive type to render. TRIANGLES, LINES etc.
- **first** (*int*) – The first vertex to render from
- **vertices** (*int*) – Number of vertices to render
- **instances** (*int*) – OpenGL instance, used in using vertices over and over

render_indirect(*buffer: arcade.gl.buffer.Buffer*, *mode: ctypes.c_uint*, *count*, *first*, *stride*)

Render the VertexArray to the framebuffer using indirect rendering.

> **Warning:** This requires OpenGL 4.3

Parameters

- **buffer** (Buffer) – The buffer containing one or multiple draw parameters
- **mode** (*GLuint*) – Primitive type to render. TRIANGLES, LINES etc.
- **count** (*int*) – The number if indirect draw calls to run
- **first** (*int*) – The first indirect draw call to start on
- **stride** (*int*) – The byte stride of the draw command buffer. Keep the default (0) if the buffer is tightly packed.

transform_interleaved(*buffer: arcade.gl.buffer.Buffer*, *mode: ctypes.c_uint*, *output_mode: ctypes.c_uint*, *first: int = 0*, *vertices: int = 0*, *instances: int = 1*, *buffer_offset=0*)

Run a transform feedback.

Parameters

- **buffer** (Buffer) – The buffer to write the output
- **mode** (*gl.GLenum*) – The input primitive mode
- **output_mode** (*gl.GLenum*) – The output primitive mode
- **first** (*int*) – Offset start vertex
- **vertices** (*int*) – Number of vertices to render
- **instances** (*int*) – Number of instances to render
- **buffer_offset** (*int*) – Byte offset for the buffer (target)

transform_separate(*buffers: List[arcade.gl.buffer.Buffer]*, *mode: ctypes.c_uint*, *output_mode: ctypes.c_uint*, *first: int = 0*, *vertices: int = 0*, *instances: int = 1*, *buffer_offset=0*)

glo

7.26.6 Framebuffer

FrameBuffer

class arcade.gl.**Framebuffer**(*ctx:* Context, **, *color_attachments=None*, *depth_attachment=None*)

Bases: `object`

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()`:

```
# 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
ctx.framebuffer(
    color_attachments=[
        ctx.texture((100, 100), components=4),
        ctx.texture((100, 100), components=4),
    ]
)
```

Parameters

- **ctx** (`Context`) – The context this framebuffer belongs to
- **color_attachments** (*List[*`arcade.gl.Texture`*]*) – List of color attachments.
- **depth_attachment** (`arcade.gl.Texture`) – A depth attachment (optional)

is_default = False

Is this the default framebuffer? (window buffer)

property glo: ctypes.c_uint

The OpenGL id/name of the framebuffer

Type GLuint

property viewport: Tuple[int, int, int, int]

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
```

property scissor: Optional[Tuple[int, int, int, int]]

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)

property ctx: *Context*

The context this object belongs to.

Type *arcade.gl.Context*

property width: int

The width of the framebuffer in pixels

Type int

property height: int

The height of the framebuffer in pixels

Type int

property size: Tuple[int, int]

Size as a `(w, h)` tuple

Type tuple (int, int)

property samples: int

Number of samples (MSAA)

Type int

property color_attachments: List[*arcade.gl.texture.Texture*]

A list of color attachments

Type list of *arcade.gl.Texture*

property depth_attachment: *arcade.gl.texture.Texture*

Depth attachment

Type *arcade.gl.Texture*

property depth_mask: bool

Get or set the depth mask (default: `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

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** (*bool*) – Force the framebuffer binding even if the system already believes it's already bound.

clear(*color=(0.0, 0.0, 0.0, 0.0)*, ***, *depth: float = 1.0*, *normalized: bool = False*, *viewport: Optional[Tuple[int, int, int, int]] = None*)

Clears the framebuffer:

```
# Clear framebuffer using the color red in normalized form
fbo.clear(color=(1.0, 0.0, 0.0, 1.0), normalized=True)
# Clear the framebuffer using arcade's colors (not normalized)
fb.clear(color=arcade.color.WHITE)
```

If the background color is an `RGB` value instead of `RGBA`` we assume alpha value 255.

Parameters

- **color** (*tuple*) – A 3 or 4 component tuple containing the color
- **depth** (*float*) – Value to clear the depth buffer (unused)
- **normalized** (*bool*) – 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'*) → bytearray

Read framebuffer pixels

Parameters

- **viewport** (*Tuple[int,int,int,int]*) – The x, y, with, height to read
- **components** (*int*) –
- **attachment** (*int*) – The attachment id to read from
- **dtype** (*str*) – 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: `arcade.gl.framebuffer.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)

property viewport: Tuple[int, int, int, int]

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
```

property scissor: Optional[Tuple[int, int, int, int]]

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)

7.26.7 Query

class arcade.gl.Query(*ctx: Context*, *samples=True*, *time=True*, *primitives=True*)

Bases: `object`

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)
```

property ctx: *Context*

The context this query object belongs to

Type *`arcade.gl.Context`*

property samples_passed: int

How many samples was written. These are per component (RGBA)

Type int

property time_elapsed: int

The time elapsed in nanoseconds

Type int

property primitives_generated: int

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.

7.26.8 Program

Program

class arcade.gl.**Program**(*ctx: Context*, *, *vertex_shader: str*, *fragment_shader: str = None*, *geometry_shader: str = None*, *tess_control_shader: str = None*, *tess_evaluation_shader: str = None*, *varyings: List[str] = None*, *varyings_capture_mode: str = 'interleaved'*)

Bases: `object`

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** (`Context`) – The context this program belongs to
- **vertex_shader** (*str*) – vertex shader source
- **fragment_shader** (*str*) – fragment shader source
- **geometry_shader** (*str*) – geometry shader source
- **tess_control_shader** (*str*) – tessellation control shader source
- **tess_evaluation_shader** (*str*) – tessellation evaluation shader source
- **varyings** (*List[str]*) – List of out attributes used in transform feedback.
- **varyings_capture_mode** (*str*) – 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

property ctx: *Context*

The context this program belongs to

Type *arcade.gl.Context*

property glo: int

The OpenGL resource id for this program

Type int

property attributes: Iterable[arcade.gl.types.AttribFormat]

List of attribute information

property varyings: List[str]

Out attributes names used in transform feedback

Type list of str

property out_attributes: List[str]

Out attributes names used in transform feedback.

> **Warning:** Old alias for `varyings`. May be removed in the future.

Type list of str

property varyings_capture_mode: str

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.

property geometry_input: int

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

property geometry_output: int

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

property geometry_vertices: int

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*)

set_uniform_safe(*name: str*, *value: Any*)

Safely set a uniform catching KeyError.

Parameters

- **name** (*str*) – The uniform name
- **value** (*Any*) – 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** (*str*) – Name of uniform
- **value** (*List[Any]*) – List of values

use()

Activates the shader. This is normally done for you automatically.

static compile_shader(*source: str*, *shader_type: ctypes.c_uint*) → ctypes.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**(*program_id*, *location*, *name*, *data_type*, *array_length*)

Bases: object

A Program uniform

Parameters

- **location** (*int*) – The location of the uniform in the program
- **name** (*str*) – Name of the uniform in the program
- **data_type** (*gl.GLenum*) – The data type of the uniform (GL_FLOAT

property location: int

The location of the uniform in the program

property name: str

Name of the uniform

property array_length: int

Length of the uniform array. If not an array 1 will be returned

property components: int

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: `object`

Wrapper for a uniform block in shaders.

glo

index

size

name

property binding: int

Get or set the binding index for this uniform block

getter()

setter(*value: int*)

7.26.9 Compute Shader

class `arcade.gl.ComputeShader`(*ctx: Context*, *glsl_source: str*)

Bases: `object`

A higher level wrapper for an OpenGL compute shader.

property glo: int

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** (*int*) – The number of work groups to be launched in the X dimension.
- **group_y** (*int*) – The number of work groups to be launched in the y dimension.
- **group_z** (*int*) – The number of work groups to be launched in the z dimension.

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

7.26.10 Exceptions

class arcade.gl.**ShaderException**

Bases: `Exception`

Exception class for shader-specific problems.

7.27 GUI

7.27.1 arcade.gui.UIMessageBox

class arcade.gui.**UIMessageBox**(*, *width: float*, *height: float*, *message_text: str*, *buttons=('Ok',)*, *callback=None*)

A simple dialog box that pops up a message with buttons to close.

Parameters

- **width** – Width of the message box
- **height** – Height of the message box
- **message_text** –
- **buttons** – List of strings, which are shown as buttons
- **callback** – Callback function, will receive the text of the clicked button

7.27.2 arcade.gui.UIDraggableMixin

class arcade.gui.**UIDraggableMixin**(*x=0*, *y=0*, *width=100*, *height=100*, *children: Iterable[arcade.gui.widgets.UIWidget] = ()*, *size_hint=None*, *size_hint_min=None*, *size_hint_max=None*, *style=None*, ***kwargs*)

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`

7.27.3 arcade.gui.UIMouseFilterMixin

class arcade.gui.**UIMouseFilterMixin**(*x: float = 0, y: float = 0, width: float = 100, height: float = 100, children: Iterable[arcade.gui.widgets.UIWidget] = (), size_hint=None, size_hint_min=None, size_hint_max=None, style=None, **kwargs*)

`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.

7.27.4 arcade.gui.UIWindowLikeMixin

class arcade.gui.**UIWindowLikeMixin**(*x=0, y=0, width=100, height=100, children: Iterable[arcade.gui.widgets.UIWidget] = (), size_hint=None, size_hint_min=None, size_hint_max=None, style=None, **kwargs*)

Makes a widget window like:

- handles all mouse events that occur within the widgets boundaries
- can be dragged

7.27.5 arcade.gui.Surface

class arcade.gui.**Surface**(**, size: Tuple[int, int], position: Tuple[int, int] = (0, 0), pixel_ratio: float = 1.0*)

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: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]] = (0, 0, 0, 0)*)

Clear the surface

draw() → None

Draws the current buffer on screen

draw_sprite(*x, y, width, height, sprite*)

Draw a sprite to the surface

limit(*x, y, width, height*)

Reduces the draw area to the given rect

property position: Tuple[int, int]

Get or set the surface position

resize(**, size: Tuple[int, int], pixel_ratio: float*) → None

Resize the internal texture by re-allocating a new one

Parameters

- **size** (*Tuple[int, int]*) – The new size in pixels (xy)
- **pixel_ratio** (*float*) – The pixel scale of the window

property size

Size of the surface in window coordinates

property size_scaled

The physical size of the buffer

7.27.6 arcade.gui.UIManager

class arcade.gui.**UIManager**(*window: Optional[arcade.application.Window] = None*, *auto_enable=False*)

V2 UIManager

```
manager = UIManager()
manager.enable() # hook up window events

manager.add(Dummy())

def on_draw():
    self.clear()

    ...

    manager.draw() # draws the UI on screen
```

add(*widget: arcade.gui.ui_manager.W*, ***, *index=None*) → arcade.gui.ui_manager.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.

Parameters

- **widget** – widget to add
- **index** – position a widget is added, None has the highest priority

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()

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()*.

`enable()`

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.

`get_widgets_at`(*pos*, *cls=<class 'arcade.gui.widgets.UIWidget'>*) → Iterable[arcade.gui.ui_manager.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 instance of

Returns iterator of widgets of given type at position

`remove`(*child:* arcade.gui.widgets.UIWidget)

Removes the given widget from UIManager.

Parameters **child** (`UIWidget`) – widget to remove

`trigger_render()`

Request rendering of all widgets

`walk_widgets`(*, *root: Optional[*arcade.gui.widgets.UIWidget*] = None*) → Iterable[*arcade.gui.widgets.UIWidget*]

walks through widget tree, in reverse draw order (most top drawn widget first)

7.28 GUI Widgets

7.28.1 arcade.gui.UIAnchorWidget

class `arcade.gui.UIAnchorWidget`(*, *child:* arcade.gui.widgets.UIWidget, *anchor_x: str = 'center'*, *align_x: float = 0*, *anchor_y: str = 'center'*, *align_y: float = 0*, *size_hint=None*, *size_hint_min=None*, *size_hint_max=None*, *style=None*, ***kwargs*)

Widget, which places itself relative to the parent.

Parameters

- **child** – Child of this wrapper
- **anchor_x** – Which anchor to use for x axis (left, center, right)
- **align_x** – offset for x value (- = left, + = right)
- **anchor_y** – Which anchor to use for y axis (top, center, bottom)
- **align_y** – offset for y value (- = down, + = up)
- **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

7.28.2 arcade.gui.UIBorder

class `arcade.gui.UIBorder`(*child:* arcade.gui.widgets.UIWidget, *border_width=2*, *border_color=(0, 0, 0, 255)*, *size_hint=None*, *size_hint_min=None*, *size_hint_max=None*, *style=None*, ***kwargs*)

Wraps a Widget with a border of given color.

Parameters

- **child** – Child of this wrapper
- **border_width** – Width of the border
- **border_color** – Color of the border
- **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

7.28.3 arcade.gui.UIBoxLayout

class `arcade.gui.UIBoxLayout`(*x=0*, *y=0*, *vertical=True*, *align='center'*, *children:* *Iterable*[arcade.gui.widgets.UIWidget] *= ()*, *size_hint=None*, *size_hint_min=None*, *size_hint_max=None*, *space_between=0*, *style=None*, ***kwargs*)

Places widgets next to each other. Depending on the vertical attribute, the Widgets are placed top to bottom or left to right.

Parameters

- **x** (`float`) – x coordinate of bottom left
- **y** (`float`) – y coordinate of bottom left
- **vertical** – Layout children vertical (True) or horizontal (False)
- **align** – Align children in orthogonal direction (x: left, center, right / y: top, center, bottom)
- **children** – Initial children, more can be added
- **size_hint** – A hint for `UILayout`, if this `UIWidget` would like to grow
- **size_hint_min** – min width and height in pixel
- **size_hint_max** – max width and height in pixel
- **space_between** – Space between the children

7.28.4 arcade.gui.UIDummy

class arcade.gui.**UIDummy**(*x=0*, *y=0*, *width=100*, *height=100*, *color=(0, 0, 0)*, *size_hint=None*, *size_hint_min=None*, *size_hint_max=None*, *style=None*, ***kwargs*)

Solid color widget, used for testing.

Parameters

- **x** (*float*) – x coordinate of bottom left
- **y** (*float*) – 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

7.28.5 arcade.gui.UIFlatButton

class arcade.gui.**UIFlatButton**(*x: float = 0*, *y: float = 0*, *width: float = 100*, *height: float = 50*, *text=''*, *size_hint=None*, *size_hint_min=None*, *size_hint_max=None*, *style=None*, ***kwargs*)

A text button, with support for background color and a border.

Parameters

- **x** (*float*) – x coordinate of bottom left
- **y** (*float*) – y coordinate of bottom left
- **width** (*float*) – width of widget. Defaults to texture width if not specified.
- **height** (*float*) – height of widget. Defaults to texture height if not specified.
- **text** (*str*) – text to add to the button.
- **style** – Used to style the button

7.28.6 arcade.gui.UIInputText

class arcade.gui.**UIInputText**(*x: float = 0*, *y: float = 0*, *width: float = 100*, *height: float = 50*, *text: str = ''*, *font_name=('Arial',)*, *font_size: float = 12*, *text_color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]] = (0, 0, 0, 255)*, *multiline=False*, *size_hint=None*, *size_hint_min=None*, *size_hint_max=None*, *style=None*, ***kwargs*)

An input field the user can type text into.

Parameters

- **x** (*float*) – x coordinate of bottom left
- **y** (*float*) – y coordinate of bottom left

- **width** – width of widget
- **height** – height of widget
- **text** – Text to show
- **font_name** – string or tuple of font names, to load
- **font_size** – size of the text
- **text_color** – color of the text
- **multiline** – support for multiline
- **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

7.28.7 arcade.gui.UIInteractiveWidget

class arcade.gui.**UIInteractiveWidget**(*x=0*, *y=0*, *width=100*, *height=100*, *size_hint=None*, *size_hint_min=None*, *size_hint_max=None*, *style=None*, ***kwargs*)

Base class for widgets which use mouse interaction (hover, pressed, clicked)

Parameters

- **x** (*float*) – x coordinate of bottom left
- **y** (*float*) – 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

7.28.8 arcade.gui.UILabel

class arcade.gui.**UILabel**(*x: float = 0*, *y: float = 0*, *width: Optional[float] = None*, *height: Optional[float] = None*, *text: str = ''*, *font_name=('Arial',)*, *font_size: float = 12*, *text_color: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]] = (255, 255, 255, 255)*, *bold=False*, *italic=False*, *stretch=False*, *anchor_x='left'*, *anchor_y='bottom'*, *align='left'*, *dpi=None*, *multiline: bool = False*, *size_hint=None*, *size_hint_min=None*, *size_hint_max=None*, *style=None*, ***kwargs*)

A simple text label. Also supports multiline text. In case you want to scroll text use a `UITextArea` By default a `UILabel` will fit its initial content, if the text changed use `UILabel.fit_content()` to adjust the size.

Parameters

- **x** (*float*) – x coordinate of bottom left
- **y** (*float*) – y coordinate of bottom left

- **width** (*float*) – width of widget. Defaults to text width if not specified.
- **height** (*float*) – height of widget. Defaults to text height if not specified.
- **text** (*str*) – text of the label.
- **font_name** – a list of fonts to use. Program will start at the beginning of the list and keep trying to load fonts until success.
- **font_size** (*float*) – size of font.
- **text_color** (*arcade.Color*) – Color of font.
- **bold** (*bool*) – Bold font style.
- **italic** (*bool*) – Italic font style.
- **stretch** (*bool*) – Stretch font style.
- **anchor_x** (*str*) – Anchor point of the X coordinate: one of `"left"`, `"center"` or `"right"`.
- **anchor_y** (*str*) – Anchor point of the Y coordinate: one of `"bottom"`, `"baseline"`, `"center"` or `"top"`.
- **align** (*str*) – Horizontal alignment of text on a line, only applies if a width is supplied. One of `"left"`, `"center"` or `"right"`.
- **dpi** (*float*) – Resolution of the fonts in this layout. Defaults to 96.
- **multiline** (*bool*) – if multiline is true, a n will start a new line. A UITextWidget with multiline of true is the same thing as UITextArea.
- **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.

fit_content()

Sets the width and height of this UIWidget to contain the whole text.

7.28.9 arcade.gui.UILayout

class arcade.gui.**UILayout**(*x=0*, *y=0*, *width=100*, *height=100*, *children: Iterable[arcade.gui.widgets.UIWidget] = ()*, *size_hint=None*, *size_hint_min=None*, *size_hint_max=None*, *style=None*, ***kwargs*)

Base class for widgets, which position themselves or their children.

Parameters

- **x** (*float*) – x coordinate of bottom left
- **y** (*float*) – 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.

7.28.10 arcade.gui.UIPadding

class arcade.gui.**UIPadding**(*child:* arcade.gui.widgets.UIWidget, *padding=(0, 0, 0, 0)*, *bg_color=None*, *size_hint=(1, 1)*, *size_hint_min=None*, *size_hint_max=None*, ***kwargs*)

Wraps a Widget and applies padding.

Parameters

- **child** – Child of this wrapper
- **bg_color** – background color
- **padding** – Space between the outer border of this widget and the child
- **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

7.28.11 arcade.gui.UISpace

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*, *style=None*, ***kwargs*)

Widget reserving space, can also have a background color.

Parameters

- **x** (*float*) – x coordinate of bottom left
- **y** (*float*) – 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

7.28.12 arcade.gui.UISpriteWidget

class arcade.gui.**UISpriteWidget**(**, x=0, y=0, width=100, height=100, sprite: Optional[arcade.sprite.Sprite] = None, size_hint=None, size_hint_min=None, size_hint_max=None, style=None, **kwargs*)

Create a UI element with a sprite that controls what is displayed.

Parameters

- **x** (*float*) – x coordinate of bottom left
- **y** (*float*) – 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

7.28.13 arcade.gui.UITextArea

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: Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]] = (255, 255, 255, 255), multiline: bool = True, scroll_speed: Optional[float] = None, size_hint=None, size_hint_min=None, size_hint_max=None, style=None, **kwargs*)

A text area for scrollable text.

Parameters

- **x** (*float*) – x coordinate of bottom left
- **y** (*float*) – y coordinate of bottom left
- **width** – width of widget
- **height** – height of widget
- **text** – Text to show
- **font_name** – string or tuple of font names, to load
- **font_size** – size of the text
- **text_color** – color of the text
- **multiline** – support for multiline
- **scroll_speed** – speed of scrolling
- **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

`fit_content()`

Sets the width and height of this UIWidget to contain the whole text.

7.28.14 arcade.gui.UITextureButton

class `arcade.gui.UITextureButton`(*x: float = 0, y: float = 0, width: Optional[float] = None, height: Optional[float] = None, texture: Optional[arcade.texture.Texture] = None, texture_hovered: Optional[arcade.texture.Texture] = None, texture_pressed: Optional[arcade.texture.Texture] = None, text: str = '', scale: Optional[float] = None, size_hint=None, size_hint_min=None, size_hint_max=None, style=None, **kwargs*)

A button with an image for the face of the button.

Parameters

- **x** (*float*) – x coordinate of bottom left
- **y** (*float*) – y coordinate of bottom left
- **width** (*float*) – width of widget. Defaults to texture width if not specified.
- **height** (*float*) – height of widget. Defaults to texture height if not specified.
- **texture** (`Texture`) – texture to display for the widget.
- **texture_hovered** (`Texture`) – different texture to display if mouse is hovering over button.
- **texture_pressed** (`Texture`) – different texture to display if mouse button is pressed while hovering over button.
- **text** (*str*) – text to add to the button.
- **style** – style information for the button.
- **scale** (*float*) – 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

7.28.15 arcade.gui.UITexturePane

class `arcade.gui.UITexturePane`(*child: arcade.gui.widgets.UIWidget, tex: arcade.texture.Texture, padding=(0, 0, 0, 0), size_hint=(1, 1), size_hint_min=None, size_hint_max=None, style=None, **kwargs*)

This wrapper draws a background before child widget is rendered

Parameters

- **child** – Child of this wrapper
- **tex** – Texture to use as background
- **padding** – Space between the outer border of this widget and the child
- **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

7.28.16 arcade.gui.UIWidget

class arcade.gui.**UIWidget**(*x: float = 0*, *y: float = 0*, *width: float = 100*, *height: float = 100*, *children: Iterable[arcade.gui.widgets.UIWidget] = ()*, *size_hint=None*, *size_hint_min=None*, *size_hint_max=None*, *style=None*, ***kwargs*)

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** (`float`) – x coordinate of bottom left
  - **y** (`float`) – 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: arcade.gui.widgets.W*, ***, *index=None*) → arcade.gui.widgets.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() → arcade.gui.widgets.W

Places this widget in the center of the current window.

clear()

Clears the child list.

dispatch_ui_event(*event: arcade.gui.events.UIEvent*)

Dispatch a `UIEvent` using pyglet event dispatch mechanism

do_render(*surface: arcade.gui.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.

move(*dx=0*, *dy=0*)

Move the widget by dx and dy.

Parameters

- **dx** – x axis difference
- **dy** – y axis difference

on_event(*event*: arcade.gui.events.UIEvent) → Optional[bool]

Passes `UIEvent` s through the widget tree.

on_update(*dt*)

Custom logic which will be triggered.

property position

Returns bottom left coordinates

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*: arcade.gui.widgets.UIWidget)

Removes the given child from children list.

scale(*factor*)

Scales the size of the widget (x,y,width, height) by factor. :param factor: scale factor

trigger_full_render()

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(*texture*: arcade.texture.Texture, *top=0*, *right=0*, *bottom=0*, *left=0*) → *arcade.gui.widgets.UITexturePane*

Wraps the widget with a background

Parameters

- **texture** – Background texture
- **top** – Top padding
- **right** – Right padding
- **bottom** – Bottom padding
- **left** – Left padding

Returns Wrapped Texture with self as child

with_border(*width=2*, *color=(0, 0, 0)*) → *arcade.gui.widgets.UIBorder*

Wraps this widget with a border

Parameters

- **width** – Border width

- **color** – Border color

Returns Wrapped Border with self as child

with_space_around(*top: float = 0, right: float = 0, bottom: float = 0, left: float = 0, bg_color: Optional[Union[Tuple[int, int, int], List[int], Tuple[int, int, int, int]]] = None*) → *arcade.gui.widgets.UIPadding*

Wraps this widget and applies padding

Parameters

- **top** – Top padding
- **right** – Right padding
- **bottom** – Bottom padding
- **left** – Left padding
- **bg_color** – Background color

Returns Wrapped Padding with self as child

7.28.17 arcade.gui.UIWidgetParent

class arcade.gui.**UIWidgetParent**

trigger_render()

Widget might request parent to rerender due to transparent part of the widget

7.28.18 arcade.gui.UIWrapper

class arcade.gui.**UIWrapper**(**, child: arcade.gui.widgets.UIWidget, padding=(0, 0, 0, 0), size_hint=None, size_hint_min=None, size_hint_max=None, style=None*)

Wraps a `arcade.gui.UIWidget` and reserves space around it, exactly one child supported.

Parameters

- **child** – Single child of this wrapper
- **padding** – space around (top, right, bottom, left)
- **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

7.29 GUI Events

7.29.1 arcade.gui.UIEvent

class arcade.gui.**UIEvent**(*source: Any*)

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.

7.29.2 arcade.gui.UIKeyEvent

class arcade.gui.**UIKeyEvent**(*source: Any*, *symbol: int*, *modifiers: int*)

7.29.3 arcade.gui.UIKeyPressEvent

class arcade.gui.**UIKeyPressEvent**(*source: Any*, *symbol: int*, *modifiers: int*)

7.29.4 arcade.gui.UIKeyReleaseEvent

class arcade.gui.**UIKeyReleaseEvent**(*source: Any*, *symbol: int*, *modifiers: int*)

7.29.5 arcade.gui.UIMouseDragEvent

class arcade.gui.**UIMouseDragEvent**(*source: Any*, *x: float*, *y: float*, *dx: float*, *dy: float*, *buttons: int*, *modifiers: int*)

7.29.6 arcade.gui.UIMouseEvent

class arcade.gui.**UIMouseEvent**(*source: Any*, *x: float*, *y: float*)

Covers all mouse event

7.29.7 arcade.gui.UIMouseMovementEvent

class arcade.gui.**UIMouseMovementEvent**(*source: Any*, *x: float*, *y: float*, *dx: float*, *dy: float*)

7.29.8 arcade.gui.UIMousePressEvent

class arcade.gui.**UIMousePressEvent**(*source: Any*, *x: float*, *y: float*, *button: int*, *modifiers: int*)

7.29.9 arcade.gui.UIMouseReleaseEvent

class arcade.gui.**UIMouseReleaseEvent**(*source: Any*, *x: float*, *y: float*, *button: int*, *modifiers: int*)

7.29.10 arcade.gui.UIMouseScrollEvent

class arcade.gui.**UIMouseScrollEvent**(*source: Any*, *x: float*, *y: float*, *scroll_x: int*, *scroll_y: int*)

7.29.11 arcade.gui.UIOnChangeEvent

class arcade.gui.**UIOnChangeEvent**(*source: Any*, *old_value: Any*, *new_value: Any*)

Value of a widget changed

7.29.12 arcade.gui.UIOnClickEvent

class arcade.gui.**UIOnClickEvent**(*source: Any*, *x: float*, *y: float*)

7.29.13 arcade.gui.UIOnUpdateEvent

class arcade.gui.**UIOnUpdateEvent**(*source: Any*, *dt: int*)

Arcade on_update callback passed as `UIEvent`

7.29.14 arcade.gui.UITextEvent

class arcade.gui.**UITextEvent**(*source: Any*, *text: str*)

7.29.15 arcade.gui.UITextMotionEvent

class arcade.gui.**UITextMotionEvent**(*source: Any*, *motion: Any*)

7.29.16 arcade.gui.UITextMotionSelectEvent

class arcade.gui.**UITextMotionSelectEvent**(*source: Any*, *selection: Any*)

7.30 arcade.key package

Mapping of keyboard keys to values.

```python
# flake8: noqa
"""
Constants used to signify what keys on the keyboard were pressed.
"""

# 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
MOD_ACCEL = 2

# 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
```

(continues on next page)

(continued from previous page)

```
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
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
```

(continues on next page)

(continued from previous page)

```
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
F12 = 65481
F13 = 65482
F14 = 65483
F15 = 65484
F16 = 65485
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
```

(continues on next page)

(continued from previous page)

```
# Numbers on the main keyboard
KEY_0 = 48
KEY_1 = 49
KEY_2 = 50
KEY_3 = 51
KEY_4 = 52
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
```

(continues on next page)

(continued from previous page)

```
Z = 122
BRACELEFT = 123
BAR = 124
BRACERIGHT = 125
ASCIITILDE = 126
```

7.31 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)`

7.32 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)`

7.33 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 3: :resources:fonts/ttf/

Kenney Mini.ttf	Kenney High Square.ttf	Kenney Pixel Square.ttf
Kenney Rocket Square.ttf	Kenney Future Narrow.ttf	Kenney Rocket.ttf
Kenney High.ttf	Kenney Mini Square.ttf	Kenney Pixel.ttf
Kenney Blocks.ttf	Kenney Future.ttf	

Table 4: :resources:images/space_shooter/

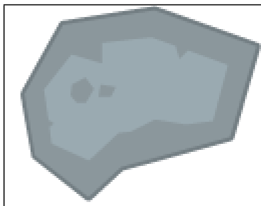
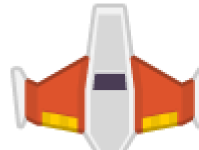
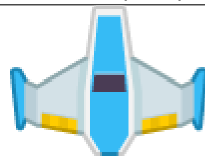
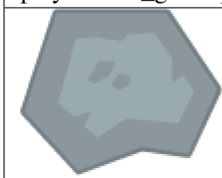
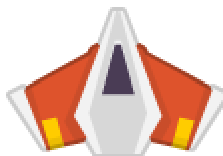
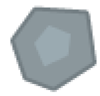
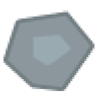
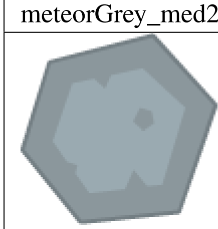
 meteorGrey_big2.png	 meteorGrey_tiny2.png	 playerShip1_orange.png
 playerLife1_green.png	 playerShip1_blue.png	 laserBlue01.png
 meteorGrey_big1.png	 playerShip2_orange.png	 meteorGrey_med1.png
 meteorGrey_small2.png	 playerShip1_green.png	 playerShip3_orange.png
 meteorGrey_tiny1.png	 playerLife1_blue.png	 playerLife1_orange.png
 meteorGrey_med2.png	 meteorGrey_small1.png	 meteorGrey_big3.png
 meteorGrey_big4.png	 laserRed01.png	

Table 5: :resources:images/isometric_dungeon/

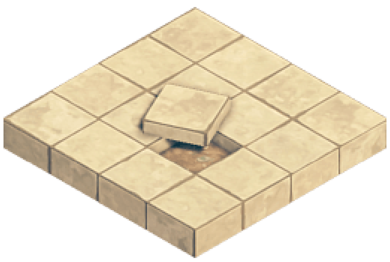
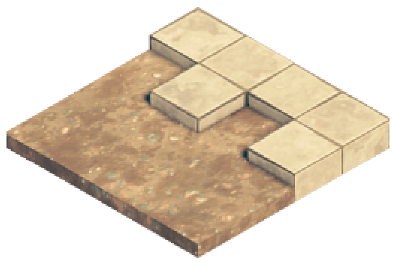
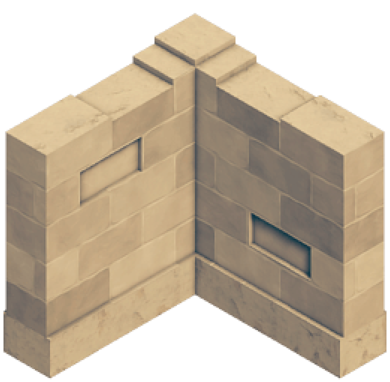
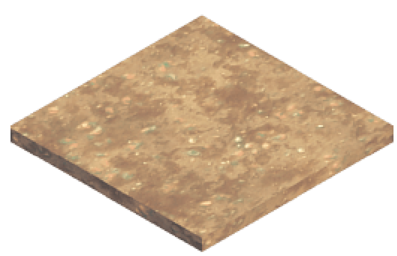
 stoneTile_W.png	 stoneSideUneven_N.png	 stoneUneven_W.png
 stoneWallCorner_N.png	 stoneMissingTiles_W.png	 dirt_S.png

Table 6: :resources:images/items/

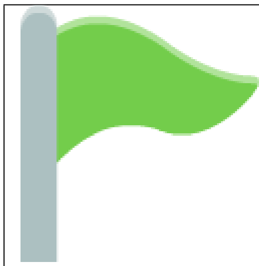
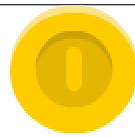
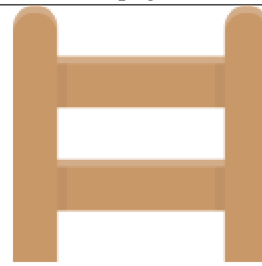
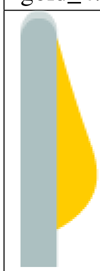
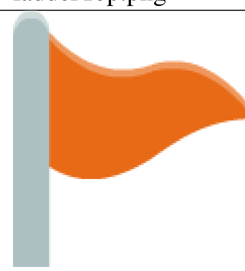
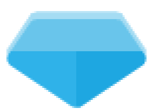
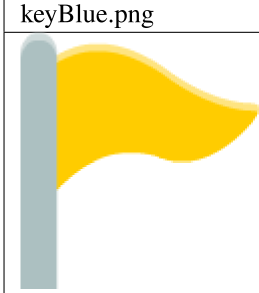
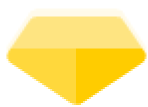
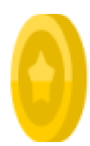
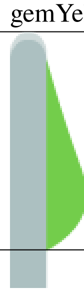
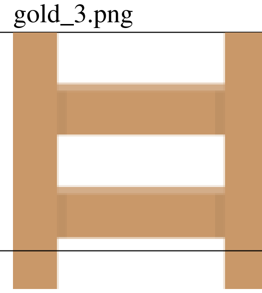
 flagGreen2.png	 coinGold_ul.png	 coinGold.png
 gold_4.png	 flagRed2.png	 ladderTop.png
 flagYellow_down.png	 flagRed_down.png	 flagRed1.png
 keyBlue.png	 gemBlue.png	 coinGold_ll.png
 flagYellow2.png	 gemYellow.png	 gold_3.png
 keyYellow.png	 flagGreen_down.png	 ladderMid.png

Table 7: :resources:images/animated_characters/female_adventurer/

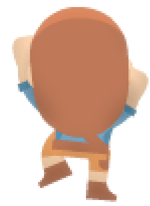
 femaleAdventurer_walk3.png	 femaleAdventurer_climb1.png	 femaleAdventurer_walk7.png
 femaleAdventurer_walk2.png	 femaleAdventurer_jump.png	 femaleAdventurer_walk1.png
 femaleAdventurer_idle.png	 femaleAdventurer_walk5.png	 femaleAdventurer_walk0.png
 femaleAdventurer_climb0.png	 femaleAdventurer_walk6.png	 femaleAdventurer_fall.png
 femaleAdventurer_walk4.png		

Table 8: :resources:images/animated_characters/male_adventurer/

 maleAdventurer_walk4.png	 maleAdventurer_jump.png	 maleAdventurer_walk7.png
 maleAdventurer_idle.png	 maleAdventurer_fall.png	 maleAdventurer_walk2.png
 maleAdventurer_climb1.png	 maleAdventurer_walk6.png	 maleAdventurer_walk0.png
 maleAdventurer_walk5.png	 maleAdventurer_walk3.png	 maleAdventurer_walk1.png
 maleAdventurer_climb0.png		

Table 9: :resources:images/animated_characters/female_person/

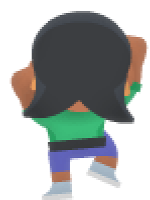
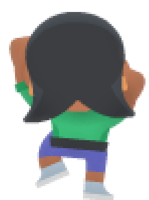
 femalePerson_walk4.png	 femalePerson_walk3.png	 femalePerson_jump.png
 femalePerson_fall.png	 femalePerson_walk2.png	 femalePerson_walk5.png
 femalePerson_walk6.png	 femalePerson_walk1.png	 femalePerson_climb0.png
 femalePerson_walk7.png	 femalePerson_climb1.png	 femalePerson_idle.png
 femalePerson_walk0.png		

Table 10: :resources:images/animated_characters/male_person/

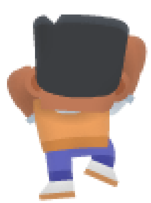
 malePerson_walk0.png	 malePerson_walk2.png	 malePerson_idle.png
 malePerson_fall.png	 malePerson_climb1.png	 malePerson_jump.png
 malePerson_walk6.png	 malePerson_walk7.png	 malePerson_walk3.png
 malePerson_climb0.png	 malePerson_walk4.png	 malePerson_walk1.png
 malePerson_walk5.png		

Table 11: :resources:images/animated_characters/zombie/

 zombie_jump.png	 zombie_climb1.png	 zombie_walk6.png
 zombie_walk5.png	 zombie_idle.png	 zombie_fall.png
 zombie_walk2.png	 zombie_walk7.png	 zombie_walk0.png
 zombie_walk3.png	 zombie_walk4.png	 zombie_walk1.png
 zombie_climb0.png		

Table 12: :resources:images/animated_characters/robot/

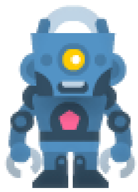
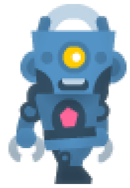
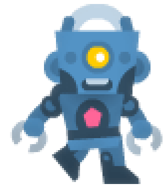
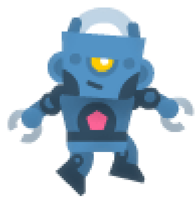
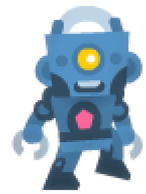
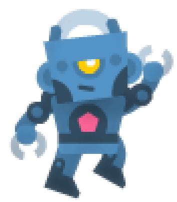
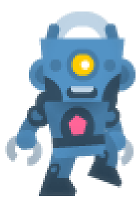
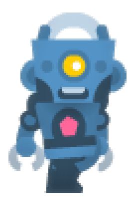
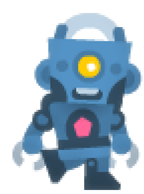
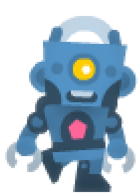
 robot_idle.png	 robot_walk6.png	 robot_walk4.png
 robot_walk0.png	 robot_fall.png	 robot_walk3.png
 robot_climb0.png	 robot_climb1.png	 robot_jump.png
 robot_walk7.png	 robot_walk2.png	 robot_walk1.png
 robot_walk5.png		

Table 13: :resources:images/spritesheets/

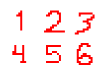
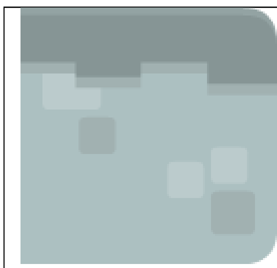
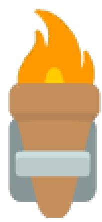
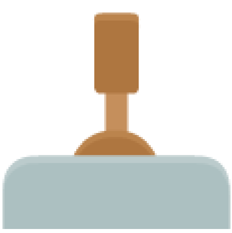
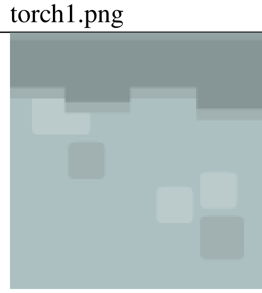
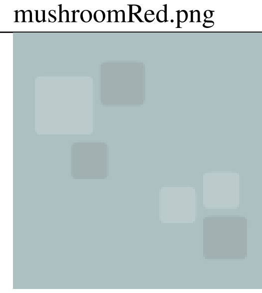
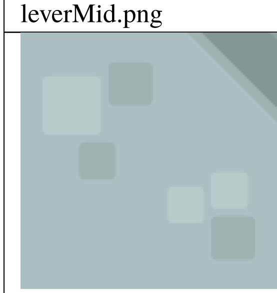
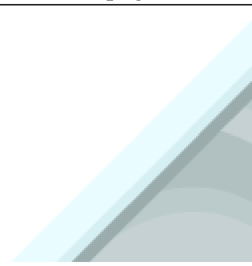
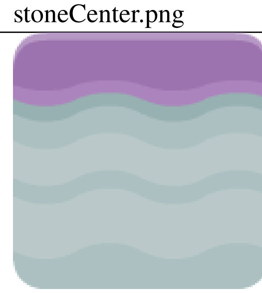
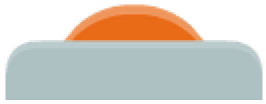
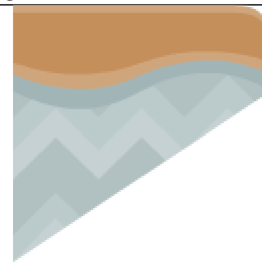
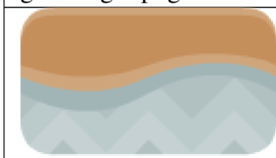
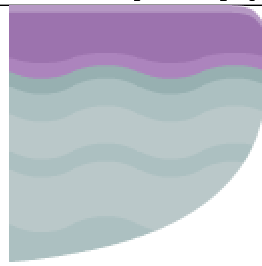
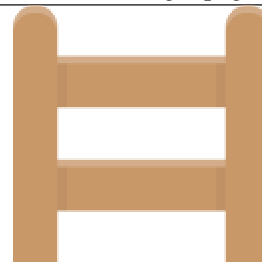
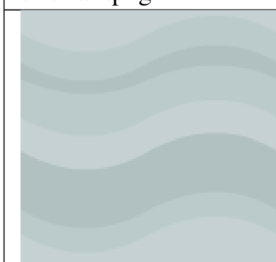
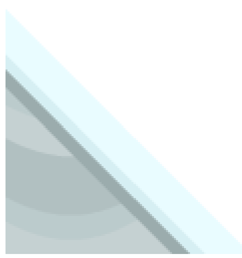
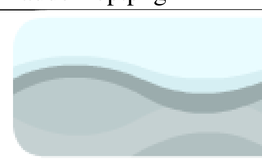
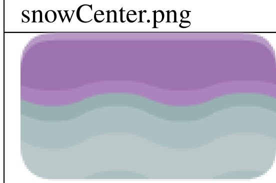
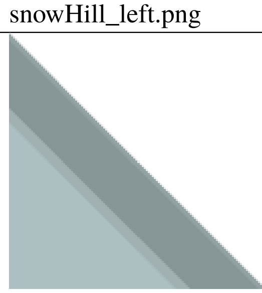
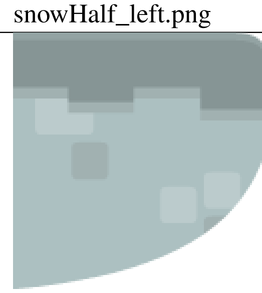
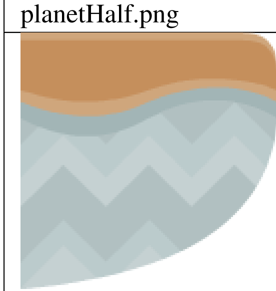
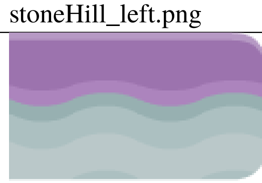
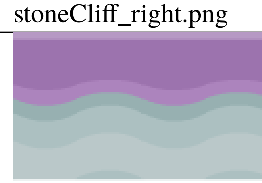
 explosion.png	 tiles.png	 number_sheet.png
codepage_437.png		

Table 14: :resources:images/tiles/

 stoneRight.png	 torch1.png	 mushroomRed.png
 leverMid.png	 stoneMid.png	 stoneCenter.png
 stoneCorner_left.png	 snowHill_right.png	 planet.png

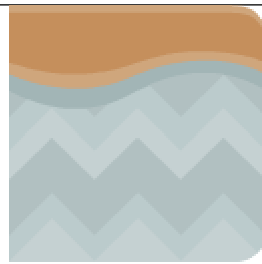
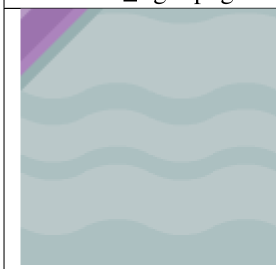
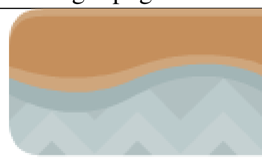
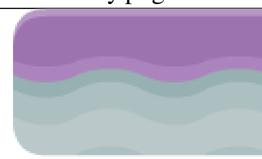
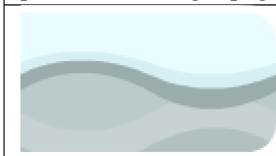
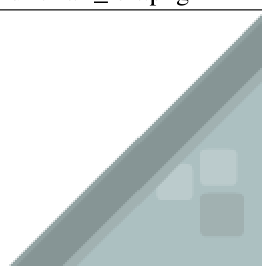
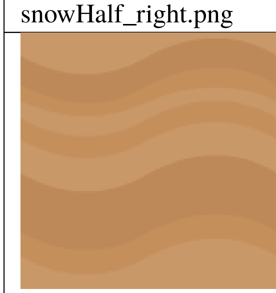
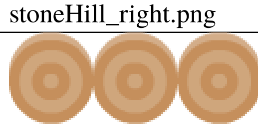
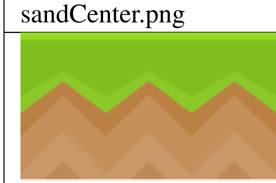
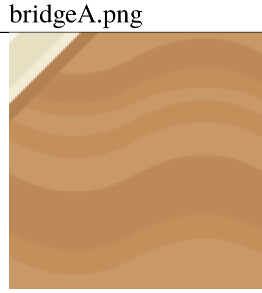
continues on next page

Table 14 – continued from previous page

 grassRight.png	 switchRed_pressed.png	 dirtCliffAlt_right.png
 dirtHalf.png	 planetCliff_right.png	 ladderTop.png
 snowCenter.png	 snowHill_left.png	 snowHalf_left.png
 planetHalf.png	 stoneHill_left.png	 stoneCliff_right.png
 dirtCliff_right.png	 planetHalf_right.png	 planetHalf_mid.png

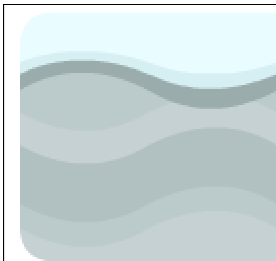
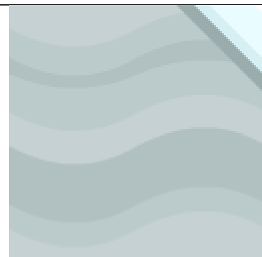
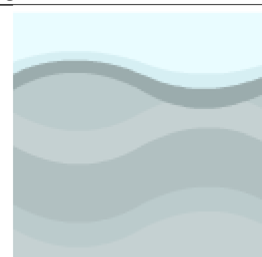
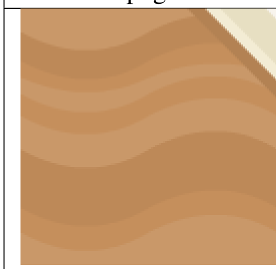
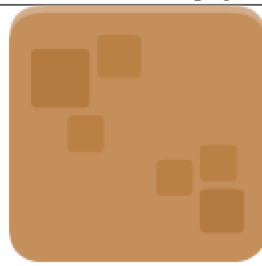
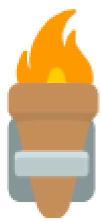
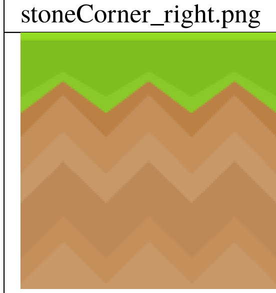
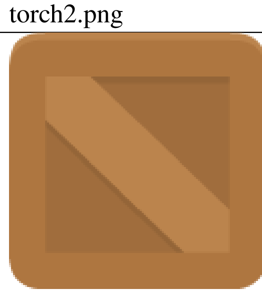
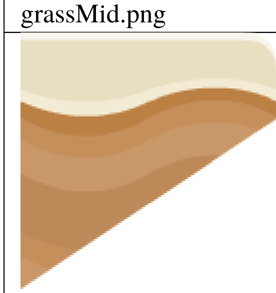
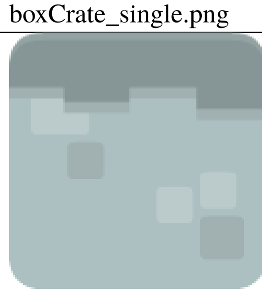
continues on next page

Table 14 – continued from previous page

 stoneHalf_right.png	 dirtRight.png	 brickGrey.png
 planetCorner_right.png	 dirtHalf_left.png	 planetHalf_left.png
 snowHalf_right.png	 stoneHill_right.png	 stoneHalf_mid.png
 sandCenter.png	 bridgeA.png	 plantPurple.png
 grassHalf_mid.png	 sandCorner_right.png	 grass_sprout.png

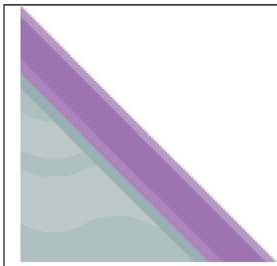
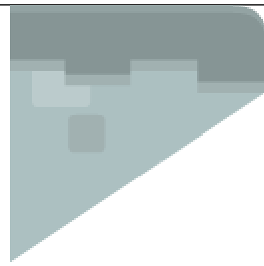
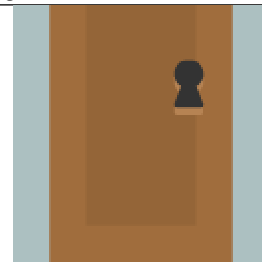
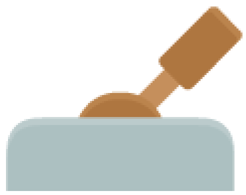
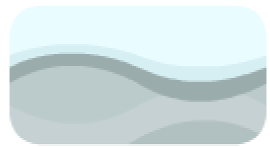
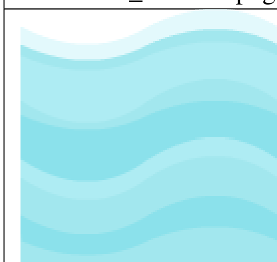
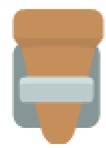
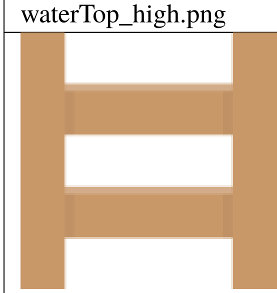
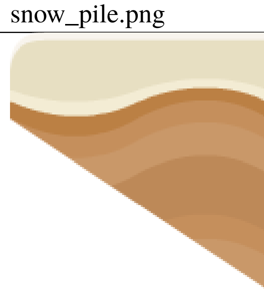
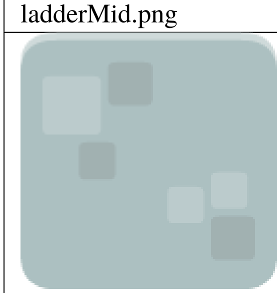
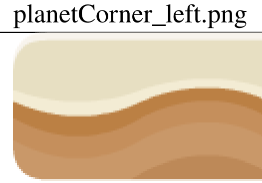
continues on next page

Table 14 – continued from previous page

 snowLeft.png	 snowCorner_left.png	 snowMid.png
 sandCorner_left.png	 brickBrown.png	 signRight.png
 stoneCorner_right.png	 torch2.png	 grassCliff_left.png
 grassMid.png	 boxCrate_single.png	 grassCorner_right.png
 sandCliffAlt_right.png	 stone.png	 bush.png

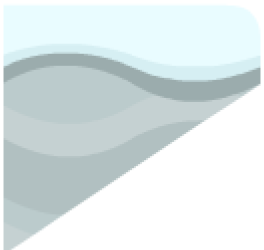
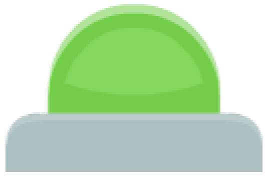
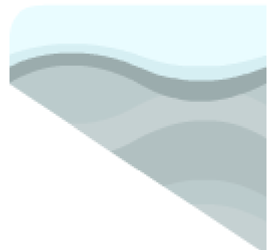
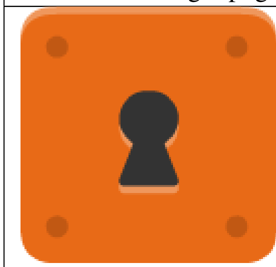
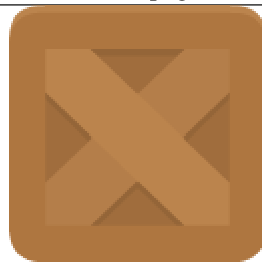
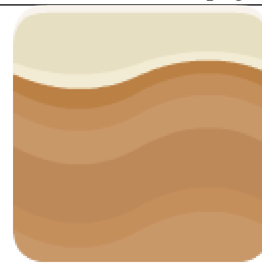
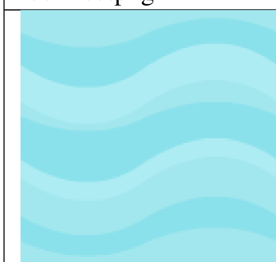
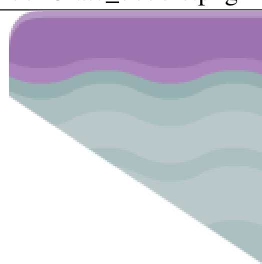
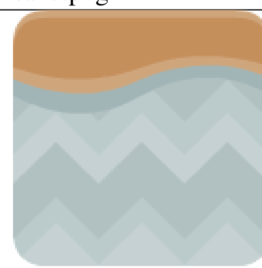
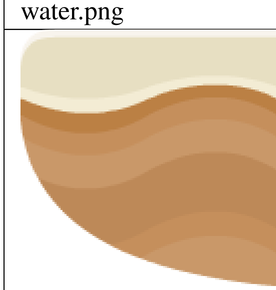
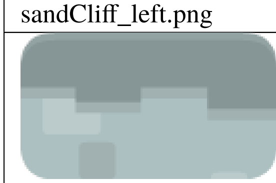
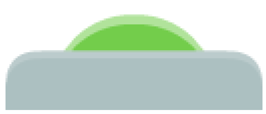
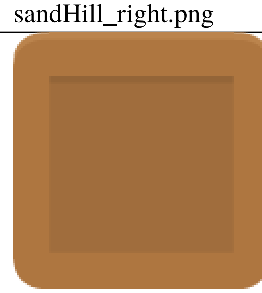
continues on next page

Table 14 – continued from previous page

 planetHill_left.png	 stoneCliffAlt_right.png	 doorClosed_mid.png
 dirtCenter_rounded.png	 leverRight.png	 snowHalf.png
 waterTop_high.png	snow_pile.png	 torchOff.png
 ladderMid.png	 sandCliffAlt_left.png	 planetCorner_left.png
 stoneCenter_rounded.png	 sandCliff_right.png	 sandHalf_left.png

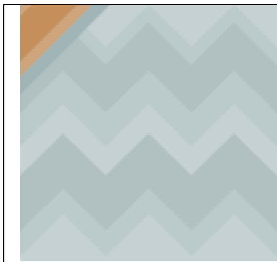
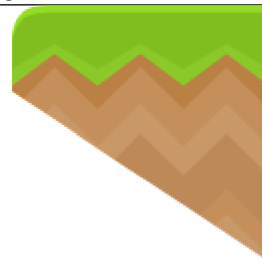
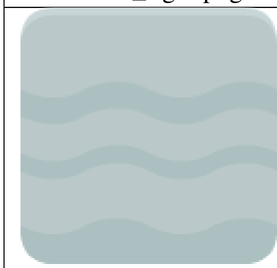
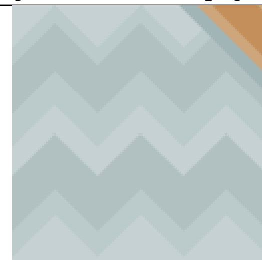
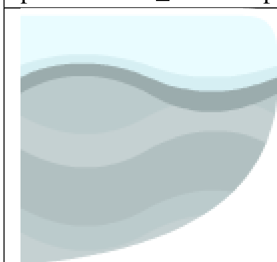
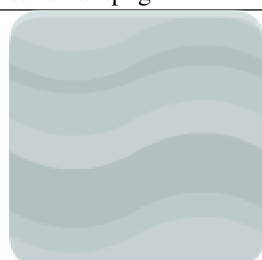
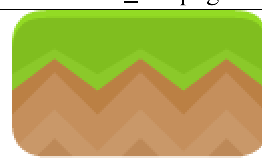
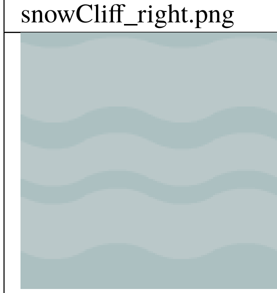
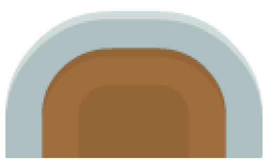
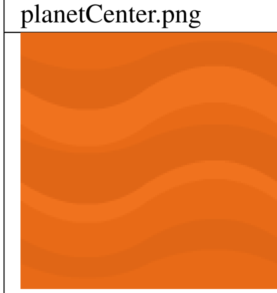
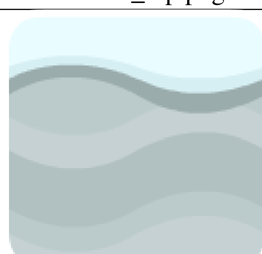
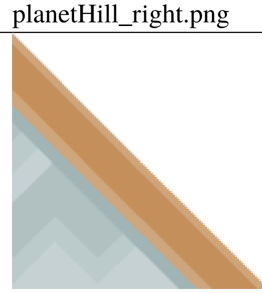
continues on next page

Table 14 – continued from previous page

 snowCliffAlt_right.png	 switchGreen.png	 snowCliffAlt_left.png
 lockRed.png	 boxCrate_double.png	 sand.png
 water.png	 planetCliffAlt_left.png	 dirt.png
 sandCliff_left.png	 lavaTop_high.png	 sandHill_right.png
 stoneHalf.png	 switchGreen_pressed.png	 boxCrate.png

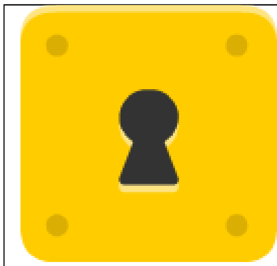
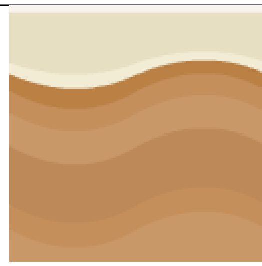
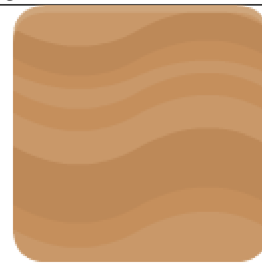
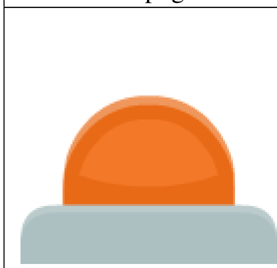
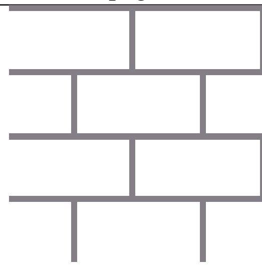
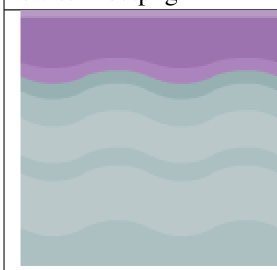
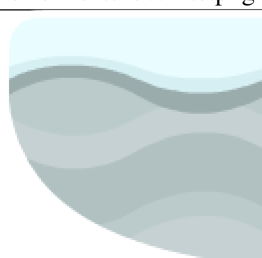
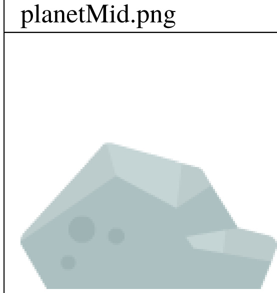
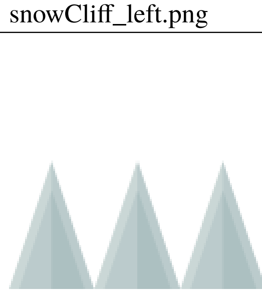
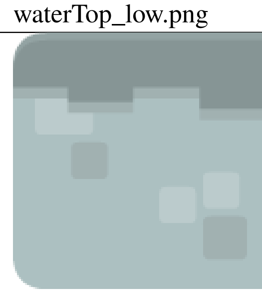
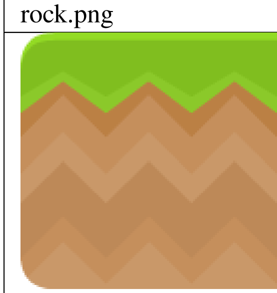
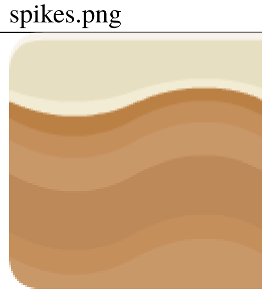
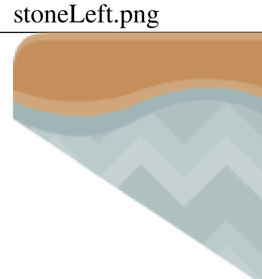
continues on next page

Table 14 – continued from previous page

 dirtCorner_right.png	 grassCorner_left.png	 grassCliffAlt_left.png
 planetCenter_rounded.png	 sandHalf.png	 dirtCorner_left.png
 snowCliff_right.png	 snowCenter_rounded.png	 grassHalf.png
 planetCenter.png	 doorClosed_top.png	 planetHill_right.png
 lava.png	 snow.png	 dirtHill_left.png

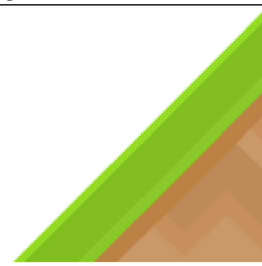
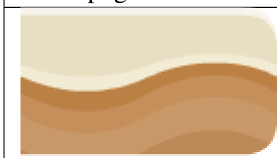
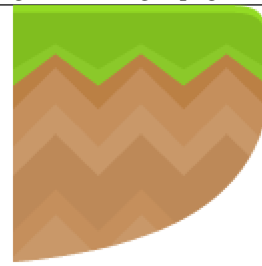
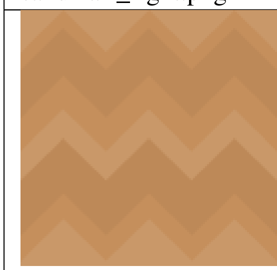
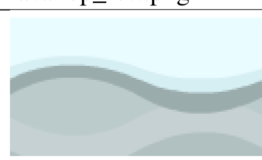
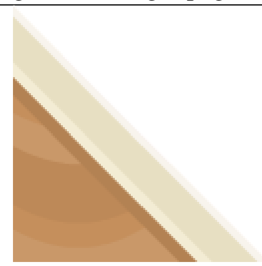
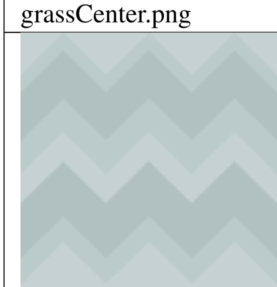
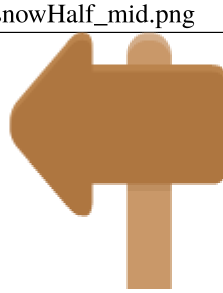
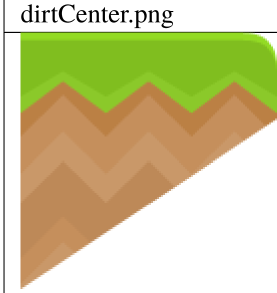
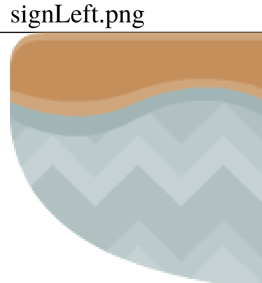
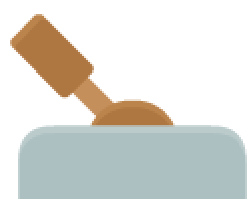
continues on next page

Table 14 – continued from previous page

 lockYellow.png	 sandMid.png	 sandCenter_rounded.png
 switchRed.png	 brickTextureWhite.png	 grassCenter_round.png
 planetMid.png	 snowCliff_left.png	waterTop_low.png
 rock.png	 spikes.png	 stoneLeft.png
 grassLeft.png	 sandLeft.png	 dirtCliffAlt_left.png

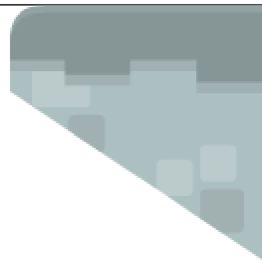
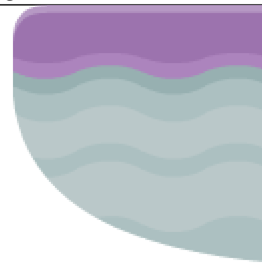
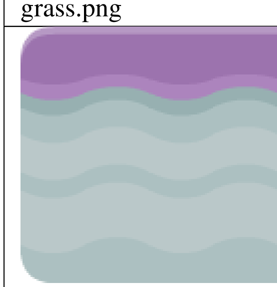
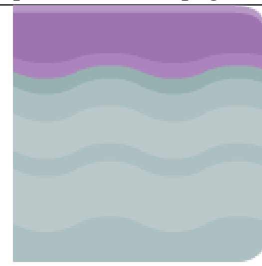
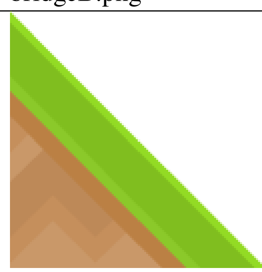
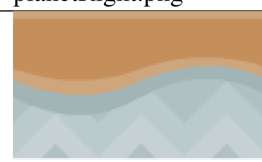
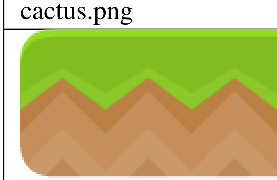
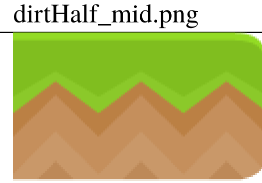
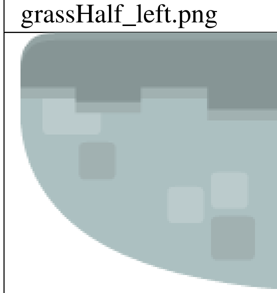
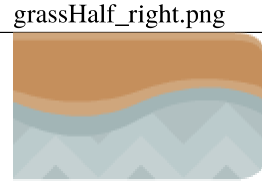
continues on next page

Table 14 – continued from previous page

 bomb.png	 stoneHalf_left.png	 grassHill_right.png
 sandHalf_right.png	 lavaTop_low.png	 grassCliff_right.png
 grassCenter.png	 snowHalf_mid.png	 sandHill_left.png
 dirtCenter.png	 signLeft.png	 dirtLeft.png
 grassCliffAlt_right.png	 dirtCliff_left.png	 leverLeft.png

continues on next page

Table 14 – continued from previous page

 grass.png	 stoneCliffAlt_left.png	 planetCliff_left.png
 planetLeft.png	 bridgeB.png	 planetRight.png
 cactus.png	 grassHill_left.png	 dirtHalf_mid.png
 grassHalf_left.png	 dirtMid.png	 grassHalf_right.png
 stoneCliff_left.png	 snowCorner_right.png	 dirtHalf_right.png

continues on next page

Table 14 – continued from previous page

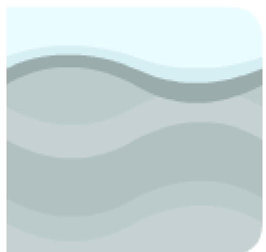
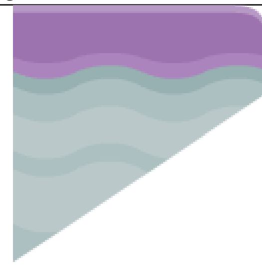
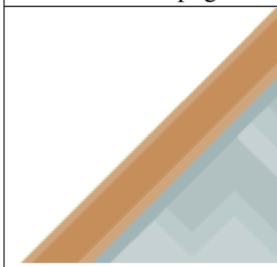
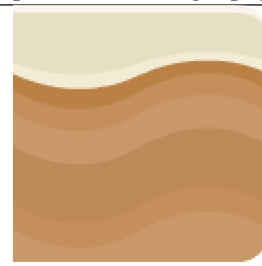
 sandHalf_mid.png	 snowRight.png	 planetCliffAlt_right.png
 dirtHill_right.png	 signExit.png	 sandRight.png

Table 15: :resources:images/pinball/

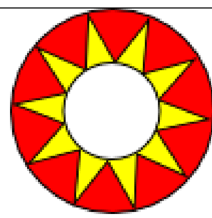
pool_cue_ball.png	 bumper.png	

Table 16: :resources:images/test_textures/

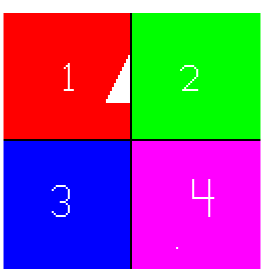
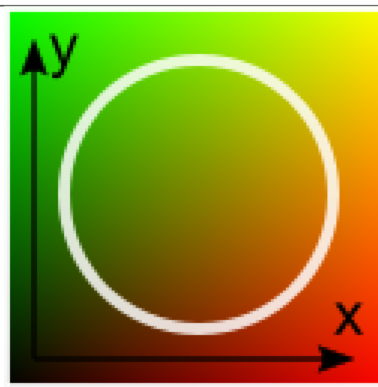
 test_texture.png	 xy_square.png	

Table 17: :resources:images/backgrounds/

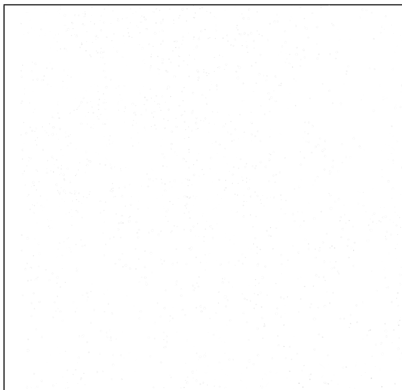
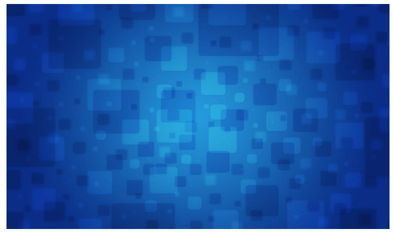
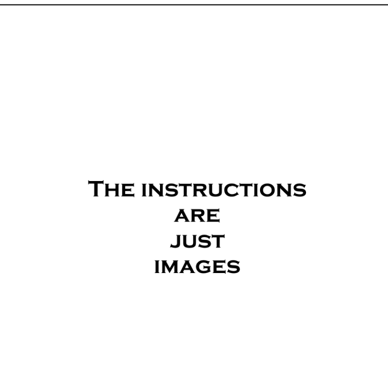
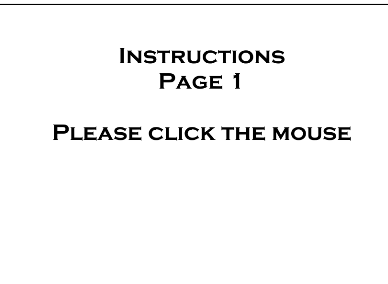
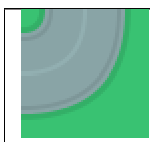
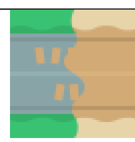
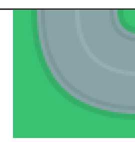
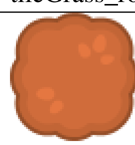
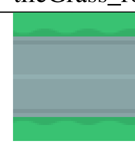
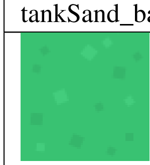
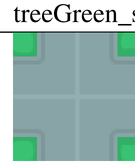
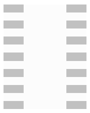
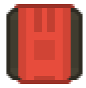
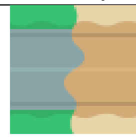
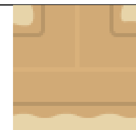
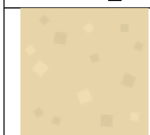
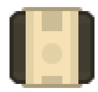
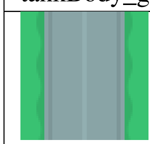
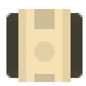
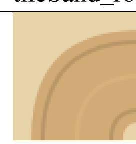
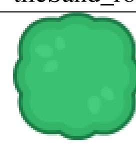
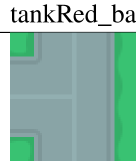
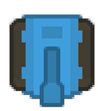
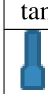
 stars.png	 abstract_1.jpg	 instructions_1.png
 abstract_2.jpg	 instructions_0.png	

Table 18: :resources:images/topdown_tanks/

 tileGrass_roadCornerUL.png	 tileGrass_roadTransitionE_dirt.png	 tileGrass_roadCornerUR.png
 tankSand_barrel3.png	 treeBrown_large.png	 tileGrass_roadEast.png
 tileGrass2.png	 tankBody_dark_outline.png	 treeGreen_small.png
 tankBody_dark.png	 tankGreen_barrel1_outline.png	 tileGrass_roadCrossing.png

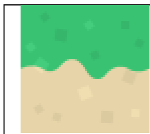
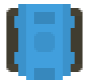
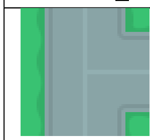
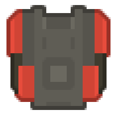
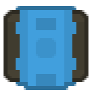
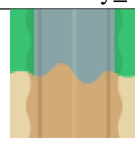
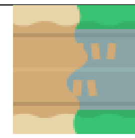
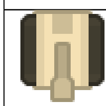
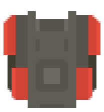
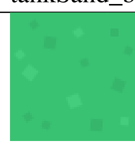
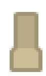
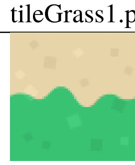
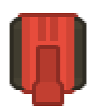
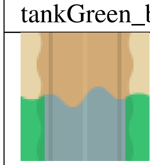
continues on next page

Table 18 – continued from previous page

 tracksLarge.png	 tankBody_red_outline.png	 tileSand_roadCrossingRound.png
 tankRed_barrel2.png	 tileGrass_roadTransitionE.png	 tileSand_roadSplitN.png
 tileSand1.png	 tankDark_barrel2_outline.png	 tankBody_sand_outline.png
 tankBody_green.png	 tankRed_barrel1_outline.png	 tileSand_roadEast.png
 tileGrass_roadNorth.png	 tankBody_sand.png	 tileSand_roadCornerLR.png
 tankGreen_barrel3_outline.png	 tankDark_barrel3_outline.png	 treeGreen_large.png
 tankBody_darkLarge_outline.png	 tankRed_barrel2_outline.png	 tileSand_roadCrossing.png
 tileSand_roadCornerLL.png	 tileGrass_roadSplitW.png	 tank_blue.png
 tankBlue_barrel2.png	 tankBody_darkLarge.png	 tankDark_barrel1.png
 tankBlue_barrel2_outline.png	 tankSand_barrel1.png	 tankBlue_barrel3_outline.png

continues on next page

Table 18 – continued from previous page

 tileGrass_transitionS.png	 tankBody_blue.png	 tracksSmall.png
 tileGrass_roadSplitE.png	 tankBody_bigRed_outline.png	 tankSand_barrel2.png
 tankBody_blue_outline.png	 tileGrass_roadTransitionS.png	 tile- Grass_roadTransitionW_dirt.png
 tank_sand.png	 tankRed_barrel3.png	 tankRed_barrel3_outline.png
 tankGreen_barrel2.png	 tankBlue_barrel1_outline.png	 tileSand_roadCornerUL.png
 tankBlue_barrel3.png	 tankSand_barrel3_outline.png	 tileGrass_roadTransitionW.png
 tankBody_bigRed.png	 tileGrass1.png	 tankSand_barrel1_outline.png
 tankGreen_barrel3.png	 tileGrass_transitionN.png	 tank_red.png
 tileGrass_roadTransitionN.png	 tankRed_barrel1.png	 tileGrass_roadCornerLR.png

continues on next page

Table 18 – continued from previous page

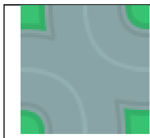
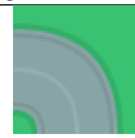
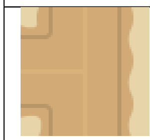
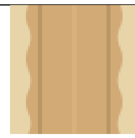
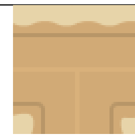
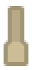
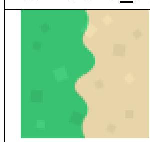
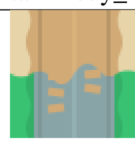
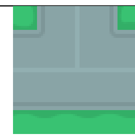
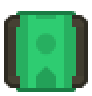
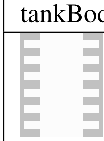
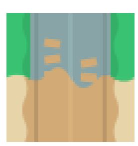
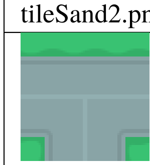
 tileGrass_roadCrossingRound.png	 tankBlue_barrel1.png	 tileGrass_roadCornerLL.png
 tileSand_roadSplitW.png	 tileSand_roadNorth.png	 tileSand_roadSplitS.png
 tankSand_barrel2_outline.png	 tankBody_huge.png	 tileGrass_transitionW.png
 tileGrass_transitionE.png	 tileGrass_roadTransitionN_dirt.png	 tileGrass_roadSplitN.png
 tank_dark.png	 tileSand_roadSplitE.png	 treeBrown_small.png
 tankBody_green_outline.png	 tileSand_roadCornerUR.png	 tankGreen_barrel2_outline.png
 tracksDouble.png	 tankBody_red.png	 tankDark_barrel2.png
 tileSand2.png	 tileGrass_roadTransitionS_dirt.png	 tankBody_huge_outline.png
 tileGrass_roadSplitS.png	 tankDark_barrel3.png	 tankGreen_barrel1.png
 tank_green.png	 tankDark_barrel1_outline.png	

Table 19: :resources:images/cards/

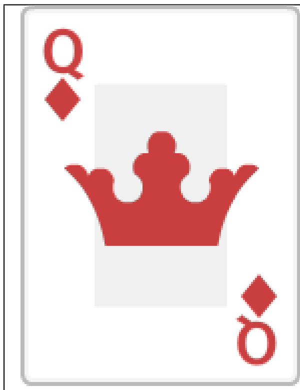
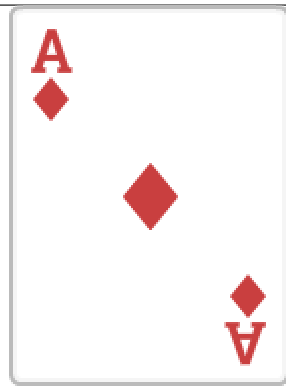
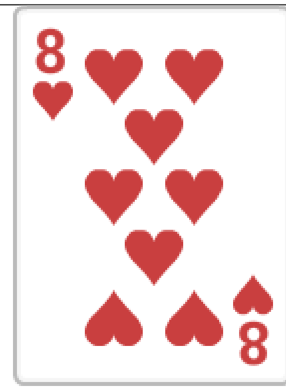
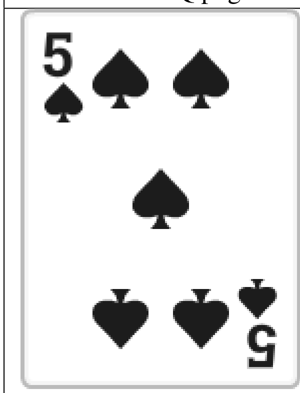
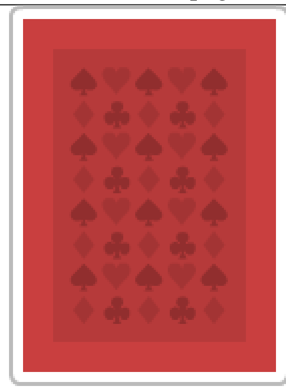
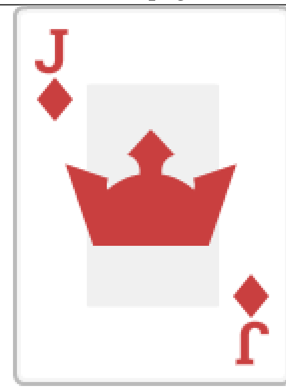
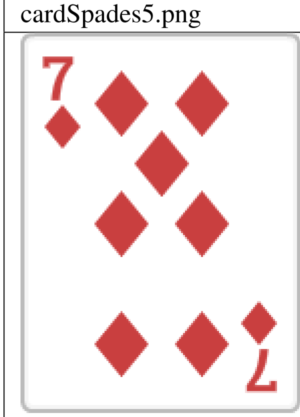
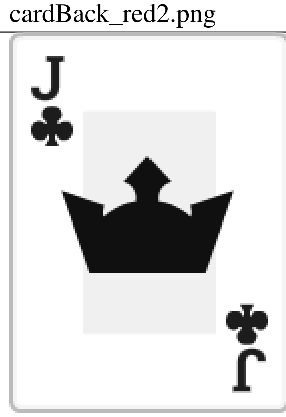
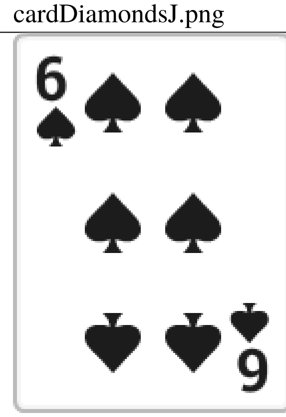
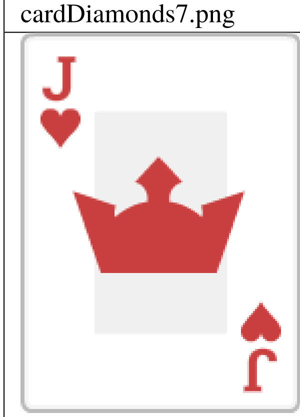
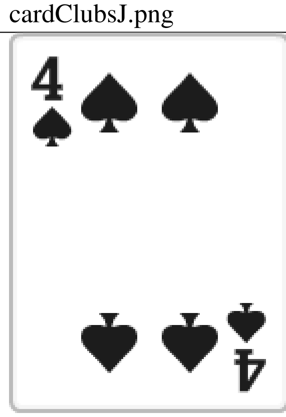
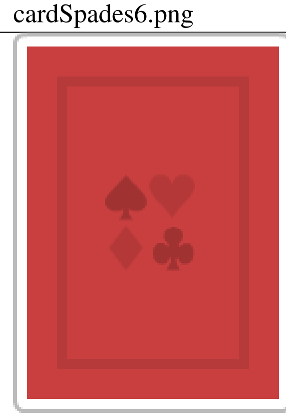
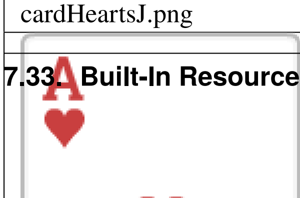
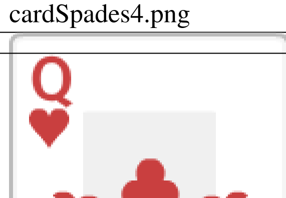
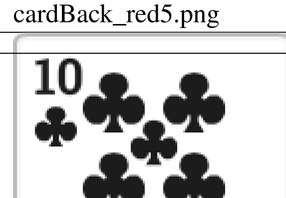
 cardDiamondsQ.png	 cardDiamondsA.png	 cardHearts8.png
 cardSpades5.png	 cardBack_red2.png	 cardDiamondsJ.png
 cardDiamonds7.png	 cardClubsJ.png	 cardSpades6.png
 cardHeartsJ.png	 cardSpades4.png	 cardBack_red5.png
		

Table 20: :resources:images/enemies/

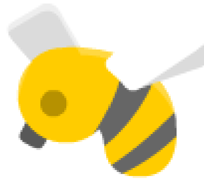
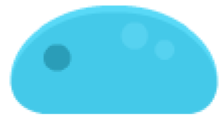
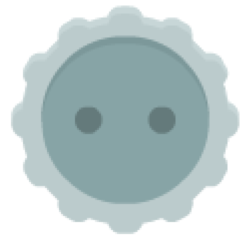
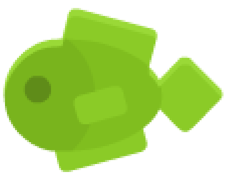
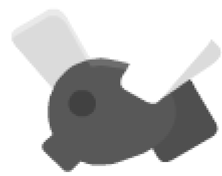
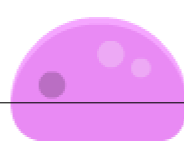
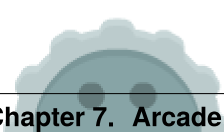
 wormGreen_move.png	 bee.png	 slimeBlue_move.png
 ladybug.png	 wormGreen_dead.png	 saw.png
 fishGreen.png	 wormGreen.png	 fly.png
 wormPink.png	 mouse.png	 slimeBlue.png
 slimeGreen.png	 frog.png	 frog_move.png
 fishPink.png	 slimePurple.png	 sawHalf.png

Table 21: :resources:images/cybercity_background/

 far-buildings.png	 back-buildings.png	 foreground.png

Table 22: :resources:images/alien/

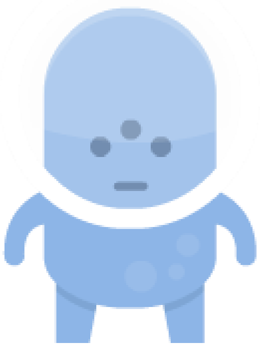
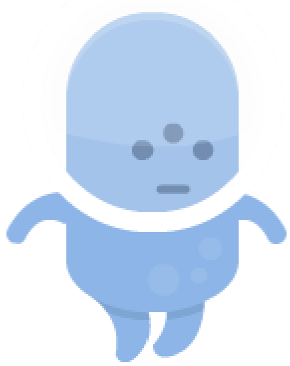
 alienBlue_front.png	 alienBlue_climb1.png	 alienBlue_walk1.png
 alienBlue_jump.png	 alienBlue_walk2.png	 alienBlue_climb2.png

Table 23: :resources:tiled_maps/

test_map_1.json	map.json	pymunk_test_map.json
map2_level_1.json	map_with_ladders.json	level_1.json
test_map_2.json	items.json	standard_tileset.json
test_map_3.json	grass.json	test_map_5.json
dirt.json	test_map_6.json	test_objects.json
test_map_7.json	map2_level_2.json	level_2.json
maps.tiled-project	more_tiles.json	spritesheet.json

Table 24: :resources:sounds/

Table 25: :resources:music/

Table 26: :resources:onscreen_controls/shaded_dark/

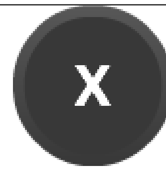
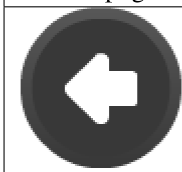
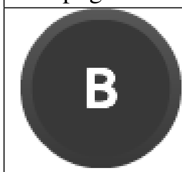
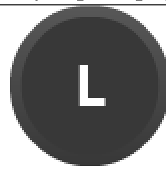
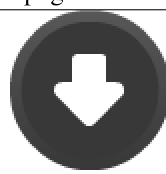
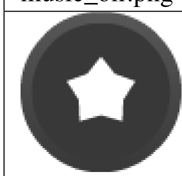
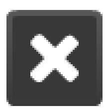
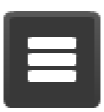
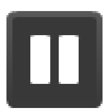
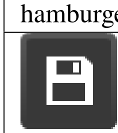
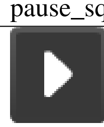
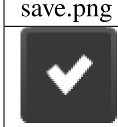
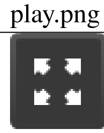
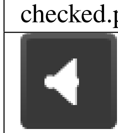
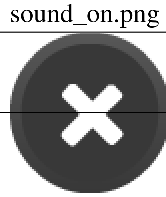
 wrench.png	 start.png	 x.png
 left.png	 key_square.png	 star_square.png
 b.png	 l.png	 y.png
 music_off.png	 down.png	 up.png
 star_round.png	 close.png	 r.png
 hamburger.png	 pause_square.png	 key_round.png
 save.png	 play.png	 search.png
 checked.png	 expand.png	 back.png
 sound_off.png	 sound_on.png	 music_on.png
 pause.png	 cancel.png	 select.png

Table 27: :resources:onscreen_controls/shaded_light/

wrench.png	start.png	key.png
x.png	left.png	star_square.png
b.png	l.png	y.png
music_off.png	down.png	up.png
star_round.png	close.png	r.png
hamburger.png	pause_square.png	key_round.png
save.png	play.png	search.png
checked.png	expand.png	back.png
sound_off.png	sound_on.png	music_on.png
pause.png	cancel.png	select.png

Table 28: :resources:onscreen_controls/flat_dark/

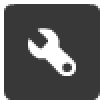
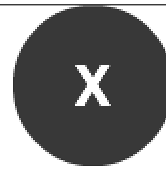
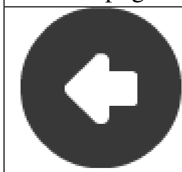
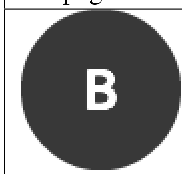
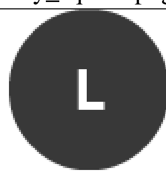
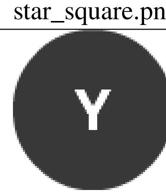
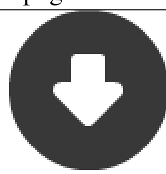
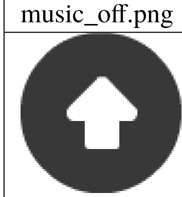
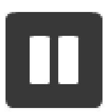
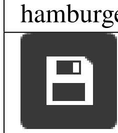
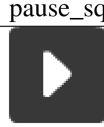
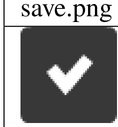
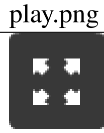
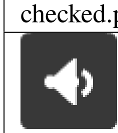
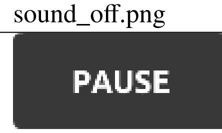
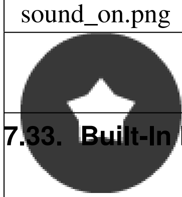
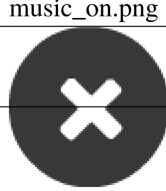
 wrench.png	 start.png	 x.png
 left.png	 key_square.png	 star_square.png
 b.png	 l.png	 y.png
 music_off.png	 down.png	 flatDark20.png
 up.png	 close.png	 r.png
 hamburger.png	 pause_square.png	 key_round.png
 save.png	 play.png	 search.png
 checked.png	 expand.png	 sound_off.png
 sound_on.png	 music_on.png	 pause.png
 star.png	 cancel.png	 select.png

Table 29: :resources:onscreen_controls/flat_light/

wrench.png	start.png	x.png
left.png	key_square.png	star_square.png
b.png	l.png	y.png
music_off.png	down.png	up.png
star_round.png	close.png	r.png
hamburger.png	pause_square.png	key_round.png
save.png	play.png	search.png
checked.png	expand.png	back.png
sound_off.png	sound_on.png	music_on.png
pause.png	cancel.png	select.png

Table 30: :resources:gui_basic_assets/

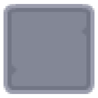
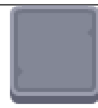
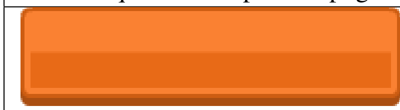
 button_square_blue_pressed.png	 button_square_blue.png	 red_button_press.png
 red_button_hover.png	 red_button_normal.png	

Table 31: :resources:gui_basic_assets/items/

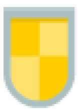
 shield_gold.png	 sword_gold.png	

Table 32: :resources:gui_basic_assets/icons/

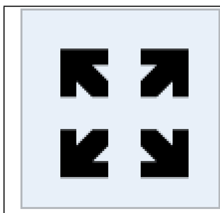
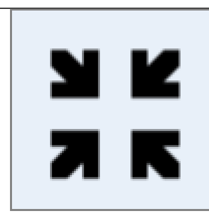
 larger.png	 smaller.png	

Table 33: :resources:gui_basic_assets/window/

 grey_panel.png		

7.34 Working with the Keyboard

7.34.1 Modifiers

The modifiers that are held down when the event is generated are combined in a bitwise fashion and provided in the modifiers parameter. The modifier constants defined in arcade.key are:

```
MOD_SHIFT
MOD_CTRL
```

(continues on next page)

(continued from previous page)

```
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.
```

For example, to test if the shift key is held down:

```
if modifiers & MOD_SHIFT:
    pass
```

Unlike the corresponding key symbols, it is not possible to determine whether the left or right modifier is held down (though you could emulate this behavior by keeping track of the key states yourself).

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 there is a link to the full source code. The tutorial is divided into these parts:

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 Instructions*.

I highly recommend using the free community edition of PyCharm as an editor. If you do, see *Install Arcade with PyCharm and a Virtual Environment*.

8.1.2 Open a Window

The example below opens up a blank window. Set up a project and get the code below working. (It is also in the zip file as `01_open_window.py`.)

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

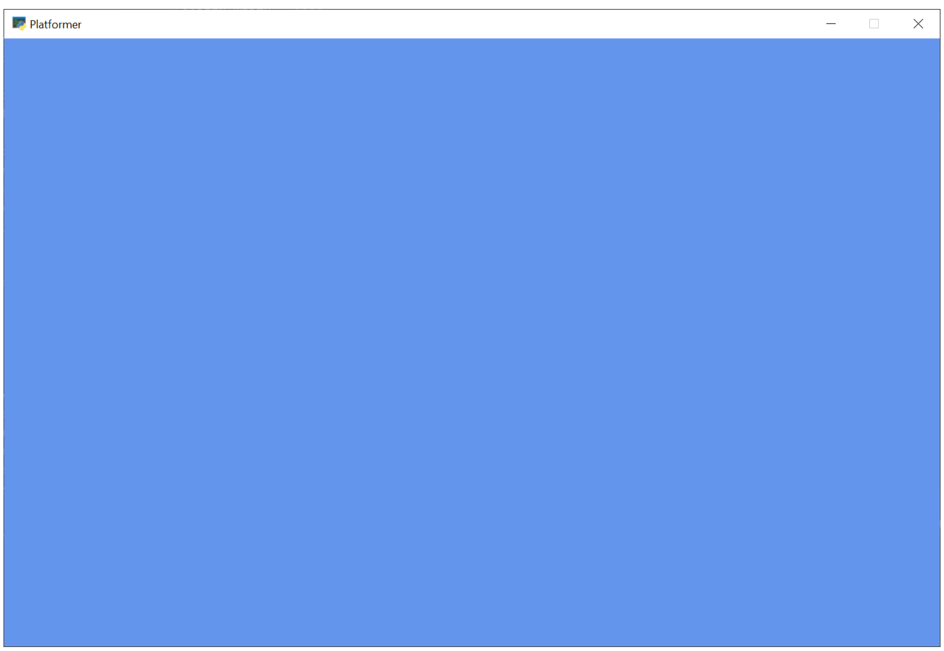
```
"""
Platformer Game
"""
import arcade

# Constants
SCREEN_WIDTH = 1000
SCREEN_HEIGHT = 650
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)

        arcade.set_background_color(arcade.csscolor.CORNFLOWER_BLUE)

```

(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."""

        self.clear()
        # Code to draw the screen goes here


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


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

You should end up with a window like this:



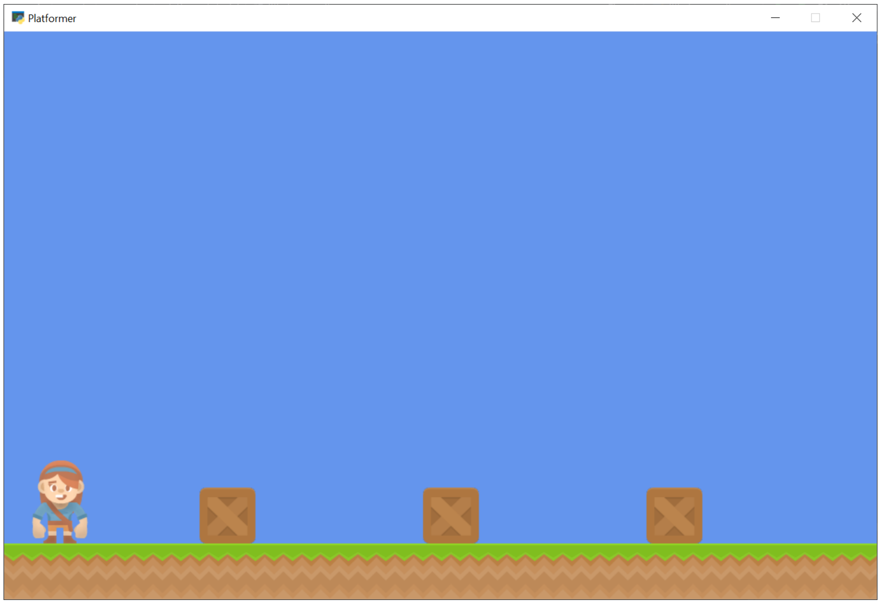
Once you get the code working, figure out how to adjust the code so you can:

- Change the screen size
- 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.

8.2 Step 2 - Add Sprites

Our next step is to add some sprites, which are graphics we can see and interact with on the screen.



8.2.1 Setup vs. Init

In the next code example, `02_draw_sprites`, we'll have both an `__init__` method and a `setup`.

The `__init__` creates the variables. The variables are set to values such as 0 or `None`. The `setup` actually creates the object instances, such as graphical sprites.

I often get the very reasonable question, "Why have two methods? Why not just put everything into `__init__`? Seems like we are doing twice the work." Here's why. With a `setup` method split out, later on we can easily add "restart/play again" functionality to the game. A simple call to `setup` will reset everything. Later, we can expand our game with different levels, and have functions such as `setup_level_1` and `setup_level_2`.

8.2.2 Sprite Lists

Sprites are managed in lists. The `SpriteList` class optimizes drawing, movement, and collision detection.

We are using three logical groups in our game. A `player_list` for the player. A `wall_list` for walls we can't move through.

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

Sprite lists have an option to use something called "spatial hashing." Spatial hashing speeds the time it takes to find collisions, but increases the time it takes to move a sprite. Since I don't expect most of my walls to move, I'll turn on spatial hashing for these lists. My player moves around a lot, so I'll leave it off for her.

8.2.3 Add Sprites to the Game

To create sprites we'll use the `arcade.Sprite` class. We can create an instance of the sprite class with code like this:

```
self.player_sprite = arcade.Sprite("images/player_1/player_stand.png", CHARACTER_SCALING)
```

The first parameter is a string or path to the image you want it to load. An optional second parameter will scale the sprite up or down. If the second parameter (in this case a constant `CHARACTER_SCALING`) is set to 0.5, and the the sprite is 128x128, then both width and height will be scaled down 50% for a 64x64 sprite.

Built-in Resources

The arcade library has a few built-in *Built-In Resources* so we can run examples without downloading images. If you see code samples where sprites are loaded beginning with "resources", that's what's being referenced.

Next, we need to tell *where* the sprite goes. You can use the attributes `center_x` and `center_y` to position the sprite. You can also use `top`, `bottom`, `left`, and `right` to get or set the sprites location by an edge instead of the center. You can also use `position` attribute to set both the x and y at the same time.

```
self.player_sprite.center_x = 64
self.player_sprite.center_y = 120
```

Finally, all instances of the `Sprite` class need to go in a `SpriteList` class.

```
self.player_list.append(self.player_sprite)
```

We manage groups of sprites by the list that they are in. In the example below there's a `wall_list` that will hold everything that the player character can't walk through. There's also a `player_list` which holds only the player.

- Documentation for the *`arcade.Sprite`* class
- Documentation for the *`arcade.SpriteList`* class

Notice that the code creates `Sprites` three ways:

- Creating a `Sprite` class, positioning it, adding it to the list
- Create a series of sprites in a loop

8.2.4 Source Code

Listing 2: 02_draw_sprites - Draw and Position Sprites

```
"""
Platformer Game
"""
import arcade

# Constants
SCREEN_WIDTH = 1000
SCREEN_HEIGHT = 650
SCREEN_TITLE = "Platformer"

# Constants used to scale our sprites from their original size
```

(continues on next page)

(continued from previous page)

```
CHARACTER_SCALING = 1
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)

        # These are 'lists' that keep track of our sprites. Each sprite should
        # go into a list.
        self.wall_list = None
        self.player_list = None

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

        arcade.set_background_color(arcade.csscolor.CORNFLOWER_BLUE)

    def setup(self):
        """Set up the game here. Call this function to restart the game."""
        # Create the Sprite lists
        self.player_list = arcade.SpriteList()
        self.wall_list = arcade.SpriteList(use_spatial_hash=True)

        # Set up the player, specifically placing it at these coordinates.
        image_source = ":resources:images/animated_characters/female_adventurer/
↪femaleAdventurer_idle.png"
        self.player_sprite = arcade.Sprite(image_source, CHARACTER_SCALING)
        self.player_sprite.center_x = 64
        self.player_sprite.center_y = 128
        self.player_list.append(self.player_sprite)

        # 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", 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
```

(continues on next page)

(continued from previous page)

```
            wall = arcade.Sprite(
                ":resources:images/tiles/boxCrate_double.png", TILE_SCALING
            )
            wall.position = coordinate
            self.wall_list.append(wall)

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

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

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


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


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

Running this code should result in some sprites drawn on the screen, as shown in the image at the top of this page.

Note: Once the code example is up and working, try adjusting the code for the following:

- Adjust the code and try putting sprites in new positions.
- Use different images for sprites (see *Built-In Resources* for the build-in images, or use your own images.)
- Practice placing individually, via a loop, and by coordinates in a list.

8.3 Step 3 - Scene Object

Next we will add a Scene to our game. A Scene is a tool to manage a number of different SpriteLists by assigning each one a name, and maintaining a draw order.

SpriteLists can be drawn directly like we saw in step 2 of this tutorial, but a Scene can be helpful to handle a lot of different lists at once and being able to draw them all with one call to the scene.

To start with we will remove our sprite lists from the `__init__` function, and replace them with a scene object.

Listing 3: 03_scene_object.py - Scene Object Definition

```
    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)

        # Our Scene Object
        self.scene = None

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

        arcade.set_background_color(arcade.csscolor.CORNFLOWER_BLUE)
```

Next we will initialize the scene object in the `setup` function and then add the SpriteLists to it instead of creating new SpriteList objects directly.

Then instead of appending the Sprites to the SpriteLists directly, we can add them to the Scene and specify by name what SpriteList we want them added to.

Listing 4: 03_scene_object.py - Add SpriteLists to the Scene

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

        # Initialize Scene
        self.scene = arcade.Scene()

        # Create the Sprite lists
        self.scene.add_sprite_list("Player")
        self.scene.add_sprite_list("Walls", use_spatial_hash=True)

        # Set up the player, specifically placing it at these coordinates.
        image_source = ":resources:images/animated_characters/female_adventurer/
↪femaleAdventurer_idle.png"
        self.player_sprite = arcade.Sprite(image_source, CHARACTER_SCALING)
        self.player_sprite.center_x = 64
        self.player_sprite.center_y = 128
        self.scene.add_sprite("Player", self.player_sprite)

        # 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", 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", TILE_SCALING
            )
```

(continues on next page)

(continued from previous page)

```
            wall.position = coordinate
            self.scene.add_sprite("Walls", wall)
```

Lastly in our `on_draw` function we can draw the scene.

Listing 5: 03_scene_object.py - Draw the Scene

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

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

        # Draw our Scene
        self.scene.draw()
```

8.3.1 Source Code

Listing 6: 03_scene_object - Scene Object

```
"""
Platformer Game
"""
import arcade

# Constants
SCREEN_WIDTH = 1000
SCREEN_HEIGHT = 650
SCREEN_TITLE = "Platformer"

# Constants used to scale our sprites from their original size
CHARACTER_SCALING = 1
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)

        # Our Scene Object
        self.scene = None

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

```

(continues on next page)

(continued from previous page)

```
        arcade.set_background_color(arcade.csscolor.CORNFLOWER_BLUE)

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

        # Initialize Scene
        self.scene = arcade.Scene()

        # Create the Sprite lists
        self.scene.add_sprite_list("Player")
        self.scene.add_sprite_list("Walls", use_spatial_hash=True)

        # Set up the player, specifically placing it at these coordinates.
        image_source = ":resources:images/animated_characters/female_adventurer/
↪femaleAdventurer_idle.png"
        self.player_sprite = arcade.Sprite(image_source, CHARACTER_SCALING)
        self.player_sprite.center_x = 64
        self.player_sprite.center_y = 128
        self.scene.add_sprite("Player", self.player_sprite)

        # 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", 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", TILE_SCALING
            )
            wall.position = coordinate
            self.scene.add_sprite("Walls", wall)

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

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

        # Draw our Scene
        self.scene.draw()


def main():
    """Main function"""
```

(continues on next page)

(continued from previous page)

```
    window = MyGame()
    window.setup()
    arcade.run()


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

8.4 Step 4 - Add User Control

Now we need to be able to get the user to move around.

First, at the top of the program add a constant that controls how many pixels per update our character travels:

Listing 7: 04_user_control.py - Player Move Speed Constant

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

Next, at the end of our `setup` method, we need to create a physics engine that will move our player and keep her from running through walls. The `PhysicsEngineSimple` class takes two parameters: The moving sprite, and a list of sprites the moving sprite can't move through.

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.

Listing 8: 04_user_control.py - Create Physics Engine

```
        # Create the 'physics engine'
        self.physics_engine = arcade.PhysicsEngineSimple(
            self.player_sprite, self.scene.get_sprite_list("Walls")
        )
```

Each sprite has `center_x` and `center_y` attributes. Changing these will change the location of the sprite. (There are also attributes for top, bottom, left, right, and angle that will move the sprite.)

Each sprite has `change_x` and `change_y` variables. These can be used to hold the velocity that the sprite is moving with. We will adjust these based on what key the user hits. If the user hits the right arrow key we want a positive value for `change_x`. If the value is 5, it will move 5 pixels per frame.

In this case, when the user presses a key we'll change the sprites change x and y. The physics engine will look at that, and move the player unless she'll hit a wall.

Listing 9: 04_user_control.py - Handle key-down

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

        if key == arcade.key.UP or key == arcade.key.W:
```

(continues on next page)

(continued from previous page)

```
            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
```

On releasing the key, we'll put our speed back to zero.

Listing 10: 04_user_control.py - Handle key-up

```
    def on_key_release(self, key, modifiers):
        """Called when the user releases a key."""

        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
```

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.

Our `on_update` method is called about 60 times per second. We'll ask the physics engine to move our player based on her `change_x` and `change_y`.

Listing 11: 04_user_control.py - Update the sprites

```
    def on_update(self, delta_time):
        """Movement and game logic"""

        # Move the player with the physics engine
        self.physics_engine.update()
```

8.4.1 Source Code

Listing 12: 04_user_control.py - User Control

```
"""
Platformer Game
"""
import arcade

# Constants
SCREEN_WIDTH = 1000
```

(continues on next page)

(continued from previous page)

```
SCREEN_HEIGHT = 650
SCREEN_TITLE = "Platformer"

# Constants used to scale our sprites from their original size
CHARACTER_SCALING = 1
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)

        # Our Scene Object
        self.scene = None

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

        # Our physics engine
        self.physics_engine = None

        arcade.set_background_color(arcade.csscolor.CORNFLOWER_BLUE)

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

        # Initialize Scene
        self.scene = arcade.Scene()

        # Set up the player, specifically placing it at these coordinates.
        image_source = ":resources:images/animated_characters/female_adventurer/
↪femaleAdventurer_idle.png"
        self.player_sprite = arcade.Sprite(image_source, CHARACTER_SCALING)
        self.player_sprite.center_x = 64
        self.player_sprite.center_y = 128
        self.scene.add_sprite("Player", self.player_sprite)

        # 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", TILE_SCALING)
            wall.center_x = x
            wall.center_y = 32
```

(continues on next page)

(continued from previous page)

```python
                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", TILE_SCALING
            )
            wall.position = coordinate
            self.scene.add_sprite("Walls", wall)

        # Create the 'physics engine'
        self.physics_engine = arcade.PhysicsEngineSimple(
            self.player_sprite, self.scene.get_sprite_list("Walls")
        )

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

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

        # Draw our Scene
        self.scene.draw()

    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 when the user releases a key."""

        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

```

(continues on next page)

(continued from previous page)

```
    def on_update(self, delta_time):
        """Movement and game logic"""

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


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


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

8.5 Step 5 - Add Gravity

The previous example great for top-down, 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.

Listing 13: 05_add_gravity.py - Add Gravity

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

At the end of the `setup` method, change the physics engine to `PhysicsEnginePlatformer` and include gravity as a parameter.

Listing 14: 05_add_gravity.py - Add Gravity

```
        # Create the 'physics engine'
        self.physics_engine = arcade.PhysicsEnginePlatformer(
            self.player_sprite, gravity_constant=GRAVITY, walls=self.scene["Walls"]
        )
```

We are sending our `SpriteList` for the things the player should collide with to the `walls` parameter of the the physics engine. As we'll see in later chapters, the platformer physics engine has a `platforms` and `walls` parameter. The difference between these is very important. Static non-moving spritelists should always be sent to the `walls` parameter, and moving sprites should be sent to the `platforms` parameter. Ensuring you do this will have extreme benefits to performance.

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.

We also see here some new syntax relating to our `Scene` object. You can access the scene like you would a Python dictionary in order to get your SpriteLists from it. There are multiple ways to access the SpriteLists within a Scene but this is the easiest and most straight forward. You could alternatively use `scene.get_sprite_list("My Layer")`.

Then, modify the key down and key up event handlers. We'll remove the up/down statements we had before, and make 'UP' jump when pressed.

Listing 15: 05_add_gravity.py - Add Gravity

```
    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
        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 when the user releases a key."""

        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
```

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 16: 05_add_gravity.py - Add Gravity

```
"""
Platformer Game
"""
import arcade

# Constants
SCREEN_WIDTH = 1000
SCREEN_HEIGHT = 650
SCREEN_TITLE = "Platformer"

# Constants used to scale our sprites from their original size
CHARACTER_SCALING = 1
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):
    """
```

(continues on next page)

(continued from previous page)

```python
    Main application class.
    """

    def __init__(self):

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

        # Our Scene Object
        self.scene = None

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

        # Our physics engine
        self.physics_engine = None

        arcade.set_background_color(arcade.csscolor.CORNFLOWER_BLUE)

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

        # Initialize Scene
        self.scene = arcade.Scene()

        # Set up the player, specifically placing it at these coordinates.
        image_source = ":resources:images/animated_characters/female_adventurer/
↪femaleAdventurer_idle.png"
        self.player_sprite = arcade.Sprite(image_source, CHARACTER_SCALING)
        self.player_sprite.center_x = 64
        self.player_sprite.center_y = 128
        self.scene.add_sprite("Player", self.player_sprite)

        # 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", 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", TILE_SCALING
            )
            wall.position = coordinate
            self.scene.add_sprite("Walls", wall)
```

(continues on next page)

(continued from previous page)

```python

        # Create the 'physics engine'
        self.physics_engine = arcade.PhysicsEnginePlatformer(
            self.player_sprite, gravity_constant=GRAVITY, walls=self.scene["Walls"]
        )

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

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

        # Draw our Scene
        self.scene.draw()

    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
        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 when the user releases a key."""

        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 on_update(self, delta_time):
        """Movement and game logic"""

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


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


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

8.6 Step 6 - Add a Camera

We can have our window be a small viewport into a much larger world by adding a camera to it.

First we need to create a new variable in our `__init__` method:

Listing 17: 06_camera.py - Create camera variable

```
        # A Camera that can be used for scrolling the screen
        self.camera = None
```

Next we can initialize the camera in the `setup` function:

Listing 18: 06_camera.py - Setup Camera

```
        # Set up the Camera
        self.camera = arcade.Camera(self.width, self.height)
```

Then to use our camera when drawing, we can activate it in our `on_draw` function:

Listing 19: 06_camera.py - Use camera when drawing

```
        # Activate our Camera
        self.camera.use()
```

Now at this point everything should be working the same, but the camera can do a lot more than this. We can use the `move` function of the camera to scroll it to a different position. We can use this functionality to keep the camera centered on the player:

We can create a function to calculate the coordinates for the center of our player relative to the screen, then move the camera to those. Then we can call that function in `on_update` to actually move it. The new position will be taken into account during the `use` function in `on_draw`

Listing 20: 06_camera.py - Center camera on player

```
    def center_camera_to_player(self):
        screen_center_x = self.player_sprite.center_x - (self.camera.viewport_width / 2)
        screen_center_y = self.player_sprite.center_y - (
            self.camera.viewport_height / 2
        )

        # Don't let camera travel past 0
        if screen_center_x < 0:
            screen_center_x = 0
        if screen_center_y < 0:
            screen_center_y = 0
        player_centered = screen_center_x, screen_center_y

        self.camera.move_to(player_centered)

    def on_update(self, delta_time):
        """Movement and game logic"""

        # Move the player with the physics engine
        self.physics_engine.update()
```

(continues on next page)

(continued from previous page)

```
        # Position the camera
        self.center_camera_to_player()
```

8.6.1 Source Code

Listing 21: Add a Camera

```
"""
Platformer Game
"""
import arcade

# Constants
SCREEN_WIDTH = 1000
SCREEN_HEIGHT = 650
SCREEN_TITLE = "Platformer"

# Constants used to scale our sprites from their original size
CHARACTER_SCALING = 1
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)

        # Our Scene Object
        self.scene = None

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

        # Our physics engine
        self.physics_engine = None

        # A Camera that can be used for scrolling the screen
        self.camera = None

```

(continues on next page)

(continued from previous page)

```python
        arcade.set_background_color(arcade.csscolor.CORNFLOWER_BLUE)

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

        # Set up the Camera
        self.camera = arcade.Camera(self.width, self.height)

        # Initialize Scene
        self.scene = arcade.Scene()

        # Create the Sprite lists
        self.scene.add_sprite_list("Player")
        self.scene.add_sprite_list("Walls", use_spatial_hash=True)

        # Set up the player, specifically placing it at these coordinates.
        image_source = ":resources:images/animated_characters/female_adventurer/
↪femaleAdventurer_idle.png"
        self.player_sprite = arcade.Sprite(image_source, CHARACTER_SCALING)
        self.player_sprite.center_x = 64
        self.player_sprite.center_y = 96
        self.scene.add_sprite("Player", self.player_sprite)

        # 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", 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", TILE_SCALING
            )
            wall.position = coordinate
            self.scene.add_sprite("Walls", wall)

        # Create the 'physics engine'
        self.physics_engine = arcade.PhysicsEnginePlatformer(
            self.player_sprite, gravity_constant=GRAVITY, walls=self.scene["Walls"]
        )

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

        # Clear the screen to the background color
```

(continues on next page)

(continued from previous page)

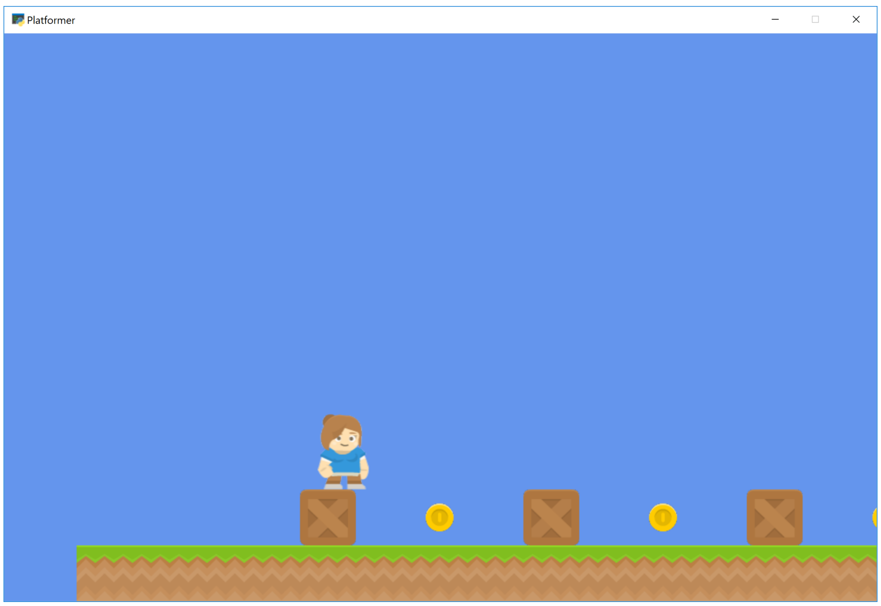
```
        self.clear()

        # Activate our Camera
        self.camera.use()

        # Draw our Scene
        self.scene.draw()

    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
        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 when the user releases a key."""

        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 center_camera_to_player(self):
        screen_center_x = self.player_sprite.center_x - (self.camera.viewport_width / 2)
        screen_center_y = self.player_sprite.center_y - (
            self.camera.viewport_height / 2
        )

        # Don't let camera travel past 0
        if screen_center_x < 0:
            screen_center_x = 0
        if screen_center_y < 0:
            screen_center_y = 0
        player_centered = screen_center_x, screen_center_y

        self.camera.move_to(player_centered)

    def on_update(self, delta_time):
        """Movement and game logic"""

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

        # Position the camera
        self.center_camera_to_player()
```

(continues on next page)

(continued from previous page)

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


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

8.7 Step 7 - Add Coins And Sound



Next we will add some coins that the player can pickup. We'll also add a sound to be played when they pick it up, as well as a sound for when they jump.

8.7.1 Adding Coins to the Scene

First we need to add our coins to the scene. Let's start by adding a constant at the top of our application for the coin sprite scaling, similar to our `TILE_SCALING` one.

Listing 22: Add Coins and Sound

```
COIN_SCALING = 0.5
```

Next in our `setup` function we can create our coins using a for loop like we've done for the ground previously, and then add them to the scene.

Listing 23: Add Coins and Sound

```
        # Use a loop to place some coins for our character to pick up
        for x in range(128, 1250, 256):
            coin = arcade.Sprite(":resources:images/items/coinGold.png", COIN_SCALING)
            coin.center_x = x
            coin.center_y = 96
            self.scene.add_sprite("Coins", coin)
```

8.7.2 Loading Sounds

Now we can load in our sounds for collecting the coin and jumping. Later we will use these variables to play the sounds when the specific events happen. Add the following to the `__init__` function to load the sounds:

Listing 24: Add Coins and Sound

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

Then we can play our jump sound when the player jumps, by adding it to the `on_key_press` function:

Listing 25: Add Coins and Sound

```
    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
                arcade.play_sound(self.jump_sound)
        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
```

8.7.3 Collision Detection

Lastly, we need to find out if the player hit a coin. We can do this in our `on_update` function by using the `arcade.check_for_collision_with_list` function. We can pass the player sprite, along with a `SpriteList` that holds the coins. The function will return a list of the coins that the player is currently colliding with. If there are no coins in contact, the list will be empty.

Then we can use the `Sprite.remove_from_sprite_lists` function which will remove a given sprite from any SpriteLists it belongs to, effectively deleting it from the game.

Note: Notice that any transparent "white-space" around the image counts as the hitbox. You can trim the space in a graphics editor, or later on, we'll go over how to customize the hitbox of a Sprite.

Add the following to the `on_update` function to add collision detection and play a sound when the player picks up a coin.

Listing 26: Add Coins and Sound

```
        # 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()
            # Play a sound
            arcade.play_sound(self.collect_coin_sound)
```

Note: Spend time placing the coins where you would like them. If you have extra time, try adding more than just coins. Also add gems or keys from the graphics provided.

You could also subclass the coin sprite and add an attribute for a score value. Then you could have coins worth one point, and gems worth 5, 10, and 15 points.

8.7.4 Source Code

Listing 27: Add Coins and Sound

```
"""
Platformer Game
"""
import arcade

# Constants
SCREEN_WIDTH = 1000
SCREEN_HEIGHT = 650
SCREEN_TITLE = "Platformer"

# Constants used to scale our sprites from their original size
CHARACTER_SCALING = 1
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)

        # Our Scene Object
        self.scene = None

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

        # Our physics engine
        self.physics_engine = None

        # A Camera that can be used for scrolling the screen
        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")

        arcade.set_background_color(arcade.csscolor.CORNFLOWER_BLUE)

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

        # Set up the Camera
        self.camera = arcade.Camera(self.width, self.height)

        # Initialize Scene
        self.scene = arcade.Scene()

        # Set up the player, specifically placing it at these coordinates.
        image_source = ":resources:images/animated_characters/female_adventurer/
↪femaleAdventurer_idle.png"
        self.player_sprite = arcade.Sprite(image_source, CHARACTER_SCALING)
        self.player_sprite.center_x = 64
        self.player_sprite.center_y = 128
        self.scene.add_sprite("Player", self.player_sprite)

        # 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", 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]]

```

(continues on next page)

(continued from previous page)

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

        # Use a loop to place some coins for our character to pick up
        for x in range(128, 1250, 256):
            coin = arcade.Sprite(":resources:images/items/coinGold.png", COIN_SCALING)
            coin.center_x = x
            coin.center_y = 96
            self.scene.add_sprite("Coins", coin)

        # Create the 'physics engine'
        self.physics_engine = arcade.PhysicsEnginePlatformer(
            self.player_sprite, gravity_constant=GRAVITY, walls=self.scene["Walls"]
        )

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

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

        # Activate our Camera
        self.camera.use()

        # Draw our Scene
        self.scene.draw()

    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
                arcade.play_sound(self.jump_sound)
        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 when the user releases a key."""

        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

```

(continues on next page)

(continued from previous page)

```
    def center_camera_to_player(self):
        screen_center_x = self.player_sprite.center_x - (self.camera.viewport_width / 2)
        screen_center_y = self.player_sprite.center_y - (
            self.camera.viewport_height / 2
        )
        if screen_center_x < 0:
            screen_center_x = 0
        if screen_center_y < 0:
            screen_center_y = 0
        player_centered = screen_center_x, screen_center_y

        self.camera.move_to(player_centered)

    def on_update(self, delta_time):
        """Movement and game logic"""

        # Move the player with the 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()
            # Play a sound
            arcade.play_sound(self.collect_coin_sound)

        # Position the camera
        self.center_camera_to_player()


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


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

8.8 Step 8 - Display The Score

Now that we can collect coins and get points, we need a way to display the score on the screen.

This process is a little bit more complex than just drawing some text at an X and Y location. For properly drawing text, or any GUI elements, we need to use a separate camera than the one we use to draw the rest of our scene.

This is because we are scrolling around the main game camera, but we want our GUI elements to stay still. Using a second camera lets us do this.

As an example, if we were not to use a second camera, and instead draw on the same camera as our scene. We would need to offset the position that we draw our text at by position of the camera. This might be easier if you're only displaying one thing, but if you have a lot of GUI elements this could get out of hand.

First start by creating the new GUI camera and the score variables in the `__init__` function.

Listing 28: Display The Score - The init method

```
        # A Camera that can be used to draw GUI elements
        self.gui_camera = None

        # Keep track of the score
        self.score = 0
```

Then we can initialize them in the `setup` function. We reset the score to 0 here because this function is intended to fully reset the game back to it's starting state.

Listing 29: Display The Score - The setup method

```
        # Set up the GUI Camera
        self.gui_camera = arcade.Camera(self.width, self.height)

        # Keep track of the score
        self.score = 0
```

Then in our `on_draw` function we can first draw our scene like normal, and then switch to the GUI camera, and then finally draw our text.

Listing 30: Display The Score - The on_draw method

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

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

        # Activate the game camera
        self.camera.use()

        # Draw our Scene
        self.scene.draw()

        # Activate the GUI camera before drawing GUI elements
        self.gui_camera.use()

        # Draw our score on the screen, scrolling it with the viewport
```

(continues on next page)

(continued from previous page)

```
        score_text = f"Score: {self.score}"
        arcade.draw_text(
            score_text,
            10,
            10,
            arcade.csscolor.WHITE,
            18,
        )
```

Lastly in the `on_update` function we just need to update the score when a player collects a coin:

Listing 31: Display The Score - The on_update method

```
        # 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()
            # Play a sound
            arcade.play_sound(self.collect_coin_sound)
            # Add one to the score
            self.score += 1
```

Note: You might also want to add:

- A count of how many coins are left to be collected.
- Number of lives left.
- A timer: timer
- This example shows how to add an FPS timer: stress_test_draw_moving

8.8.1 Source Code

Listing 32: Display The Score

```
"""
Platformer Game
"""
import arcade

# Constants
SCREEN_WIDTH = 1000
SCREEN_HEIGHT = 650
SCREEN_TITLE = "Platformer"

# Constants used to scale our sprites from their original size
CHARACTER_SCALING = 1
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)

        # Our Scene Object
        self.scene = None

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

        # Our physics engine
        self.physics_engine = None

        # A Camera that can be used for scrolling the screen
        self.camera = None

        # A Camera that can be used to draw GUI elements
        self.gui_camera = None

        # Keep track of the score
        self.score = 0

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

        arcade.set_background_color(arcade.csscolor.CORNFLOWER_BLUE)

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

        # Set up the Game Camera
        self.camera = arcade.Camera(self.width, self.height)

        # Set up the GUI Camera
        self.gui_camera = arcade.Camera(self.width, self.height)

        # Keep track of the score
        self.score = 0

```

(continues on next page)

(continued from previous page)

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

        # Set up the player, specifically placing it at these coordinates.
        image_source = ":resources:images/animated_characters/female_adventurer/
↪femaleAdventurer_idle.png"
        self.player_sprite = arcade.Sprite(image_source, CHARACTER_SCALING)
        self.player_sprite.center_x = 64
        self.player_sprite.center_y = 96
        self.scene.add_sprite("Player", self.player_sprite)

        # 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", 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", TILE_SCALING
            )
            wall.position = coordinate
            self.scene.add_sprite("Walls", wall)

        # Use a loop to place some coins for our character to pick up
        for x in range(128, 1250, 256):
            coin = arcade.Sprite(":resources:images/items/coinGold.png", COIN_SCALING)
            coin.center_x = x
            coin.center_y = 96
            self.scene.add_sprite("Coins", coin)

        # Create the 'physics engine'
        self.physics_engine = arcade.PhysicsEnginePlatformer(
            self.player_sprite, gravity_constant=GRAVITY, walls=self.scene["Walls"]
        )

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

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

        # Activate the game camera
        self.camera.use()

```

(continues on next page)

(continued from previous page)

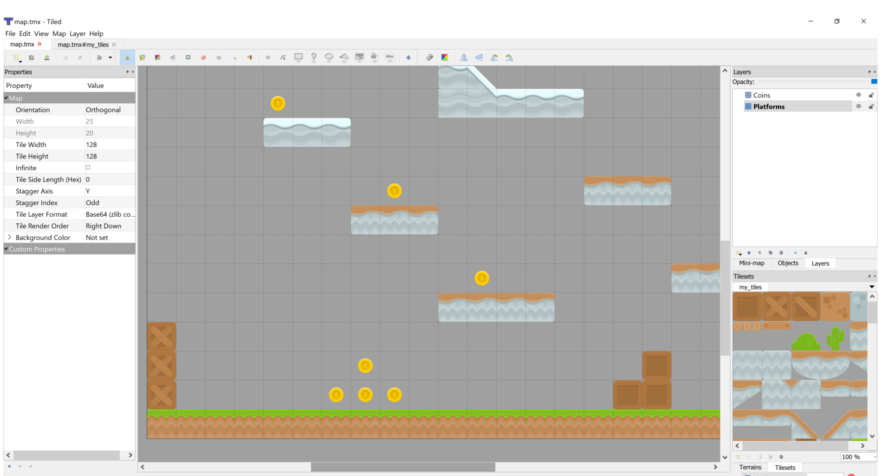
```python
        # Draw our Scene
        self.scene.draw()

        # Activate the GUI camera before drawing GUI elements
        self.gui_camera.use()

        # Draw our score on the screen, scrolling it with the viewport
        score_text = f"Score: {self.score}"
        arcade.draw_text(
            score_text,
            10,
            10,
            arcade.csscolor.WHITE,
            18,
        )

    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
                arcade.play_sound(self.jump_sound)
        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 when the user releases a key."""

        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 center_camera_to_player(self):
        screen_center_x = self.player_sprite.center_x - (self.camera.viewport_width / 2)
        screen_center_y = self.player_sprite.center_y - (
            self.camera.viewport_height / 2
        )
        if screen_center_x < 0:
            screen_center_x = 0
        if screen_center_y < 0:
            screen_center_y = 0
        player_centered = screen_center_x, screen_center_y

        self.camera.move_to(player_centered)

    def on_update(self, delta_time):
        """Movement and game logic"""

```

(continues on next page)

(continued from previous page)

```python
        # Move the player with the 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()
            # Play a sound
            arcade.play_sound(self.collect_coin_sound)
            # Add one to the score
            self.score += 1

        # Position the camera
        self.center_camera_to_player()


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


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

8.9 Step 9 - Use Tiled Map Editor



8.9.1 Create a Map File

For this part, instead of placing the tiles through code using specific points, we'll use a map editor that we can build maps with and then load in the map files.

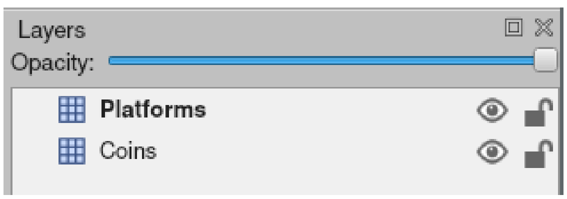
To start off with, download and install the Tiled Map Editor. (Think about donating, as it is a wonderful project provided for free.)

Tiled already has excellent documentation available at https://doc.mapeditor.org/, so for this tutorial we'll assume that you're already familiar with how to create maps using Tiled. If you're not, you can check out the Tiled documentation and come back to here.

From this point on in the tutorial, every chapter will be working with a Tiled map. If you don't want to create your own yet, Arcade ships a few examples in it's included `resources` folder, which is what these examples pull from, so you don't have to create your own maps yet if you don't want to.

We'll start with a basic `map.json` file provided by Arcade. You can open this file in Tiled and look at how it's setup, but we'll go over some of the basics now. You can save files in either the "JSON" or "TMX" format.

In this map we have two layers named "Platforms" and "Coins". On the platforms layer are all of the blocks which a player will collide with using the physics engine, and on the coins layer are all the coins the player can pickup to increase their score. That's pretty much it for this map.



These layers will be automatically loaded by Arcade as SpriteLists that we can access and draw with our scene. Let's look at how we load in the map, first we'll create a `tile_map` object in our init function:

Listing 33: Load a map - Create the object

```
        # Our TileMap Object
        self.tile_map = None
```

Then we will do the actual loading in the setup function Our new setup function will look like this:

Listing 34: Load a map - Setup the map

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

        # Set up the Cameras
        self.camera = arcade.Camera(self.width, self.height)
        self.gui_camera = arcade.Camera(self.width, self.height)

        # Name of map file to load
        map_name = ":resources:tiled_maps/map.json"

        # Layer specific options are defined based on Layer names in a dictionary
        # Doing this will make the SpriteList for the platforms layer
        # use spatial hashing for detection.
        layer_options = {
            "Platforms": {
```

(continues on next page)

(continued from previous page)

```
                "use_spatial_hash": True,
            },
        }

        # Read in the tiled map
        self.tile_map = arcade.load_tilemap(map_name, TILE_SCALING, layer_options)

        # Initialize Scene with our TileMap, this will automatically add all layers
        # from the map as SpriteLists in the scene in the proper order.
        self.scene = arcade.Scene.from_tilemap(self.tile_map)

        # Keep track of the score
        self.score = 0

        # Set up the player, specifically placing it at these coordinates.
        image_source = ":resources:images/animated_characters/female_adventurer/
↪femaleAdventurer_idle.png"
        self.player_sprite = arcade.Sprite(image_source, CHARACTER_SCALING)
        self.player_sprite.center_x = 128
        self.player_sprite.center_y = 128
        self.scene.add_sprite("Player", self.player_sprite)

        # --- Other stuff
        # Set the background color
        if self.tile_map.background_color:
            arcade.set_background_color(self.tile_map.background_color)

        # Create the 'physics engine'
        self.physics_engine = arcade.PhysicsEnginePlatformer(
            self.player_sprite, gravity_constant=GRAVITY, walls=self.scene["Platforms"]
        )
```

This is pretty much all that needs done to load in the Tilemap, we get a Scene created from it and can use it just like we have been up until now. But let's go through this setup function and look at all the updates.

In the first piece we define the name of map file we want to load, that one is pretty simple.

Next we have a `layer_options` variable. This is a dictionary which let's you assign special options to specific layers in the map. In this example, we're just adding spatial hashing to the "Platforms" layer, but we can do a few other things here.

The available options you can set for a layer are:

- `use_spatial_hash` - Make a Layer's SpriteList use spatial hashing
- `scaling` - Set per layer scaling of Sprites
- `hit_box_algorithm` - Change the hit box algorithm used when doing collision detection with this SpriteList
- `hit_box_detail` - Change the hit box detail used when doing collision detection with this SpriteList

Then we actually load in the Tilemap using the `arcade.load_tilemap` function. This will return us back an instance of the `arcade.TileMap` class. For now, we don't actually need to interact with this object much, but later we will do some more advanced things like setting enemy spawn points and movement paths from within the map editor.

Finally we use a new way to create our Scene, with the `arcade.Scene.from_tilemap` function. This let's you specify a TileMap object, and will automatically construct a scene with all of the layers in your map, arranged in the proper render order. Then you can work with the scene exactly like we have up until this point.

The last small piece we changed is when we create the physics engine, we've now have to use "Platforms" as the sprite list name since that is the name of our Layer in the map file.

And that's all! You should now have a full game loading from a map file created with Tiled.

Some things we will use Tiled for in upcoming chapters are:

- Platforms that you run into (or you can think of them as walls)
- Moving platforms
- Coins or objects to pick up
- Background objects that you don't interact with, but appear behind the player
- Foreground objects that you don't interact with, but appear in front of the player
- Insta-death blocks and zones (like lava)
- Ladders
- Enemy spawn positions
- Enemy movement paths

8.9.2 Source Code

Listing 35: Load the Map

```python
"""
Platformer Game
"""
import arcade

# Constants
SCREEN_WIDTH = 1000
SCREEN_HEIGHT = 650
SCREEN_TITLE = "Platformer"

# Constants used to scale our sprites from their original size
CHARACTER_SCALING = 1
TILE_SCALING = 0.5
COIN_SCALING = 0.5

# Movement speed of player, in pixels per frame
PLAYER_MOVEMENT_SPEED = 10
GRAVITY = 1
PLAYER_JUMP_SPEED = 20


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

    def __init__(self):
```

(continues on next page)

(continued from previous page)

```python

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

        # Our TileMap Object
        self.tile_map = None

        # Our Scene Object
        self.scene = None

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

        # Our physics engine
        self.physics_engine = None

        # A Camera that can be used for scrolling the screen
        self.camera = None

        # A Camera that can be used to draw GUI elements
        self.gui_camera = None

        # Keep track of the score
        self.score = 0

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

        arcade.set_background_color(arcade.csscolor.CORNFLOWER_BLUE)

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

        # Set up the Cameras
        self.camera = arcade.Camera(self.width, self.height)
        self.gui_camera = arcade.Camera(self.width, self.height)

        # Name of map file to load
        map_name = ":resources:tiled_maps/map.json"

        # Layer specific options are defined based on Layer names in a dictionary
        # Doing this will make the SpriteList for the platforms layer
        # use spatial hashing for detection.
        layer_options = {
            "Platforms": {
                "use_spatial_hash": True,
            },
        }

        # Read in the tiled map
        self.tile_map = arcade.load_tilemap(map_name, TILE_SCALING, layer_options)
```

(continues on next page)

(continued from previous page)

```python

        # Initialize Scene with our TileMap, this will automatically add all layers
        # from the map as SpriteLists in the scene in the proper order.
        self.scene = arcade.Scene.from_tilemap(self.tile_map)

        # Keep track of the score
        self.score = 0

        # Set up the player, specifically placing it at these coordinates.
        image_source = ":resources:images/animated_characters/female_adventurer/
↪femaleAdventurer_idle.png"
        self.player_sprite = arcade.Sprite(image_source, CHARACTER_SCALING)
        self.player_sprite.center_x = 128
        self.player_sprite.center_y = 128
        self.scene.add_sprite("Player", self.player_sprite)

        # --- Other stuff
        # Set the background color
        if self.tile_map.background_color:
            arcade.set_background_color(self.tile_map.background_color)

        # Create the 'physics engine'
        self.physics_engine = arcade.PhysicsEnginePlatformer(
            self.player_sprite, gravity_constant=GRAVITY, walls=self.scene["Platforms"]
        )

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

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

        # Activate the game camera
        self.camera.use()

        # Draw our Scene
        self.scene.draw()

        # Activate the GUI camera before drawing GUI elements
        self.gui_camera.use()

        # Draw our score on the screen, scrolling it with the viewport
        score_text = f"Score: {self.score}"
        arcade.draw_text(
            score_text,
            10,
            10,
            arcade.csscolor.WHITE,
            18,
        )

    def on_key_press(self, key, modifiers):
```

(continues on next page)

(continued from previous page)

```python
        """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
                arcade.play_sound(self.jump_sound)
        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 when the user releases a key."""

        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 center_camera_to_player(self):
        screen_center_x = self.player_sprite.center_x - (self.camera.viewport_width / 2)
        screen_center_y = self.player_sprite.center_y - (
            self.camera.viewport_height / 2
        )
        if screen_center_x < 0:
            screen_center_x = 0
        if screen_center_y < 0:
            screen_center_y = 0
        player_centered = screen_center_x, screen_center_y

        self.camera.move_to(player_centered)

    def on_update(self, delta_time):
        """Movement and game logic"""

        # Move the player with the 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()
            # Play a sound
            arcade.play_sound(self.collect_coin_sound)
            # Add one to the score
            self.score += 1

```

(continues on next page)

(continued from previous page)

```python
        # Position the camera
        self.center_camera_to_player()


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


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

8.10 Step 10 - Multiple Levels and Other Layers

Now that we've seen the basics of loading a Tiled map, we'll give another example with some more features. In this example we'll add the following things:

- New layers including foreground, background, and "Don't Touch"
  - The background layer will appear behind the player
  - The foreground layer will appear in front of the player
  - The Don't Touch layer will cause the player to be reset to the start
- The player resets to the start if they fall off the map
- If the player gets to the right side of the map, the program attempts to load the next map
  - This is achieved by naming the maps with incrementing numbers, something like "map_01.json", "map_02.json", etc. Then having a level attribute to track which number we're on and increasing it and re-running the setup function.

To start things off, let's add a few constants at the top of our game. The first one we need to define is the size of a sprite in pixels. Along with that we need to know the grid size in pixels. These are used to calculate the end of the level.

Listing 36: Multiple Levels - Constants

```python
SPRITE_PIXEL_SIZE = 128
GRID_PIXEL_SIZE = SPRITE_PIXEL_SIZE * TILE_SCALING
```

Next we need to define a starting position for the player, and then since we're starting to have a larger number of layers in our game, it will be best to store their names in variables in case we need to change them later.

Listing 37: Multiple Levels - Constants

```python
# Player starting position
PLAYER_START_X = 64
PLAYER_START_Y = 225

# Layer Names from our TileMap
LAYER_NAME_PLATFORMS = "Platforms"
```

(continues on next page)

(continued from previous page)

```
LAYER_NAME_COINS = "Coins"
LAYER_NAME_FOREGROUND = "Foreground"
LAYER_NAME_BACKGROUND = "Background"
LAYER_NAME_DONT_TOUCH = "Don't Touch"
```

Then in the `__init__` function we'll add two new values. One to know where the right edge of the map is, and one to keep track of what level we're on, and add a new game over sound.

Listing 38: Multiple Levels - Init Function

```
        # Where is the right edge of the map?
        self.end_of_map = 0

        # Level
        self.level = 1

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

Also in our `__init__` function we'll need a variable to tell us if we need to reset the score. This will be the case if the player fails the level. However, now that the player can pass a level, we need to keep the score when calling our `setup` function for the new level. Otherwise it will reset the score back to 0

Listing 39: Multiple Levels - Init Function

```
        # Do we need to reset the score?
        self.reset_score = True
```

Then in our `setup` function we'll change up our map name variable to use that new level attribute, and add some extra layer specific options for the new layers we've added to our map.

Listing 40: Multiple Levels - Setup Function

```
        # Map name
        map_name = f":resources:tiled_maps/map2_level_{self.level}.json"

        # Layer Specific Options for the Tilemap
        layer_options = {
            LAYER_NAME_PLATFORMS: {
                "use_spatial_hash": True,
            },
            LAYER_NAME_COINS: {
                "use_spatial_hash": True,
            },
            LAYER_NAME_DONT_TOUCH: {
                "use_spatial_hash": True,
            },
        }
```

Now in order to make our player appear behind the "Foreground" layer, we need to add a line in our `setup` function before we create the player Sprite. This will basically be telling our Scene where in the render order we want to place the player. Previously we haven't defined this, and so it's always just been added to the end of the render order.

Listing 41: Multiple Levels - Setup Function

```
        # 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", LAYER_NAME_FOREGROUND)

        # Set up the player, specifically placing it at these coordinates.
        image_source = ":resources:images/animated_characters/female_adventurer/
↪femaleAdventurer_idle.png"
        self.player_sprite = arcade.Sprite(image_source, CHARACTER_SCALING)
        self.player_sprite.center_x = PLAYER_START_X
        self.player_sprite.center_y = PLAYER_START_Y
        self.scene.add_sprite("Player", self.player_sprite)
```

Next in our `setup` function we need to check to see if we need to reset the score or keep it.

Listing 42: Multiple Levels - Setup Function

```
        # Initiate New Scene with our TileMap, this will automatically add all layers
        # from the map as SpriteLists in the scene in the proper order.
        self.scene = arcade.Scene.from_tilemap(self.tile_map)

        # Keep track of the score, make sure we keep the score if the player finishes a␣
↪level
        if self.reset_score:
            self.score = 0
        self.reset_score = True
```

Lastly in our `setup` function we need to calculate the `end_of_map` value we added earlier in `init`.

Listing 43: Multiple Levels - Setup Function

```
        # Calculate the right edge of the my_map in pixels
        self.end_of_map = self.tile_map.width * GRID_PIXEL_SIZE
```

The `on_draw`, `on_key_press`, and `on_key_release` functions will be unchanged for this section, so the last thing to do is add a few things to the `on_update` function. First we check if the player has fallen off of the map, and if so, we move them back to the starting position. Then we check if they collided with something from the "Don't Touch" layer, and if so reset them to the start. Lastly we check if they've reached the end of the map, and if they have we increment the level value, tell our `setup` function not to reset the score, and then re-run the `setup` function.

Listing 44: Multiple Levels - Update Function

```
        # Did the player fall off the map?
        if self.player_sprite.center_y < -100:
            self.player_sprite.center_x = PLAYER_START_X
            self.player_sprite.center_y = PLAYER_START_Y

            arcade.play_sound(self.game_over)
```

(continues on next page)

(continued from previous page)

```
        # Did the player touch something they should not?
        if arcade.check_for_collision_with_list(
            self.player_sprite, self.scene[LAYER_NAME_DONT_TOUCH]
        ):
            self.player_sprite.change_x = 0
            self.player_sprite.change_y = 0
            self.player_sprite.center_x = PLAYER_START_X
            self.player_sprite.center_y = PLAYER_START_Y

            arcade.play_sound(self.game_over)

        # See if the user 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

            # Make sure to keep the score from this level when setting up the next level
            self.reset_score = False

            # Load the next level
            self.setup()
```

Note: What else might you want to do?

- sprite_enemies_in_platformer
- sprite_face_left_or_right
- Bullets (or something you can shoot)
  - sprite_bullets
  - sprite_bullets_aimed
  - sprite_bullets_enemy_aims
- Add sprite_explosion_bitmapped
- Add sprite_move_animation

8.10.1 Source Code

Listing 45: Multiple Levels

```
"""
Platformer Game
"""
import arcade

# Constants
SCREEN_WIDTH = 1000
SCREEN_HEIGHT = 650
```

(continues on next page)

(continued from previous page)

```
SCREEN_TITLE = "Platformer"

# Constants used to scale our sprites from their original size
CHARACTER_SCALING = 1
TILE_SCALING = 0.5
COIN_SCALING = 0.5
SPRITE_PIXEL_SIZE = 128
GRID_PIXEL_SIZE = SPRITE_PIXEL_SIZE * TILE_SCALING

# Movement speed of player, in pixels per frame
PLAYER_MOVEMENT_SPEED = 10
GRAVITY = 1
PLAYER_JUMP_SPEED = 20

# Player starting position
PLAYER_START_X = 64
PLAYER_START_Y = 225

# Layer Names from our TileMap
LAYER_NAME_PLATFORMS = "Platforms"
LAYER_NAME_COINS = "Coins"
LAYER_NAME_FOREGROUND = "Foreground"
LAYER_NAME_BACKGROUND = "Background"
LAYER_NAME_DONT_TOUCH = "Don't Touch"


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)

        # Our TileMap Object
        self.tile_map = None

        # Our Scene Object
        self.scene = None

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

        # Our physics engine
        self.physics_engine = None

        # A Camera that can be used for scrolling the screen
        self.camera = None

        # A Camera that can be used to draw GUI elements
```

(continues on next page)

(continued from previous page)

```
        self.gui_camera = None

        # Keep track of the score
        self.score = 0

        # Do we need to reset the score?
        self.reset_score = True

        # Where is the right edge of the map?
        self.end_of_map = 0

        # Level
        self.level = 1

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

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

        # Set up the Cameras
        self.camera = arcade.Camera(self.width, self.height)
        self.gui_camera = arcade.Camera(self.width, self.height)

        # Map name
        map_name = f":resources:tiled_maps/map2_level_{self.level}.json"

        # Layer Specific Options for the Tilemap
        layer_options = {
            LAYER_NAME_PLATFORMS: {
                "use_spatial_hash": True,
            },
            LAYER_NAME_COINS: {
                "use_spatial_hash": True,
            },
            LAYER_NAME_DONT_TOUCH: {
                "use_spatial_hash": True,
            },
        }

        # Load in TileMap
        self.tile_map = arcade.load_tilemap(map_name, TILE_SCALING, layer_options)

        # Initiate New Scene with our TileMap, this will automatically add all layers
        # from the map as SpriteLists in the scene in the proper order.
        self.scene = arcade.Scene.from_tilemap(self.tile_map)

        # Keep track of the score, make sure we keep the score if the player finishes a␣
↪level
        if self.reset_score:
```

(continues on next page)

(continued from previous page)

```
            self.score = 0
        self.reset_score = True

        # 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", LAYER_NAME_FOREGROUND)

        # Set up the player, specifically placing it at these coordinates.
        image_source = ":resources:images/animated_characters/female_adventurer/femaleAdventurer_idle.png"
        self.player_sprite = arcade.Sprite(image_source, CHARACTER_SCALING)
        self.player_sprite.center_x = PLAYER_START_X
        self.player_sprite.center_y = PLAYER_START_Y
        self.scene.add_sprite("Player", self.player_sprite)

        # --- Load in a map from the tiled editor ---

        # Calculate the right edge of the my_map in pixels
        self.end_of_map = self.tile_map.width * GRID_PIXEL_SIZE

        # --- Other stuff
        # Set the background color
        if self.tile_map.background_color:
            arcade.set_background_color(self.tile_map.background_color)

        # Create the 'physics engine'
        self.physics_engine = arcade.PhysicsEnginePlatformer(
            self.player_sprite,
            gravity_constant=GRAVITY,
            walls=self.scene[LAYER_NAME_PLATFORMS],
        )

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

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

        # Activate the game camera
        self.camera.use()

        # Draw our Scene
        self.scene.draw()

        # Activate the GUI camera before drawing GUI elements
        self.gui_camera.use()

        # Draw our score on the screen, scrolling it with the viewport
```

(continues on next page)

(continued from previous page)

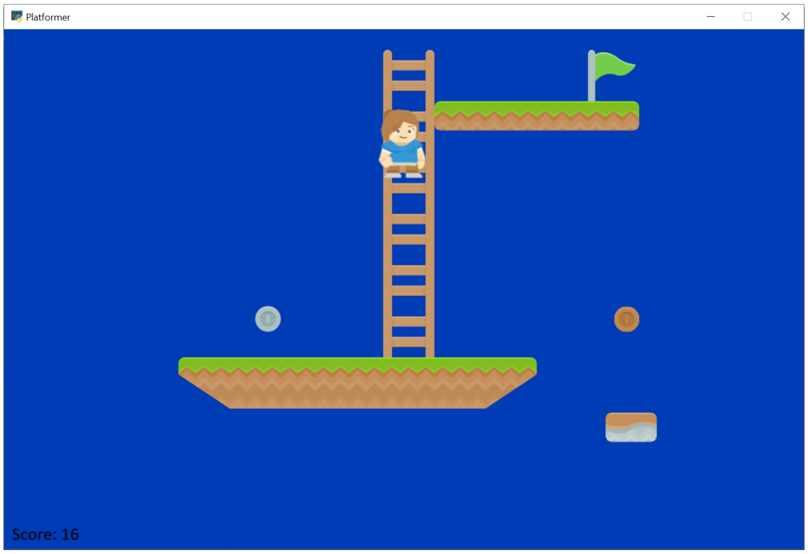
```python
        score_text = f"Score: {self.score}"
        arcade.draw_text(
            score_text,
            10,
            10,
            arcade.csscolor.BLACK,
            18,
        )

    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
                arcade.play_sound(self.jump_sound)
        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 when the user releases a key."""

        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 center_camera_to_player(self):
        screen_center_x = self.player_sprite.center_x - (self.camera.viewport_width / 2)
        screen_center_y = self.player_sprite.center_y - (
            self.camera.viewport_height / 2
        )
        if screen_center_x < 0:
            screen_center_x = 0
        if screen_center_y < 0:
            screen_center_y = 0
        player_centered = screen_center_x, screen_center_y

        self.camera.move_to(player_centered)

    def update(self, delta_time):
        """Movement and game logic"""

        # Move the player with the 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[LAYER_NAME_COINS]
        )
```

(continues on next page)

(continued from previous page)

```python

        # 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()
            # Play a sound
            arcade.play_sound(self.collect_coin_sound)
            # Add one to the score
            self.score += 1

        # Did the player fall off the map?
        if self.player_sprite.center_y < -100:
            self.player_sprite.center_x = PLAYER_START_X
            self.player_sprite.center_y = PLAYER_START_Y

            arcade.play_sound(self.game_over)

        # Did the player touch something they should not?
        if arcade.check_for_collision_with_list(
            self.player_sprite, self.scene[LAYER_NAME_DONT_TOUCH]
        ):
            self.player_sprite.change_x = 0
            self.player_sprite.change_y = 0
            self.player_sprite.center_x = PLAYER_START_X
            self.player_sprite.center_y = PLAYER_START_Y

            arcade.play_sound(self.game_over)

        # See if the user 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

            # Make sure to keep the score from this level when setting up the next level
            self.reset_score = False

            # Load the next level
            self.setup()

        # Position the camera
        self.center_camera_to_player()


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


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

8.11 Step 11 - Add Ladders, Properties, and a Moving Platform



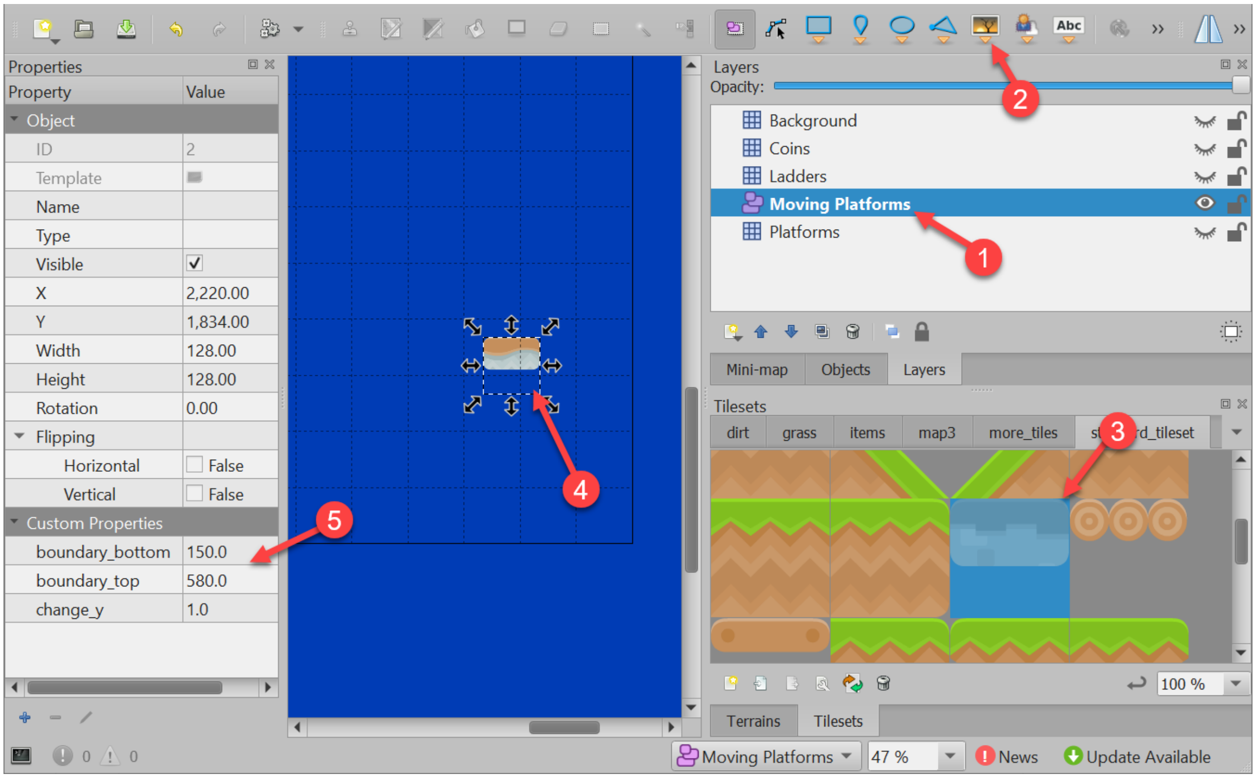
This example shows using:

- Ladders
- Properties to define point value of coins and flags
- Properties and an object layer to define a moving platform.

To create a moving platform using TMX editor, there are a few steps:

1. Define an **object layer** instead of a tile layer.
2. Select **Insert Tile**
3. Select the tile you wish to insert.
4. Place the tile.
5. Add custom properties. You can add:

- `change_x`
- `change_y`
- `boundary_bottom`
- `boundary_top`
- `boundary_left`
- `boundary_right`



Listing 46: Ladders, Animated Tiles, and Moving Platforms

```python
"""
Platformer Game
"""
import os

import arcade

# Constants
SCREEN_WIDTH = 1000
SCREEN_HEIGHT = 650
SCREEN_TITLE = "Platformer"

# Constants used to scale our sprites from their original size
CHARACTER_SCALING = 1
TILE_SCALING = 0.5
COIN_SCALING = 0.5
SPRITE_PIXEL_SIZE = 128
GRID_PIXEL_SIZE = SPRITE_PIXEL_SIZE * TILE_SCALING

# Movement speed of player, in pixels per frame
PLAYER_MOVEMENT_SPEED = 7
GRAVITY = 1.5
PLAYER_JUMP_SPEED = 30

PLAYER_START_X = 64
PLAYER_START_Y = 256
```

(continues on next page)

(continued from previous page)

```python

# Layer Names from our TileMap
LAYER_NAME_MOVING_PLATFORMS = "Moving Platforms"
LAYER_NAME_PLATFORMS = "Platforms"
LAYER_NAME_COINS = "Coins"
LAYER_NAME_BACKGROUND = "Background"
LAYER_NAME_LADDERS = "Ladders"


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

    def __init__(self):
        """
        Initializer for the game
        """

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

        # Set the path to start with this program
        file_path = os.path.dirname(os.path.abspath(__file__))
        os.chdir(file_path)

        # Our TileMap Object
        self.tile_map = None

        # Our Scene Object
        self.scene = None

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

        # Our 'physics' engine
        self.physics_engine = None

        # A Camera that can be used for scrolling the screen
        self.camera = None

        # A Camera that can be used to draw GUI elements
        self.gui_camera = None

        self.end_of_map = 0

        # Keep track of the score
        self.score = 0

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

(continues on next page)

(continued from previous page)

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

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

        # Set up the Cameras
        self.camera = arcade.Camera(self.width, self.height)
        self.gui_camera = arcade.Camera(self.width, self.height)

        # Map name
        map_name = ":resources:tiled_maps/map_with_ladders.json"

        # Layer Specific Options for the Tilemap
        layer_options = {
            LAYER_NAME_PLATFORMS: {
                "use_spatial_hash": True,
            },
            LAYER_NAME_MOVING_PLATFORMS: {
                "use_spatial_hash": False,
            },
            LAYER_NAME_LADDERS: {
                "use_spatial_hash": True,
            },
            LAYER_NAME_COINS: {
                "use_spatial_hash": True,
            },
        }

        # Load in TileMap
        self.tile_map = arcade.load_tilemap(map_name, TILE_SCALING, layer_options)

        # Initiate New Scene with our TileMap, this will automatically add all layers
        # from the map as SpriteLists in the scene in the proper order.
        self.scene = arcade.Scene.from_tilemap(self.tile_map)

        # Keep track of the score
        self.score = 0

        # Set up the player, specifically placing it at these coordinates.
        image_source = ":resources:images/animated_characters/female_adventurer/
↪femaleAdventurer_idle.png"
        self.player_sprite = arcade.Sprite(image_source, CHARACTER_SCALING)
        self.player_sprite.center_x = PLAYER_START_X
        self.player_sprite.center_y = PLAYER_START_Y
        self.scene.add_sprite("Player", self.player_sprite)

        # Calculate the right edge of the my_map in pixels
        self.end_of_map = self.tile_map.width * GRID_PIXEL_SIZE

        # --- Other stuff
        # Set the background color
        if self.tile_map.background_color:
```

(continues on next page)

(continued from previous page)

```
            arcade.set_background_color(self.tile_map.background_color)

        # Create the 'physics engine'
        self.physics_engine = arcade.PhysicsEnginePlatformer(
            self.player_sprite,
            platforms=self.scene[LAYER_NAME_MOVING_PLATFORMS],
            gravity_constant=GRAVITY,
            ladders=self.scene[LAYER_NAME_LADDERS],
            walls=self.scene[LAYER_NAME_PLATFORMS]
        )

    def on_draw(self):
        """Render the screen."""
        # Clear the screen to the background color
        self.clear()

        # Activate the game camera
        self.camera.use()

        # Draw our Scene
        self.scene.draw()

        # Activate the GUI camera before drawing GUI elements
        self.gui_camera.use()

        # Draw our score on the screen, scrolling it with the viewport
        score_text = f"Score: {self.score}"
        arcade.draw_text(
            score_text,
            10,
            10,
            arcade.csscolor.BLACK,
            18,
        )

    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.is_on_ladder():
                self.player_sprite.change_y = PLAYER_MOVEMENT_SPEED
            elif self.physics_engine.can_jump():
                self.player_sprite.change_y = PLAYER_JUMP_SPEED
                arcade.play_sound(self.jump_sound)
        elif key == arcade.key.DOWN or key == arcade.key.S:
            if self.physics_engine.is_on_ladder():
                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

```

(continues on next page)

(continued from previous page)

```
    def on_key_release(self, key, modifiers):
        """Called when the user releases a key."""

        if key == arcade.key.UP or key == arcade.key.W:
            if self.physics_engine.is_on_ladder():
                self.player_sprite.change_y = 0
        elif key == arcade.key.DOWN or key == arcade.key.S:
            if self.physics_engine.is_on_ladder():
                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 center_camera_to_player(self):
        screen_center_x = self.player_sprite.center_x - (self.camera.viewport_width / 2)
        screen_center_y = self.player_sprite.center_y - (
            self.camera.viewport_height / 2
        )
        if screen_center_x < 0:
            screen_center_x = 0
        if screen_center_y < 0:
            screen_center_y = 0
        player_centered = screen_center_x, screen_center_y

        self.camera.move_to(player_centered, 0.2)

    def update(self, delta_time):
        """Movement and game logic"""
        # Move the player with the physics engine
        self.physics_engine.update()

        # Update animations
        self.scene.update_animation(
            delta_time, [LAYER_NAME_COINS, LAYER_NAME_BACKGROUND]
        )

        # Update walls, used with moving platforms
        self.scene.update([LAYER_NAME_MOVING_PLATFORMS])

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

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

            # Figure out how many points this coin is worth
            if "Points" not in coin.properties:
                print("Warning, collected a coin without a Points property.")
            else:
```

(continues on next page)

(continued from previous page)

```
                points = int(coin.properties["Points"])
                self.score += points

            # Remove the coin
            coin.remove_from_sprite_lists()
            arcade.play_sound(self.collect_coin_sound)

        # Position the camera
        self.center_camera_to_player()


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


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

8.11.1 Source Code

Listing 47: Ladders and More

```
"""
Platformer Game
"""
import os

import arcade

# Constants
SCREEN_WIDTH = 1000
SCREEN_HEIGHT = 650
SCREEN_TITLE = "Platformer"

# Constants used to scale our sprites from their original size
CHARACTER_SCALING = 1
TILE_SCALING = 0.5
COIN_SCALING = 0.5
SPRITE_PIXEL_SIZE = 128
GRID_PIXEL_SIZE = SPRITE_PIXEL_SIZE * TILE_SCALING

# Movement speed of player, in pixels per frame
PLAYER_MOVEMENT_SPEED = 7
GRAVITY = 1.5
PLAYER_JUMP_SPEED = 30

PLAYER_START_X = 64
```

(continues on next page)

(continued from previous page)

```
PLAYER_START_Y = 256

# Layer Names from our TileMap
LAYER_NAME_MOVING_PLATFORMS = "Moving Platforms"
LAYER_NAME_PLATFORMS = "Platforms"
LAYER_NAME_COINS = "Coins"
LAYER_NAME_BACKGROUND = "Background"
LAYER_NAME_LADDERS = "Ladders"


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

    def __init__(self):
        """
        Initializer for the game
        """

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

        # Set the path to start with this program
        file_path = os.path.dirname(os.path.abspath(__file__))
        os.chdir(file_path)

        # Our TileMap Object
        self.tile_map = None

        # Our Scene Object
        self.scene = None

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

        # Our 'physics' engine
        self.physics_engine = None

        # A Camera that can be used for scrolling the screen
        self.camera = None

        # A Camera that can be used to draw GUI elements
        self.gui_camera = None

        self.end_of_map = 0

        # Keep track of the score
        self.score = 0

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

(continues on next page)

(continued from previous page)

```python
        self.jump_sound = arcade.load_sound(":resources:sounds/jump1.wav")
        self.game_over = arcade.load_sound(":resources:sounds/gameover1.wav")

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

        # Set up the Cameras
        self.camera = arcade.Camera(self.width, self.height)
        self.gui_camera = arcade.Camera(self.width, self.height)

        # Map name
        map_name = ":resources:tiled_maps/map_with_ladders.json"

        # Layer Specific Options for the Tilemap
        layer_options = {
            LAYER_NAME_PLATFORMS: {
                "use_spatial_hash": True,
            },
            LAYER_NAME_MOVING_PLATFORMS: {
                "use_spatial_hash": False,
            },
            LAYER_NAME_LADDERS: {
                "use_spatial_hash": True,
            },
            LAYER_NAME_COINS: {
                "use_spatial_hash": True,
            },
        }

        # Load in TileMap
        self.tile_map = arcade.load_tilemap(map_name, TILE_SCALING, layer_options)

        # Initiate New Scene with our TileMap, this will automatically add all layers
        # from the map as SpriteLists in the scene in the proper order.
        self.scene = arcade.Scene.from_tilemap(self.tile_map)

        # Keep track of the score
        self.score = 0

        # Set up the player, specifically placing it at these coordinates.
        image_source = ":resources:images/animated_characters/female_adventurer/femaleAdventurer_idle.png"
        self.player_sprite = arcade.Sprite(image_source, CHARACTER_SCALING)
        self.player_sprite.center_x = PLAYER_START_X
        self.player_sprite.center_y = PLAYER_START_Y
        self.scene.add_sprite("Player", self.player_sprite)

        # Calculate the right edge of the my_map in pixels
        self.end_of_map = self.tile_map.width * GRID_PIXEL_SIZE

        # --- Other stuff
        # Set the background color
```

(continues on next page)

(continued from previous page)

```
        if self.tile_map.background_color:
            arcade.set_background_color(self.tile_map.background_color)

        # Create the 'physics engine'
        self.physics_engine = arcade.PhysicsEnginePlatformer(
            self.player_sprite,
            platforms=self.scene[LAYER_NAME_MOVING_PLATFORMS],
            gravity_constant=GRAVITY,
            ladders=self.scene[LAYER_NAME_LADDERS],
            walls=self.scene[LAYER_NAME_PLATFORMS]
        )

    def on_draw(self):
        """Render the screen."""
        # Clear the screen to the background color
        self.clear()

        # Activate the game camera
        self.camera.use()

        # Draw our Scene
        self.scene.draw()

        # Activate the GUI camera before drawing GUI elements
        self.gui_camera.use()

        # Draw our score on the screen, scrolling it with the viewport
        score_text = f"Score: {self.score}"
        arcade.draw_text(
            score_text,
            10,
            10,
            arcade.csscolor.BLACK,
            18,
        )

    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.is_on_ladder():
                self.player_sprite.change_y = PLAYER_MOVEMENT_SPEED
            elif self.physics_engine.can_jump():
                self.player_sprite.change_y = PLAYER_JUMP_SPEED
                arcade.play_sound(self.jump_sound)
        elif key == arcade.key.DOWN or key == arcade.key.S:
            if self.physics_engine.is_on_ladder():
                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
```

(continues on next page)

(continued from previous page)

```python

    def on_key_release(self, key, modifiers):
        """Called when the user releases a key."""

        if key == arcade.key.UP or key == arcade.key.W:
            if self.physics_engine.is_on_ladder():
                self.player_sprite.change_y = 0
        elif key == arcade.key.DOWN or key == arcade.key.S:
            if self.physics_engine.is_on_ladder():
                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 center_camera_to_player(self):
        screen_center_x = self.player_sprite.center_x - (self.camera.viewport_width / 2)
        screen_center_y = self.player_sprite.center_y - (
            self.camera.viewport_height / 2
        )
        if screen_center_x < 0:
            screen_center_x = 0
        if screen_center_y < 0:
            screen_center_y = 0
        player_centered = screen_center_x, screen_center_y

        self.camera.move_to(player_centered, 0.2)

    def update(self, delta_time):
        """Movement and game logic"""
        # Move the player with the physics engine
        self.physics_engine.update()

        # Update animations
        self.scene.update_animation(
            delta_time, [LAYER_NAME_COINS, LAYER_NAME_BACKGROUND]
        )

        # Update walls, used with moving platforms
        self.scene.update([LAYER_NAME_MOVING_PLATFORMS])

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

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

            # Figure out how many points this coin is worth
            if "Points" not in coin.properties:
                print("Warning, collected a coin without a Points property.")
```

(continues on next page)

(continued from previous page)

```
            else:
                points = int(coin.properties["Points"])
                self.score += points

            # Remove the coin
            coin.remove_from_sprite_lists()
            arcade.play_sound(self.collect_coin_sound)

        # Position the camera
        self.center_camera_to_player()


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


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

8.12 Step 12 - Add Character Animations, and Better Keyboard Control

Add character animations!

Listing 48: Animate Characters

```
"""
Platformer Game

python -m arcade.examples.platform_tutorial.11_animate_character
"""
import os

import arcade

# Constants
SCREEN_WIDTH = 1000
SCREEN_HEIGHT = 650
SCREEN_TITLE = "Platformer"

# Constants used to scale our sprites from their original size
TILE_SCALING = 0.5
CHARACTER_SCALING = TILE_SCALING * 2
COIN_SCALING = TILE_SCALING
SPRITE_PIXEL_SIZE = 128
GRID_PIXEL_SIZE = SPRITE_PIXEL_SIZE * TILE_SCALING

```

(continues on next page)

(continued from previous page)

```
# Movement speed of player, in pixels per frame
PLAYER_MOVEMENT_SPEED = 7
GRAVITY = 1.5
PLAYER_JUMP_SPEED = 30

PLAYER_START_X = SPRITE_PIXEL_SIZE * TILE_SCALING * 2
PLAYER_START_Y = SPRITE_PIXEL_SIZE * TILE_SCALING * 1

# Constants used to track if the player is facing left or right
RIGHT_FACING = 0
LEFT_FACING = 1

LAYER_NAME_MOVING_PLATFORMS = "Moving Platforms"
LAYER_NAME_PLATFORMS = "Platforms"
LAYER_NAME_COINS = "Coins"
LAYER_NAME_BACKGROUND = "Background"
LAYER_NAME_LADDERS = "Ladders"
LAYER_NAME_PLAYER = "Player"


def load_texture_pair(filename):
    """
    Load a texture pair, with the second being a mirror image.
    """
    return [
        arcade.load_texture(filename),
        arcade.load_texture(filename, flipped_horizontally=True),
    ]


class PlayerCharacter(arcade.Sprite):
    """Player Sprite"""

    def __init__(self):

        # Set up parent class
        super().__init__()

        # Default to face-right
        self.character_face_direction = RIGHT_FACING

        # Used for flipping between image sequences
        self.cur_texture = 0
        self.scale = CHARACTER_SCALING

        # Track our state
        self.jumping = False
        self.climbing = False
        self.is_on_ladder = False

        # --- Load Textures ---

```

(continues on next page)

(continued from previous page)

```
        # Images from Kenney.nl's Asset Pack 3
        main_path = ":resources:images/animated_characters/male_person/malePerson"

        # Load textures for idle standing
        self.idle_texture_pair = load_texture_pair(f"{main_path}_idle.png")
        self.jump_texture_pair = load_texture_pair(f"{main_path}_jump.png")
        self.fall_texture_pair = load_texture_pair(f"{main_path}_fall.png")

        # Load textures for walking
        self.walk_textures = []
        for i in range(8):
            texture = 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]

        # Hit box will be set based on the first image used. If you want to specify
        # a different hit box, you can do it like the code below.
        # set_hit_box = [[-22, -64], [22, -64], [22, 28], [-22, 28]]
        self.hit_box = self.texture.hit_box_points

    def update_animation(self, delta_time: float = 1 / 60):

        # Figure out if we need to flip face left or right
        if self.change_x < 0 and self.character_face_direction == RIGHT_FACING:
            self.character_face_direction = LEFT_FACING
        elif self.change_x > 0 and self.character_face_direction == LEFT_FACING:
            self.character_face_direction = RIGHT_FACING

        # Climbing animation
        if self.is_on_ladder:
            self.climbing = True
        if not self.is_on_ladder and self.climbing:
            self.climbing = False
        if self.climbing and abs(self.change_y) > 1:
            self.cur_texture += 1
            if self.cur_texture > 7:
                self.cur_texture = 0
        if self.climbing:
            self.texture = self.climbing_textures[self.cur_texture // 4]
            return

        # Jumping animation
        if self.change_y > 0 and not self.is_on_ladder:
```

(continues on next page)

(continued from previous page)

```
                self.texture = self.jump_texture_pair[self.character_face_direction]
                return
            elif self.change_y < 0 and not self.is_on_ladder:
                self.texture = self.fall_texture_pair[self.character_face_direction]
                return

            # Idle animation
            if self.change_x == 0:
                self.texture = self.idle_texture_pair[self.character_face_direction]
                return

            # 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
            ]


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

    def __init__(self):
        """
        Initializer for the game
        """

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

        # Set the path to start with this program
        file_path = os.path.dirname(os.path.abspath(__file__))
        os.chdir(file_path)

        # Track the current state of what key is pressed
        self.left_pressed = False
        self.right_pressed = False
        self.up_pressed = False
        self.down_pressed = False
        self.jump_needs_reset = False

        # Our TileMap Object
        self.tile_map = None

        # Our Scene Object
        self.scene = None

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

(continues on next page)

(continued from previous page)

```python

        # Our 'physics' engine
        self.physics_engine = None

        # A Camera that can be used for scrolling the screen
        self.camera = None

        # A Camera that can be used to draw GUI elements
        self.gui_camera = None

        self.end_of_map = 0

        # Keep track of the score
        self.score = 0

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

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

        # Set up the Cameras
        self.camera = arcade.Camera(self.width, self.height)
        self.gui_camera = arcade.Camera(self.width, self.height)

        # Map name
        map_name = ":resources:tiled_maps/map_with_ladders.json"

        # Layer Specific Options for the Tilemap
        layer_options = {
            LAYER_NAME_PLATFORMS: {
                "use_spatial_hash": True,
            },
            LAYER_NAME_MOVING_PLATFORMS: {
                "use_spatial_hash": False,
            },
            LAYER_NAME_LADDERS: {
                "use_spatial_hash": True,
            },
            LAYER_NAME_COINS: {
                "use_spatial_hash": True,
            },
        }

        # Load in TileMap
        self.tile_map = arcade.load_tilemap(map_name, TILE_SCALING, layer_options)

        # Initiate New Scene with our TileMap, this will automatically add all layers
        # from the map as SpriteLists in the scene in the proper order.
        self.scene = arcade.Scene.from_tilemap(self.tile_map)
```

(continues on next page)

(continued from previous page)

```python

        # Keep track of the score
        self.score = 0

        # Set up the player, specifically placing it at these coordinates.
        self.player_sprite = PlayerCharacter()
        self.player_sprite.center_x = PLAYER_START_X
        self.player_sprite.center_y = PLAYER_START_Y
        self.scene.add_sprite(LAYER_NAME_PLAYER, self.player_sprite)

        # Calculate the right edge of the my_map in pixels
        self.end_of_map = self.tile_map.width * GRID_PIXEL_SIZE

        # --- Other stuff
        # Set the background color
        if self.tile_map.background_color:
            arcade.set_background_color(self.tile_map.background_color)

        # Create the 'physics engine'
        self.physics_engine = arcade.PhysicsEnginePlatformer(
            self.player_sprite,
            platforms=self.scene[LAYER_NAME_MOVING_PLATFORMS],
            gravity_constant=GRAVITY,
            ladders=self.scene[LAYER_NAME_LADDERS],
            walls=self.scene[LAYER_NAME_PLATFORMS]
        )

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

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

        # Activate the game camera
        self.camera.use()

        # Draw our Scene
        self.scene.draw()

        # Activate the GUI camera before drawing GUI elements
        self.gui_camera.use()

        # Draw our score on the screen, scrolling it with the viewport
        score_text = f"Score: {self.score}"
        arcade.draw_text(
            score_text,
            10,
            10,
            arcade.csscolor.BLACK,
            18,
        )

```

(continues on next page)

(continued from previous page)

```
        # Draw hit boxes.
        # for wall in self.wall_list:
        #     wall.draw_hit_box(arcade.color.BLACK, 3)
        #
        # self.player_sprite.draw_hit_box(arcade.color.RED, 3)

    def process_keychange(self):
        """
        Called when we change a key up/down or we move on/off a ladder.
        """
        # Process up/down
        if self.up_pressed and not self.down_pressed:
            if self.physics_engine.is_on_ladder():
                self.player_sprite.change_y = PLAYER_MOVEMENT_SPEED
            elif (
                self.physics_engine.can_jump(y_distance=10)
                and not self.jump_needs_reset
            ):
                self.player_sprite.change_y = PLAYER_JUMP_SPEED
                self.jump_needs_reset = True
                arcade.play_sound(self.jump_sound)
        elif self.down_pressed and not self.up_pressed:
            if self.physics_engine.is_on_ladder():
                self.player_sprite.change_y = -PLAYER_MOVEMENT_SPEED

        # Process up/down when on a ladder and no movement
        if self.physics_engine.is_on_ladder():
            if not self.up_pressed and not self.down_pressed:
                self.player_sprite.change_y = 0
            elif self.up_pressed and self.down_pressed:
                self.player_sprite.change_y = 0

        # Process left/right
        if self.right_pressed and not self.left_pressed:
            self.player_sprite.change_x = PLAYER_MOVEMENT_SPEED
        elif self.left_pressed and not self.right_pressed:
            self.player_sprite.change_x = -PLAYER_MOVEMENT_SPEED
        else:
            self.player_sprite.change_x = 0

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

        if key == arcade.key.UP or key == arcade.key.W:
            self.up_pressed = True
        elif key == arcade.key.DOWN or key == arcade.key.S:
            self.down_pressed = True
        elif key == arcade.key.LEFT or key == arcade.key.A:
            self.left_pressed = True
        elif key == arcade.key.RIGHT or key == arcade.key.D:
            self.right_pressed = True

```

(continues on next page)

(continued from previous page)

```
        self.process_keychange()

    def on_key_release(self, key, modifiers):
        """Called when the user releases a key."""

        if key == arcade.key.UP or key == arcade.key.W:
            self.up_pressed = False
            self.jump_needs_reset = False
        elif key == arcade.key.DOWN or key == arcade.key.S:
            self.down_pressed = False
        elif key == arcade.key.LEFT or key == arcade.key.A:
            self.left_pressed = False
        elif key == arcade.key.RIGHT or key == arcade.key.D:
            self.right_pressed = False

        self.process_keychange()

    def center_camera_to_player(self):
        screen_center_x = self.player_sprite.center_x - (self.camera.viewport_width / 2)
        screen_center_y = self.player_sprite.center_y - (
            self.camera.viewport_height / 2
        )
        if screen_center_x < 0:
            screen_center_x = 0
        if screen_center_y < 0:
            screen_center_y = 0
        player_centered = screen_center_x, screen_center_y

        self.camera.move_to(player_centered, 0.2)

    def on_update(self, delta_time):
        """Movement and game logic"""

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

        # Update animations
        if self.physics_engine.can_jump():
            self.player_sprite.can_jump = False
        else:
            self.player_sprite.can_jump = True

        if self.physics_engine.is_on_ladder() and not self.physics_engine.can_jump():
            self.player_sprite.is_on_ladder = True
            self.process_keychange()
        else:
            self.player_sprite.is_on_ladder = False
            self.process_keychange()

        # Update Animations
        self.scene.update_animation(
            delta_time, [LAYER_NAME_COINS, LAYER_NAME_BACKGROUND, LAYER_NAME_PLAYER]
```

(continues on next page)

(continued from previous page)

```
        )

        # Update walls, used with moving platforms
        self.scene.update([LAYER_NAME_MOVING_PLATFORMS])

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

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

            # Figure out how many points this coin is worth
            if "Points" not in coin.properties:
                print("Warning, collected a coin without a Points property.")
            else:
                points = int(coin.properties["Points"])
                self.score += points

            # Remove the coin
            coin.remove_from_sprite_lists()
            arcade.play_sound(self.collect_coin_sound)

        # Position the camera
        self.center_camera_to_player()


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


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

8.12.1 Source Code

Listing 49: Animate the player character

```
"""
Platformer Game

python -m arcade.examples.platform_tutorial.11_animate_character
"""
import os

import arcade
```

(continues on next page)

(continued from previous page)

```python

# Constants
SCREEN_WIDTH = 1000
SCREEN_HEIGHT = 650
SCREEN_TITLE = "Platformer"

# Constants used to scale our sprites from their original size
TILE_SCALING = 0.5
CHARACTER_SCALING = TILE_SCALING * 2
COIN_SCALING = TILE_SCALING
SPRITE_PIXEL_SIZE = 128
GRID_PIXEL_SIZE = SPRITE_PIXEL_SIZE * TILE_SCALING

# Movement speed of player, in pixels per frame
PLAYER_MOVEMENT_SPEED = 7
GRAVITY = 1.5
PLAYER_JUMP_SPEED = 30

PLAYER_START_X = SPRITE_PIXEL_SIZE * TILE_SCALING * 2
PLAYER_START_Y = SPRITE_PIXEL_SIZE * TILE_SCALING * 1

# Constants used to track if the player is facing left or right
RIGHT_FACING = 0
LEFT_FACING = 1

LAYER_NAME_MOVING_PLATFORMS = "Moving Platforms"
LAYER_NAME_PLATFORMS = "Platforms"
LAYER_NAME_COINS = "Coins"
LAYER_NAME_BACKGROUND = "Background"
LAYER_NAME_LADDERS = "Ladders"
LAYER_NAME_PLAYER = "Player"


def load_texture_pair(filename):
    """
    Load a texture pair, with the second being a mirror image.
    """
    return [
        arcade.load_texture(filename),
        arcade.load_texture(filename, flipped_horizontally=True),
    ]


class PlayerCharacter(arcade.Sprite):
    """Player Sprite"""

    def __init__(self):

        # Set up parent class
        super().__init__()

        # Default to face-right
```

(continues on next page)

(continued from previous page)

```
        self.character_face_direction = RIGHT_FACING

        # Used for flipping between image sequences
        self.cur_texture = 0
        self.scale = CHARACTER_SCALING

        # Track our state
        self.jumping = False
        self.climbing = False
        self.is_on_ladder = False

        # --- Load Textures ---

        # Images from Kenney.nl's Asset Pack 3
        main_path = ":resources:images/animated_characters/male_person/malePerson"

        # Load textures for idle standing
        self.idle_texture_pair = load_texture_pair(f"{main_path}_idle.png")
        self.jump_texture_pair = load_texture_pair(f"{main_path}_jump.png")
        self.fall_texture_pair = load_texture_pair(f"{main_path}_fall.png")

        # Load textures for walking
        self.walk_textures = []
        for i in range(8):
            texture = 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]

        # Hit box will be set based on the first image used. If you want to specify
        # a different hit box, you can do it like the code below.
        # set_hit_box = [[-22, -64], [22, -64], [22, 28], [-22, 28]]
        self.hit_box = self.texture.hit_box_points

    def update_animation(self, delta_time: float = 1 / 60):

        # Figure out if we need to flip face left or right
        if self.change_x < 0 and self.character_face_direction == RIGHT_FACING:
            self.character_face_direction = LEFT_FACING
        elif self.change_x > 0 and self.character_face_direction == LEFT_FACING:
            self.character_face_direction = RIGHT_FACING

        # Climbing animation
        if self.is_on_ladder:
```

(continues on next page)

(continued from previous page)

```
            self.climbing = True
        if not self.is_on_ladder and self.climbing:
            self.climbing = False
        if self.climbing and abs(self.change_y) > 1:
            self.cur_texture += 1
            if self.cur_texture > 7:
                self.cur_texture = 0
        if self.climbing:
            self.texture = self.climbing_textures[self.cur_texture // 4]
            return

        # Jumping animation
        if self.change_y > 0 and not self.is_on_ladder:
            self.texture = self.jump_texture_pair[self.character_face_direction]
            return
        elif self.change_y < 0 and not self.is_on_ladder:
            self.texture = self.fall_texture_pair[self.character_face_direction]
            return

        # Idle animation
        if self.change_x == 0:
            self.texture = self.idle_texture_pair[self.character_face_direction]
            return

        # 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
        ]


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

    def __init__(self):
        """
        Initializer for the game
        """

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

        # Set the path to start with this program
        file_path = os.path.dirname(os.path.abspath(__file__))
        os.chdir(file_path)

        # Track the current state of what key is pressed
        self.left_pressed = False
```

(continues on next page)

(continued from previous page)

```
        self.right_pressed = False
        self.up_pressed = False
        self.down_pressed = False
        self.jump_needs_reset = False

        # Our TileMap Object
        self.tile_map = None

        # Our Scene Object
        self.scene = None

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

        # Our 'physics' engine
        self.physics_engine = None

        # A Camera that can be used for scrolling the screen
        self.camera = None

        # A Camera that can be used to draw GUI elements
        self.gui_camera = None

        self.end_of_map = 0

        # Keep track of the score
        self.score = 0

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

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

        # Set up the Cameras
        self.camera = arcade.Camera(self.width, self.height)
        self.gui_camera = arcade.Camera(self.width, self.height)

        # Map name
        map_name = ":resources:tiled_maps/map_with_ladders.json"

        # Layer Specific Options for the Tilemap
        layer_options = {
            LAYER_NAME_PLATFORMS: {
                "use_spatial_hash": True,
            },
            LAYER_NAME_MOVING_PLATFORMS: {
                "use_spatial_hash": False,
            },
            LAYER_NAME_LADDERS: {
```

(continues on next page)

(continued from previous page)

```
                "use_spatial_hash": True,
            },
            LAYER_NAME_COINS: {
                "use_spatial_hash": True,
            },
        }

        # Load in TileMap
        self.tile_map = arcade.load_tilemap(map_name, TILE_SCALING, layer_options)

        # Initiate New Scene with our TileMap, this will automatically add all layers
        # from the map as SpriteLists in the scene in the proper order.
        self.scene = arcade.Scene.from_tilemap(self.tile_map)

        # Keep track of the score
        self.score = 0

        # Set up the player, specifically placing it at these coordinates.
        self.player_sprite = PlayerCharacter()
        self.player_sprite.center_x = PLAYER_START_X
        self.player_sprite.center_y = PLAYER_START_Y
        self.scene.add_sprite(LAYER_NAME_PLAYER, self.player_sprite)

        # Calculate the right edge of the my_map in pixels
        self.end_of_map = self.tile_map.width * GRID_PIXEL_SIZE

        # --- Other stuff
        # Set the background color
        if self.tile_map.background_color:
            arcade.set_background_color(self.tile_map.background_color)

        # Create the 'physics engine'
        self.physics_engine = arcade.PhysicsEnginePlatformer(
            self.player_sprite,
            platforms=self.scene[LAYER_NAME_MOVING_PLATFORMS],
            gravity_constant=GRAVITY,
            ladders=self.scene[LAYER_NAME_LADDERS],
            walls=self.scene[LAYER_NAME_PLATFORMS]
        )

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

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

        # Activate the game camera
        self.camera.use()

        # Draw our Scene
        self.scene.draw()

```

(continues on next page)

(continued from previous page)

```
        # Activate the GUI camera before drawing GUI elements
        self.gui_camera.use()

        # Draw our score on the screen, scrolling it with the viewport
        score_text = f"Score: {self.score}"
        arcade.draw_text(
            score_text,
            10,
            10,
            arcade.csscolor.BLACK,
            18,
        )

        # Draw hit boxes.
        # for wall in self.wall_list:
        #     wall.draw_hit_box(arcade.color.BLACK, 3)
        #
        # self.player_sprite.draw_hit_box(arcade.color.RED, 3)

    def process_keychange(self):
        """
        Called when we change a key up/down or we move on/off a ladder.
        """
        # Process up/down
        if self.up_pressed and not self.down_pressed:
            if self.physics_engine.is_on_ladder():
                self.player_sprite.change_y = PLAYER_MOVEMENT_SPEED
            elif (
                self.physics_engine.can_jump(y_distance=10)
                and not self.jump_needs_reset
            ):
                self.player_sprite.change_y = PLAYER_JUMP_SPEED
                self.jump_needs_reset = True
                arcade.play_sound(self.jump_sound)
        elif self.down_pressed and not self.up_pressed:
            if self.physics_engine.is_on_ladder():
                self.player_sprite.change_y = -PLAYER_MOVEMENT_SPEED

        # Process up/down when on a ladder and no movement
        if self.physics_engine.is_on_ladder():
            if not self.up_pressed and not self.down_pressed:
                self.player_sprite.change_y = 0
            elif self.up_pressed and self.down_pressed:
                self.player_sprite.change_y = 0

        # Process left/right
        if self.right_pressed and not self.left_pressed:
            self.player_sprite.change_x = PLAYER_MOVEMENT_SPEED
        elif self.left_pressed and not self.right_pressed:
            self.player_sprite.change_x = -PLAYER_MOVEMENT_SPEED
        else:
            self.player_sprite.change_x = 0
```

(continues on next page)

(continued from previous page)

```

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

        if key == arcade.key.UP or key == arcade.key.W:
            self.up_pressed = True
        elif key == arcade.key.DOWN or key == arcade.key.S:
            self.down_pressed = True
        elif key == arcade.key.LEFT or key == arcade.key.A:
            self.left_pressed = True
        elif key == arcade.key.RIGHT or key == arcade.key.D:
            self.right_pressed = True

        self.process_keychange()

    def on_key_release(self, key, modifiers):
        """Called when the user releases a key."""

        if key == arcade.key.UP or key == arcade.key.W:
            self.up_pressed = False
            self.jump_needs_reset = False
        elif key == arcade.key.DOWN or key == arcade.key.S:
            self.down_pressed = False
        elif key == arcade.key.LEFT or key == arcade.key.A:
            self.left_pressed = False
        elif key == arcade.key.RIGHT or key == arcade.key.D:
            self.right_pressed = False

        self.process_keychange()

    def center_camera_to_player(self):
        screen_center_x = self.player_sprite.center_x - (self.camera.viewport_width / 2)
        screen_center_y = self.player_sprite.center_y - (
            self.camera.viewport_height / 2
        )
        if screen_center_x < 0:
            screen_center_x = 0
        if screen_center_y < 0:
            screen_center_y = 0
        player_centered = screen_center_x, screen_center_y

        self.camera.move_to(player_centered, 0.2)

    def on_update(self, delta_time):
        """Movement and game logic"""

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

        # Update animations
        if self.physics_engine.can_jump():
            self.player_sprite.can_jump = False
```

(continues on next page)

(continued from previous page)

```python
        else:
            self.player_sprite.can_jump = True

        if self.physics_engine.is_on_ladder() and not self.physics_engine.can_jump():
            self.player_sprite.is_on_ladder = True
            self.process_keychange()
        else:
            self.player_sprite.is_on_ladder = False
            self.process_keychange()

        # Update Animations
        self.scene.update_animation(
            delta_time, [LAYER_NAME_COINS, LAYER_NAME_BACKGROUND, LAYER_NAME_PLAYER]
        )

        # Update walls, used with moving platforms
        self.scene.update([LAYER_NAME_MOVING_PLATFORMS])

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

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

            # Figure out how many points this coin is worth
            if "Points" not in coin.properties:
                print("Warning, collected a coin without a Points property.")
            else:
                points = int(coin.properties["Points"])
                self.score += points

            # Remove the coin
            coin.remove_from_sprite_lists()
            arcade.play_sound(self.collect_coin_sound)

        # Position the camera
        self.center_camera_to_player()


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


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

8.13 Step 13 - Add Enemies

Listing 50: Animate Characters

```python
"""
Platformer Game

python -m arcade.examples.platform_tutorial.11_animate_character
"""
import math
import os

import arcade

# Constants
SCREEN_WIDTH = 1000
SCREEN_HEIGHT = 650
SCREEN_TITLE = "Platformer"

# Constants used to scale our sprites from their original size
TILE_SCALING = 0.5
CHARACTER_SCALING = TILE_SCALING * 2
COIN_SCALING = TILE_SCALING
SPRITE_PIXEL_SIZE = 128
GRID_PIXEL_SIZE = SPRITE_PIXEL_SIZE * TILE_SCALING

# Movement speed of player, in pixels per frame
PLAYER_MOVEMENT_SPEED = 7
GRAVITY = 1.5
PLAYER_JUMP_SPEED = 30

PLAYER_START_X = 2
PLAYER_START_Y = 1

# Constants used to track if the player is facing left or right
RIGHT_FACING = 0
LEFT_FACING = 1

LAYER_NAME_MOVING_PLATFORMS = "Moving Platforms"
LAYER_NAME_PLATFORMS = "Platforms"
LAYER_NAME_COINS = "Coins"
LAYER_NAME_BACKGROUND = "Background"
LAYER_NAME_LADDERS = "Ladders"
LAYER_NAME_PLAYER = "Player"
LAYER_NAME_ENEMIES = "Enemies"


def load_texture_pair(filename):
    """
    Load a texture pair, with the second being a mirror image.
    """
    return [
```

(continues on next page)

(continued from previous page)

```python
        arcade.load_texture(filename),
        arcade.load_texture(filename, flipped_horizontally=True),
    ]


class Entity(arcade.Sprite):
    def __init__(self, name_folder, name_file):
        super().__init__()

        # Default to facing right
        self.facing_direction = RIGHT_FACING

        # Used for image sequences
        self.cur_texture = 0
        self.scale = CHARACTER_SCALING
        self.character_face_direction = RIGHT_FACING

        main_path = f":resources:images/animated_characters/{name_folder}/{name_file}"

        self.idle_texture_pair = load_texture_pair(f"{main_path}_idle.png")
        self.jump_texture_pair = load_texture_pair(f"{main_path}_jump.png")
        self.fall_texture_pair = load_texture_pair(f"{main_path}_fall.png")

        # Load textures for walking
        self.walk_textures = []
        for i in range(8):
            texture = 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]

        # Hit box will be set based on the first image used. If you want to specify
        # a different hit box, you can do it like the code below.
        # set_hit_box = [[-22, -64], [22, -64], [22, 28], [-22, 28]]
        self.hit_box = self.texture.hit_box_points


class Enemy(Entity):
    def __init__(self, name_folder, name_file):

        # Setup parent class
        super().__init__(name_folder, name_file)


```

(continues on next page)

(continued from previous page)

```python
class RobotEnemy(Enemy):
    def __init__(self):

        # Set up parent class
        super().__init__("robot", "robot")


class ZombieEnemy(Enemy):
    def __init__(self):

        # Set up parent class
        super().__init__("zombie", "zombie")


class PlayerCharacter(Entity):
    """Player Sprite"""

    def __init__(self):

        # Set up parent class
        super().__init__("male_person", "malePerson")

        # Track our state
        self.jumping = False
        self.climbing = False
        self.is_on_ladder = False

    def update_animation(self, delta_time: float = 1 / 60):

        # Figure out if we need to flip face left or right
        if self.change_x < 0 and self.facing_direction == RIGHT_FACING:
            self.facing_direction = LEFT_FACING
        elif self.change_x > 0 and self.facing_direction == LEFT_FACING:
            self.facing_direction = RIGHT_FACING

        # Climbing animation
        if self.is_on_ladder:
            self.climbing = True
        if not self.is_on_ladder and self.climbing:
            self.climbing = False
        if self.climbing and abs(self.change_y) > 1:
            self.cur_texture += 1
            if self.cur_texture > 7:
                self.cur_texture = 0
        if self.climbing:
            self.texture = self.climbing_textures[self.cur_texture // 4]
            return

        # Jumping animation
        if self.change_y > 0 and not self.is_on_ladder:
            self.texture = self.jump_texture_pair[self.facing_direction]
            return
```

(continues on next page)

(continued from previous page)

```
        elif self.change_y < 0 and not self.is_on_ladder:
            self.texture = self.fall_texture_pair[self.facing_direction]
            return

        # Idle animation
        if self.change_x == 0:
            self.texture = self.idle_texture_pair[self.facing_direction]
            return

        # Walking animation
        self.cur_texture += 1
        if self.cur_texture > 7:
            self.cur_texture = 0
        self.texture = self.walk_textures[self.cur_texture][self.facing_direction]


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

    def __init__(self):
        """
        Initializer for the game
        """

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

        # Set the path to start with this program
        file_path = os.path.dirname(os.path.abspath(__file__))
        os.chdir(file_path)

        # Track the current state of what key is pressed
        self.left_pressed = False
        self.right_pressed = False
        self.up_pressed = False
        self.down_pressed = False
        self.jump_needs_reset = False

        # Our TileMap Object
        self.tile_map = None

        # Our Scene Object
        self.scene = None

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

        # Our 'physics' engine
        self.physics_engine = None

```

(continues on next page)

(continued from previous page)

```python
        # A Camera that can be used for scrolling the screen
        self.camera = None

        # A Camera that can be used to draw GUI elements
        self.gui_camera = None

        self.end_of_map = 0

        # Keep track of the score
        self.score = 0

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

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

        # Set up the Cameras
        self.camera = arcade.Camera(self.width, self.height)
        self.gui_camera = arcade.Camera(self.width, self.height)

        # Map name
        map_name = ":resources:tiled_maps/map_with_ladders.json"

        # Layer Specific Options for the Tilemap
        layer_options = {
            LAYER_NAME_PLATFORMS: {
                "use_spatial_hash": True,
            },
            LAYER_NAME_MOVING_PLATFORMS: {
                "use_spatial_hash": False,
            },
            LAYER_NAME_LADDERS: {
                "use_spatial_hash": True,
            },
            LAYER_NAME_COINS: {
                "use_spatial_hash": True,
            },
        }

        # Load in TileMap
        self.tile_map = arcade.load_tilemap(map_name, TILE_SCALING, layer_options)

        # Initiate New Scene with our TileMap, this will automatically add all layers
        # from the map as SpriteLists in the scene in the proper order.
        self.scene = arcade.Scene.from_tilemap(self.tile_map)

        # Keep track of the score
        self.score = 0

```

(continues on next page)

(continued from previous page)

```
        # Set up the player, specifically placing it at these coordinates.
        self.player_sprite = PlayerCharacter()
        self.player_sprite.center_x = (
            self.tile_map.tile_width * TILE_SCALING * PLAYER_START_X
        )
        self.player_sprite.center_y = (
            self.tile_map.tile_height * TILE_SCALING * PLAYER_START_Y
        )
        self.scene.add_sprite(LAYER_NAME_PLAYER, self.player_sprite)

        # Calculate the right edge of the my_map in pixels
        self.end_of_map = self.tile_map.width * GRID_PIXEL_SIZE

        # -- Enemies
        enemies_layer = self.tile_map.object_lists[LAYER_NAME_ENEMIES]

        for my_object in enemies_layer:
            cartesian = self.tile_map.get_cartesian(
                my_object.shape[0], my_object.shape[1]
            )
            enemy_type = my_object.properties["type"]
            if enemy_type == "robot":
                enemy = RobotEnemy()
            elif enemy_type == "zombie":
                enemy = ZombieEnemy()
            else:
                raise Exception(f"Unknown enemy type {enemy_type}.")
            enemy.center_x = math.floor(
                cartesian[0] * TILE_SCALING * self.tile_map.tile_width
            )
            enemy.center_y = math.floor(
                (cartesian[1] + 1) * (self.tile_map.tile_height * TILE_SCALING)
            )
            self.scene.add_sprite(LAYER_NAME_ENEMIES, enemy)

        # --- Other stuff
        # Set the background color
        if self.tile_map.background_color:
            arcade.set_background_color(self.tile_map.background_color)

        # Create the 'physics engine'
        self.physics_engine = arcade.PhysicsEnginePlatformer(
            self.player_sprite,
            platforms=self.scene[LAYER_NAME_MOVING_PLATFORMS],
            gravity_constant=GRAVITY,
            ladders=self.scene[LAYER_NAME_LADDERS],
            walls=self.scene[LAYER_NAME_PLATFORMS]
        )

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

```

(continues on next page)

(continued from previous page)

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

        # Activate the game camera
        self.camera.use()

        # Draw our Scene
        self.scene.draw()

        # Activate the GUI camera before drawing GUI elements
        self.gui_camera.use()

        # Draw our score on the screen, scrolling it with the viewport
        score_text = f"Score: {self.score}"
        arcade.draw_text(
            score_text,
            10,
            10,
            arcade.csscolor.BLACK,
            18,
        )

    def process_keychange(self):
        """
        Called when we change a key up/down or we move on/off a ladder.
        """
        # Process up/down
        if self.up_pressed and not self.down_pressed:
            if self.physics_engine.is_on_ladder():
                self.player_sprite.change_y = PLAYER_MOVEMENT_SPEED
            elif (
                self.physics_engine.can_jump(y_distance=10)
                and not self.jump_needs_reset
            ):
                self.player_sprite.change_y = PLAYER_JUMP_SPEED
                self.jump_needs_reset = True
                arcade.play_sound(self.jump_sound)
        elif self.down_pressed and not self.up_pressed:
            if self.physics_engine.is_on_ladder():
                self.player_sprite.change_y = -PLAYER_MOVEMENT_SPEED

        # Process up/down when on a ladder and no movement
        if self.physics_engine.is_on_ladder():
            if not self.up_pressed and not self.down_pressed:
                self.player_sprite.change_y = 0
            elif self.up_pressed and self.down_pressed:
                self.player_sprite.change_y = 0

        # Process left/right
        if self.right_pressed and not self.left_pressed:
            self.player_sprite.change_x = PLAYER_MOVEMENT_SPEED
        elif self.left_pressed and not self.right_pressed:
```

(continues on next page)

(continued from previous page)

```
            self.player_sprite.change_x = -PLAYER_MOVEMENT_SPEED
        else:
            self.player_sprite.change_x = 0

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

        if key == arcade.key.UP or key == arcade.key.W:
            self.up_pressed = True
        elif key == arcade.key.DOWN or key == arcade.key.S:
            self.down_pressed = True
        elif key == arcade.key.LEFT or key == arcade.key.A:
            self.left_pressed = True
        elif key == arcade.key.RIGHT or key == arcade.key.D:
            self.right_pressed = True

        self.process_keychange()

    def on_key_release(self, key, modifiers):
        """Called when the user releases a key."""

        if key == arcade.key.UP or key == arcade.key.W:
            self.up_pressed = False
            self.jump_needs_reset = False
        elif key == arcade.key.DOWN or key == arcade.key.S:
            self.down_pressed = False
        elif key == arcade.key.LEFT or key == arcade.key.A:
            self.left_pressed = False
        elif key == arcade.key.RIGHT or key == arcade.key.D:
            self.right_pressed = False

        self.process_keychange()

    def center_camera_to_player(self):
        screen_center_x = self.player_sprite.center_x - (self.camera.viewport_width / 2)
        screen_center_y = self.player_sprite.center_y - (
            self.camera.viewport_height / 2
        )
        if screen_center_x < 0:
            screen_center_x = 0
        if screen_center_y < 0:
            screen_center_y = 0
        player_centered = screen_center_x, screen_center_y

        self.camera.move_to(player_centered, 0.2)

    def on_update(self, delta_time):
        """Movement and game logic"""

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

```

(continues on next page)

(continued from previous page)

```
        # Update animations
        if self.physics_engine.can_jump():
            self.player_sprite.can_jump = False
        else:
            self.player_sprite.can_jump = True

        if self.physics_engine.is_on_ladder() and not self.physics_engine.can_jump():
            self.player_sprite.is_on_ladder = True
            self.process_keychange()
        else:
            self.player_sprite.is_on_ladder = False
            self.process_keychange()

        # Update Animations
        self.scene.update_animation(
            delta_time,
            [
                LAYER_NAME_COINS,
                LAYER_NAME_BACKGROUND,
                LAYER_NAME_PLAYER,
                LAYER_NAME_ENEMIES,
            ],
        )

        # Update walls, used with moving platforms
        self.scene.update([LAYER_NAME_MOVING_PLATFORMS])

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

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

            # Figure out how many points this coin is worth
            if "Points" not in coin.properties:
                print("Warning, collected a coin without a Points property.")
            else:
                points = int(coin.properties["Points"])
                self.score += points

            # Remove the coin
            coin.remove_from_sprite_lists()
            arcade.play_sound(self.collect_coin_sound)

        # Position the camera
        self.center_camera_to_player()


def main():
    """Main function"""
```

(continues on next page)

(continued from previous page)

```python
    window = MyGame()
    window.setup()
    arcade.run()


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

8.13.1 Source Code

Listing 51: Add Enemies

```python
"""
Platformer Game

python -m arcade.examples.platform_tutorial.11_animate_character
"""
import math
import os

import arcade

# Constants
SCREEN_WIDTH = 1000
SCREEN_HEIGHT = 650
SCREEN_TITLE = "Platformer"

# Constants used to scale our sprites from their original size
TILE_SCALING = 0.5
CHARACTER_SCALING = TILE_SCALING * 2
COIN_SCALING = TILE_SCALING
SPRITE_PIXEL_SIZE = 128
GRID_PIXEL_SIZE = SPRITE_PIXEL_SIZE * TILE_SCALING

# Movement speed of player, in pixels per frame
PLAYER_MOVEMENT_SPEED = 7
GRAVITY = 1.5
PLAYER_JUMP_SPEED = 30

PLAYER_START_X = 2
PLAYER_START_Y = 1

# Constants used to track if the player is facing left or right
RIGHT_FACING = 0
LEFT_FACING = 1

LAYER_NAME_MOVING_PLATFORMS = "Moving Platforms"
LAYER_NAME_PLATFORMS = "Platforms"
LAYER_NAME_COINS = "Coins"
LAYER_NAME_BACKGROUND = "Background"
```

(continues on next page)

(continued from previous page)

```python
LAYER_NAME_LADDERS = "Ladders"
LAYER_NAME_PLAYER = "Player"
LAYER_NAME_ENEMIES = "Enemies"


def load_texture_pair(filename):
    """
    Load a texture pair, with the second being a mirror image.
    """
    return [
        arcade.load_texture(filename),
        arcade.load_texture(filename, flipped_horizontally=True),
    ]


class Entity(arcade.Sprite):
    def __init__(self, name_folder, name_file):
        super().__init__()

        # Default to facing right
        self.facing_direction = RIGHT_FACING

        # Used for image sequences
        self.cur_texture = 0
        self.scale = CHARACTER_SCALING
        self.character_face_direction = RIGHT_FACING

        main_path = f":resources:images/animated_characters/{name_folder}/{name_file}"

        self.idle_texture_pair = load_texture_pair(f"{main_path}_idle.png")
        self.jump_texture_pair = load_texture_pair(f"{main_path}_jump.png")
        self.fall_texture_pair = load_texture_pair(f"{main_path}_fall.png")

        # Load textures for walking
        self.walk_textures = []
        for i in range(8):
            texture = 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]

        # Hit box will be set based on the first image used. If you want to specify
        # a different hit box, you can do it like the code below.
        # set_hit_box = [[-22, -64], [22, -64], [22, 28], [-22, 28]]
```

(continues on next page)

(continued from previous page)

```
        self.hit_box = self.texture.hit_box_points


class Enemy(Entity):
    def __init__(self, name_folder, name_file):

        # Setup parent class
        super().__init__(name_folder, name_file)


class RobotEnemy(Enemy):
    def __init__(self):

        # Set up parent class
        super().__init__("robot", "robot")


class ZombieEnemy(Enemy):
    def __init__(self):

        # Set up parent class
        super().__init__("zombie", "zombie")


class PlayerCharacter(Entity):
    """Player Sprite"""

    def __init__(self):

        # Set up parent class
        super().__init__("male_person", "malePerson")

        # Track our state
        self.jumping = False
        self.climbing = False
        self.is_on_ladder = False

    def update_animation(self, delta_time: float = 1 / 60):

        # Figure out if we need to flip face left or right
        if self.change_x < 0 and self.facing_direction == RIGHT_FACING:
            self.facing_direction = LEFT_FACING
        elif self.change_x > 0 and self.facing_direction == LEFT_FACING:
            self.facing_direction = RIGHT_FACING

        # Climbing animation
        if self.is_on_ladder:
            self.climbing = True
        if not self.is_on_ladder and self.climbing:
            self.climbing = False
        if self.climbing and abs(self.change_y) > 1:
            self.cur_texture += 1
```

(continues on next page)

(continued from previous page)

```
            if self.cur_texture > 7:
                self.cur_texture = 0
        if self.climbing:
            self.texture = self.climbing_textures[self.cur_texture // 4]
            return

        # Jumping animation
        if self.change_y > 0 and not self.is_on_ladder:
            self.texture = self.jump_texture_pair[self.facing_direction]
            return
        elif self.change_y < 0 and not self.is_on_ladder:
            self.texture = self.fall_texture_pair[self.facing_direction]
            return

        # Idle animation
        if self.change_x == 0:
            self.texture = self.idle_texture_pair[self.facing_direction]
            return

        # Walking animation
        self.cur_texture += 1
        if self.cur_texture > 7:
            self.cur_texture = 0
        self.texture = self.walk_textures[self.cur_texture][self.facing_direction]


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

    def __init__(self):
        """
        Initializer for the game
        """

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

        # Set the path to start with this program
        file_path = os.path.dirname(os.path.abspath(__file__))
        os.chdir(file_path)

        # Track the current state of what key is pressed
        self.left_pressed = False
        self.right_pressed = False
        self.up_pressed = False
        self.down_pressed = False
        self.jump_needs_reset = False

        # Our TileMap Object
        self.tile_map = None
```

(continues on next page)

(continued from previous page)

```

        # Our Scene Object
        self.scene = None

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

        # Our 'physics' engine
        self.physics_engine = None

        # A Camera that can be used for scrolling the screen
        self.camera = None

        # A Camera that can be used to draw GUI elements
        self.gui_camera = None

        self.end_of_map = 0

        # Keep track of the score
        self.score = 0

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

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

        # Set up the Cameras
        self.camera = arcade.Camera(self.width, self.height)
        self.gui_camera = arcade.Camera(self.width, self.height)

        # Map name
        map_name = ":resources:tiled_maps/map_with_ladders.json"

        # Layer Specific Options for the Tilemap
        layer_options = {
            LAYER_NAME_PLATFORMS: {
                "use_spatial_hash": True,
            },
            LAYER_NAME_MOVING_PLATFORMS: {
                "use_spatial_hash": False,
            },
            LAYER_NAME_LADDERS: {
                "use_spatial_hash": True,
            },
            LAYER_NAME_COINS: {
                "use_spatial_hash": True,
            },
        }

```

(continues on next page)

(continued from previous page)

```
        # Load in TileMap
        self.tile_map = arcade.load_tilemap(map_name, TILE_SCALING, layer_options)

        # Initiate New Scene with our TileMap, this will automatically add all layers
        # from the map as SpriteLists in the scene in the proper order.
        self.scene = arcade.Scene.from_tilemap(self.tile_map)

        # Keep track of the score
        self.score = 0

        # Set up the player, specifically placing it at these coordinates.
        self.player_sprite = PlayerCharacter()
        self.player_sprite.center_x = (
            self.tile_map.tile_width * TILE_SCALING * PLAYER_START_X
        )
        self.player_sprite.center_y = (
            self.tile_map.tile_height * TILE_SCALING * PLAYER_START_Y
        )
        self.scene.add_sprite(LAYER_NAME_PLAYER, self.player_sprite)

        # Calculate the right edge of the my_map in pixels
        self.end_of_map = self.tile_map.width * GRID_PIXEL_SIZE

        # -- Enemies
        enemies_layer = self.tile_map.object_lists[LAYER_NAME_ENEMIES]

        for my_object in enemies_layer:
            cartesian = self.tile_map.get_cartesian(
                my_object.shape[0], my_object.shape[1]
            )
            enemy_type = my_object.properties["type"]
            if enemy_type == "robot":
                enemy = RobotEnemy()
            elif enemy_type == "zombie":
                enemy = ZombieEnemy()
            else:
                raise Exception(f"Unknown enemy type {enemy_type}.")
            enemy.center_x = math.floor(
                cartesian[0] * TILE_SCALING * self.tile_map.tile_width
            )
            enemy.center_y = math.floor(
                (cartesian[1] + 1) * (self.tile_map.tile_height * TILE_SCALING)
            )
            self.scene.add_sprite(LAYER_NAME_ENEMIES, enemy)

        # --- Other stuff
        # Set the background color
        if self.tile_map.background_color:
            arcade.set_background_color(self.tile_map.background_color)

        # Create the 'physics engine'
        self.physics_engine = arcade.PhysicsEnginePlatformer(
```

(continues on next page)

(continued from previous page)

```
            self.player_sprite,
            platforms=self.scene[LAYER_NAME_MOVING_PLATFORMS],
            gravity_constant=GRAVITY,
            ladders=self.scene[LAYER_NAME_LADDERS],
            walls=self.scene[LAYER_NAME_PLATFORMS]
        )

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

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

        # Activate the game camera
        self.camera.use()

        # Draw our Scene
        self.scene.draw()

        # Activate the GUI camera before drawing GUI elements
        self.gui_camera.use()

        # Draw our score on the screen, scrolling it with the viewport
        score_text = f"Score: {self.score}"
        arcade.draw_text(
            score_text,
            10,
            10,
            arcade.csscolor.BLACK,
            18,
        )

    def process_keychange(self):
        """
        Called when we change a key up/down or we move on/off a ladder.
        """
        # Process up/down
        if self.up_pressed and not self.down_pressed:
            if self.physics_engine.is_on_ladder():
                self.player_sprite.change_y = PLAYER_MOVEMENT_SPEED
            elif (
                self.physics_engine.can_jump(y_distance=10)
                and not self.jump_needs_reset
            ):
                self.player_sprite.change_y = PLAYER_JUMP_SPEED
                self.jump_needs_reset = True
                arcade.play_sound(self.jump_sound)
        elif self.down_pressed and not self.up_pressed:
            if self.physics_engine.is_on_ladder():
                self.player_sprite.change_y = -PLAYER_MOVEMENT_SPEED

        # Process up/down when on a ladder and no movement
```

(continues on next page)

(continued from previous page)

```
        if self.physics_engine.is_on_ladder():
            if not self.up_pressed and not self.down_pressed:
                self.player_sprite.change_y = 0
            elif self.up_pressed and self.down_pressed:
                self.player_sprite.change_y = 0

        # Process left/right
        if self.right_pressed and not self.left_pressed:
            self.player_sprite.change_x = PLAYER_MOVEMENT_SPEED
        elif self.left_pressed and not self.right_pressed:
            self.player_sprite.change_x = -PLAYER_MOVEMENT_SPEED
        else:
            self.player_sprite.change_x = 0

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

        if key == arcade.key.UP or key == arcade.key.W:
            self.up_pressed = True
        elif key == arcade.key.DOWN or key == arcade.key.S:
            self.down_pressed = True
        elif key == arcade.key.LEFT or key == arcade.key.A:
            self.left_pressed = True
        elif key == arcade.key.RIGHT or key == arcade.key.D:
            self.right_pressed = True

        self.process_keychange()

    def on_key_release(self, key, modifiers):
        """Called when the user releases a key."""

        if key == arcade.key.UP or key == arcade.key.W:
            self.up_pressed = False
            self.jump_needs_reset = False
        elif key == arcade.key.DOWN or key == arcade.key.S:
            self.down_pressed = False
        elif key == arcade.key.LEFT or key == arcade.key.A:
            self.left_pressed = False
        elif key == arcade.key.RIGHT or key == arcade.key.D:
            self.right_pressed = False

        self.process_keychange()

    def center_camera_to_player(self):
        screen_center_x = self.player_sprite.center_x - (self.camera.viewport_width / 2)
        screen_center_y = self.player_sprite.center_y - (
            self.camera.viewport_height / 2
        )
        if screen_center_x < 0:
            screen_center_x = 0
        if screen_center_y < 0:
            screen_center_y = 0
```

(continues on next page)

(continued from previous page)

```
        player_centered = screen_center_x, screen_center_y

        self.camera.move_to(player_centered, 0.2)

    def on_update(self, delta_time):
        """Movement and game logic"""

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

        # Update animations
        if self.physics_engine.can_jump():
            self.player_sprite.can_jump = False
        else:
            self.player_sprite.can_jump = True

        if self.physics_engine.is_on_ladder() and not self.physics_engine.can_jump():
            self.player_sprite.is_on_ladder = True
            self.process_keychange()
        else:
            self.player_sprite.is_on_ladder = False
            self.process_keychange()

        # Update Animations
        self.scene.update_animation(
            delta_time,
            [
                LAYER_NAME_COINS,
                LAYER_NAME_BACKGROUND,
                LAYER_NAME_PLAYER,
                LAYER_NAME_ENEMIES,
            ],
        )

        # Update walls, used with moving platforms
        self.scene.update([LAYER_NAME_MOVING_PLATFORMS])

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

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

            # Figure out how many points this coin is worth
            if "Points" not in coin.properties:
                print("Warning, collected a coin without a Points property.")
            else:
                points = int(coin.properties["Points"])
                self.score += points

```

(continues on next page)

(continued from previous page)

```python
                # Remove the coin
                coin.remove_from_sprite_lists()
                arcade.play_sound(self.collect_coin_sound)

        # Position the camera
        self.center_camera_to_player()


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


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

8.14 Step 14 - Moving Enemies

Listing 52: Moving the enemies

```python
"""
Platformer Game

python -m arcade.examples.platform_tutorial.11_animate_character
"""
import math
import os

import arcade

# Constants
SCREEN_WIDTH = 1000
SCREEN_HEIGHT = 650
SCREEN_TITLE = "Platformer"

# Constants used to scale our sprites from their original size
TILE_SCALING = 0.5
CHARACTER_SCALING = TILE_SCALING * 2
COIN_SCALING = TILE_SCALING
SPRITE_PIXEL_SIZE = 128
GRID_PIXEL_SIZE = SPRITE_PIXEL_SIZE * TILE_SCALING

# Movement speed of player, in pixels per frame
PLAYER_MOVEMENT_SPEED = 7
GRAVITY = 1.5
PLAYER_JUMP_SPEED = 30

# How many pixels to keep as a minimum margin between the character
```

(continues on next page)

(continued from previous page)

```
# and the edge of the screen.
LEFT_VIEWPORT_MARGIN = 200
RIGHT_VIEWPORT_MARGIN = 200
BOTTOM_VIEWPORT_MARGIN = 150
TOP_VIEWPORT_MARGIN = 100

PLAYER_START_X = 2
PLAYER_START_Y = 1

# Constants used to track if the player is facing left or right
RIGHT_FACING = 0
LEFT_FACING = 1

LAYER_NAME_MOVING_PLATFORMS = "Moving Platforms"
LAYER_NAME_PLATFORMS = "Platforms"
LAYER_NAME_COINS = "Coins"
LAYER_NAME_BACKGROUND = "Background"
LAYER_NAME_LADDERS = "Ladders"
LAYER_NAME_PLAYER = "Player"
LAYER_NAME_ENEMIES = "Enemies"


def load_texture_pair(filename):
    """
    Load a texture pair, with the second being a mirror image.
    """
    return [
        arcade.load_texture(filename),
        arcade.load_texture(filename, flipped_horizontally=True),
    ]


class Entity(arcade.Sprite):
    def __init__(self, name_folder, name_file):
        super().__init__()

        # Default to facing right
        self.facing_direction = RIGHT_FACING

        # Used for image sequences
        self.cur_texture = 0
        self.scale = CHARACTER_SCALING

        main_path = f":resources:images/animated_characters/{name_folder}/{name_file}"

        self.idle_texture_pair = load_texture_pair(f"{main_path}_idle.png")
        self.jump_texture_pair = load_texture_pair(f"{main_path}_jump.png")
        self.fall_texture_pair = load_texture_pair(f"{main_path}_fall.png")

        # Load textures for walking
        self.walk_textures = []
        for i in range(8):
```

(continues on next page)

(continued from previous page)

```
            texture = 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]

        # Hit box will be set based on the first image used. If you want to specify
        # a different hit box, you can do it like the code below.
        # self.set_hit_box([[-22, -64], [22, -64], [22, 28], [-22, 28]])
        self.set_hit_box(self.texture.hit_box_points)


class Enemy(Entity):
    def __init__(self, name_folder, name_file):

        # Setup parent class
        super().__init__(name_folder, name_file)

        self.should_update_walk = 0

    def update_animation(self, delta_time: float = 1 / 60):

        # Figure out if we need to flip face left or right
        if self.change_x < 0 and self.facing_direction == RIGHT_FACING:
            self.facing_direction = LEFT_FACING
        elif self.change_x > 0 and self.facing_direction == LEFT_FACING:
            self.facing_direction = RIGHT_FACING

        # Idle animation
        if self.change_x == 0:
            self.texture = self.idle_texture_pair[self.facing_direction]
            return

        # Walking animation
        if self.should_update_walk == 3:
            self.cur_texture += 1
            if self.cur_texture > 7:
                self.cur_texture = 0
            self.texture = self.walk_textures[self.cur_texture][self.facing_direction]
            self.should_update_walk = 0
            return

        self.should_update_walk += 1


```

(continues on next page)

(continued from previous page)

```
class RobotEnemy(Enemy):
    def __init__(self):

        # Set up parent class
        super().__init__("robot", "robot")


class ZombieEnemy(Enemy):
    def __init__(self):

        # Set up parent class
        super().__init__("zombie", "zombie")


class PlayerCharacter(Entity):
    """Player Sprite"""

    def __init__(self):

        # Set up parent class
        super().__init__("male_person", "malePerson")

        # Track our state
        self.jumping = False
        self.climbing = False
        self.is_on_ladder = False

    def update_animation(self, delta_time: float = 1 / 60):

        # Figure out if we need to flip face left or right
        if self.change_x < 0 and self.facing_direction == RIGHT_FACING:
            self.facing_direction = LEFT_FACING
        elif self.change_x > 0 and self.facing_direction == LEFT_FACING:
            self.facing_direction = RIGHT_FACING

        # Climbing animation
        if self.is_on_ladder:
            self.climbing = True
        if not self.is_on_ladder and self.climbing:
            self.climbing = False
        if self.climbing and abs(self.change_y) > 1:
            self.cur_texture += 1
            if self.cur_texture > 7:
                self.cur_texture = 0
        if self.climbing:
            self.texture = self.climbing_textures[self.cur_texture // 4]
            return

        # Jumping animation
        if self.change_y > 0 and not self.is_on_ladder:
            self.texture = self.jump_texture_pair[self.facing_direction]
            return
```

(continues on next page)

(continued from previous page)

```
        elif self.change_y < 0 and not self.is_on_ladder:
            self.texture = self.fall_texture_pair[self.facing_direction]
            return

        # Idle animation
        if self.change_x == 0:
            self.texture = self.idle_texture_pair[self.facing_direction]
            return

        # Walking animation
        self.cur_texture += 1
        if self.cur_texture > 7:
            self.cur_texture = 0
        self.texture = self.walk_textures[self.cur_texture][self.facing_direction]


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

    def __init__(self):
        """
        Initializer for the game
        """

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

        # Set the path to start with this program
        file_path = os.path.dirname(os.path.abspath(__file__))
        os.chdir(file_path)

        # Track the current state of what key is pressed
        self.left_pressed = False
        self.right_pressed = False
        self.up_pressed = False
        self.down_pressed = False
        self.jump_needs_reset = False

        # Our TileMap Object
        self.tile_map = None

        # Our Scene Object
        self.scene = None

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

        # Our 'physics' engine
        self.physics_engine = None

```

(continues on next page)

(continued from previous page)

```
        # A Camera that can be used for scrolling the screen
        self.camera = None

        # A Camera that can be used to draw GUI elements
        self.gui_camera = None

        self.end_of_map = 0

        # Keep track of the score
        self.score = 0

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

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

        # Set up the Cameras
        self.camera = arcade.Camera(self.width, self.height)
        self.gui_camera = arcade.Camera(self.width, self.height)

        # Map name
        map_name = ":resources:tiled_maps/map_with_ladders.json"

        # Layer Specific Options for the Tilemap
        layer_options = {
            LAYER_NAME_PLATFORMS: {
                "use_spatial_hash": True,
            },
            LAYER_NAME_MOVING_PLATFORMS: {
                "use_spatial_hash": False,
            },
            LAYER_NAME_LADDERS: {
                "use_spatial_hash": True,
            },
            LAYER_NAME_COINS: {
                "use_spatial_hash": True,
            },
        }

        # Load in TileMap
        self.tile_map = arcade.load_tilemap(map_name, TILE_SCALING, layer_options)

        # Initiate New Scene with our TileMap, this will automatically add all layers
        # from the map as SpriteLists in the scene in the proper order.
        self.scene = arcade.Scene.from_tilemap(self.tile_map)

        # Keep track of the score
        self.score = 0

```

(continues on next page)

(continued from previous page)

```
        # Set up the player, specifically placing it at these coordinates.
        self.player_sprite = PlayerCharacter()
        self.player_sprite.center_x = (
            self.tile_map.tile_width * TILE_SCALING * PLAYER_START_X
        )
        self.player_sprite.center_y = (
            self.tile_map.tile_height * TILE_SCALING * PLAYER_START_Y
        )
        self.scene.add_sprite(LAYER_NAME_PLAYER, self.player_sprite)

        # Calculate the right edge of the my_map in pixels
        self.end_of_map = self.tile_map.width * GRID_PIXEL_SIZE

        # -- Enemies
        enemies_layer = self.tile_map.object_lists[LAYER_NAME_ENEMIES]

        for my_object in enemies_layer:
            cartesian = self.tile_map.get_cartesian(
                my_object.shape[0], my_object.shape[1]
            )
            enemy_type = my_object.properties["type"]
            if enemy_type == "robot":
                enemy = RobotEnemy()
            elif enemy_type == "zombie":
                enemy = ZombieEnemy()
            enemy.center_x = math.floor(
                cartesian[0] * TILE_SCALING * self.tile_map.tile_width
            )
            enemy.center_y = math.floor(
                (cartesian[1] + 1) * (self.tile_map.tile_height * TILE_SCALING)
            )
            if "boundary_left" in my_object.properties:
                enemy.boundary_left = my_object.properties["boundary_left"]
            if "boundary_right" in my_object.properties:
                enemy.boundary_right = my_object.properties["boundary_right"]
            if "change_x" in my_object.properties:
                enemy.change_x = my_object.properties["change_x"]
            self.scene.add_sprite(LAYER_NAME_ENEMIES, enemy)

        # --- Other stuff
        # Set the background color
        if self.tile_map.background_color:
            arcade.set_background_color(self.tile_map.background_color)

        # Create the 'physics engine'
        self.physics_engine = arcade.PhysicsEnginePlatformer(
            self.player_sprite,
            platforms=self.scene[LAYER_NAME_MOVING_PLATFORMS],
            gravity_constant=GRAVITY,
            ladders=self.scene[LAYER_NAME_LADDERS],
            walls=self.scene[LAYER_NAME_PLATFORMS]
        )
```

(continues on next page)

(continued from previous page)

```

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

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

        # Activate the game camera
        self.camera.use()

        # Draw our Scene
        self.scene.draw()

        # Activate the GUI camera before drawing GUI elements
        self.gui_camera.use()

        # Draw our score on the screen, scrolling it with the viewport
        score_text = f"Score: {self.score}"
        arcade.draw_text(
            score_text,
            10,
            10,
            arcade.csscolor.BLACK,
            18,
        )

        # Draw hit boxes.
        # for wall in self.wall_list:
        #     wall.draw_hit_box(arcade.color.BLACK, 3)
        #
        # self.player_sprite.draw_hit_box(arcade.color.RED, 3)

    def process_keychange(self):
        """
        Called when we change a key up/down or we move on/off a ladder.
        """
        # Process up/down
        if self.up_pressed and not self.down_pressed:
            if self.physics_engine.is_on_ladder():
                self.player_sprite.change_y = PLAYER_MOVEMENT_SPEED
            elif (
                self.physics_engine.can_jump(y_distance=10)
                and not self.jump_needs_reset
            ):
                self.player_sprite.change_y = PLAYER_JUMP_SPEED
                self.jump_needs_reset = True
                arcade.play_sound(self.jump_sound)
        elif self.down_pressed and not self.up_pressed:
            if self.physics_engine.is_on_ladder():
                self.player_sprite.change_y = -PLAYER_MOVEMENT_SPEED

        # Process up/down when on a ladder and no movement
```

(continues on next page)

(continued from previous page)

```
        if self.physics_engine.is_on_ladder():
            if not self.up_pressed and not self.down_pressed:
                self.player_sprite.change_y = 0
            elif self.up_pressed and self.down_pressed:
                self.player_sprite.change_y = 0

        # Process left/right
        if self.right_pressed and not self.left_pressed:
            self.player_sprite.change_x = PLAYER_MOVEMENT_SPEED
        elif self.left_pressed and not self.right_pressed:
            self.player_sprite.change_x = -PLAYER_MOVEMENT_SPEED
        else:
            self.player_sprite.change_x = 0

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

        if key == arcade.key.UP or key == arcade.key.W:
            self.up_pressed = True
        elif key == arcade.key.DOWN or key == arcade.key.S:
            self.down_pressed = True
        elif key == arcade.key.LEFT or key == arcade.key.A:
            self.left_pressed = True
        elif key == arcade.key.RIGHT or key == arcade.key.D:
            self.right_pressed = True

        self.process_keychange()

    def on_key_release(self, key, modifiers):
        """Called when the user releases a key."""

        if key == arcade.key.UP or key == arcade.key.W:
            self.up_pressed = False
            self.jump_needs_reset = False
        elif key == arcade.key.DOWN or key == arcade.key.S:
            self.down_pressed = False
        elif key == arcade.key.LEFT or key == arcade.key.A:
            self.left_pressed = False
        elif key == arcade.key.RIGHT or key == arcade.key.D:
            self.right_pressed = False

        self.process_keychange()

    def center_camera_to_player(self):
        screen_center_x = self.player_sprite.center_x - (self.camera.viewport_width / 2)
        screen_center_y = self.player_sprite.center_y - (
            self.camera.viewport_height / 2
        )
        if screen_center_x < 0:
            screen_center_x = 0
        if screen_center_y < 0:
            screen_center_y = 0
```

(continues on next page)

(continued from previous page)

```
        player_centered = screen_center_x, screen_center_y

        self.camera.move_to(player_centered, 0.2)

    def on_update(self, delta_time):
        """Movement and game logic"""

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

        # Update animations
        if self.physics_engine.can_jump():
            self.player_sprite.can_jump = False
        else:
            self.player_sprite.can_jump = True

        if self.physics_engine.is_on_ladder() and not self.physics_engine.can_jump():
            self.player_sprite.is_on_ladder = True
            self.process_keychange()
        else:
            self.player_sprite.is_on_ladder = False
            self.process_keychange()

        # Update Animations
        self.scene.update_animation(
            delta_time,
            [
                LAYER_NAME_COINS,
                LAYER_NAME_BACKGROUND,
                LAYER_NAME_PLAYER,
                LAYER_NAME_ENEMIES,
            ],
        )

        # Update moving platforms and enemies
        self.scene.update([LAYER_NAME_MOVING_PLATFORMS, LAYER_NAME_ENEMIES])

        # See if the enemy hit a boundary and needs to reverse direction.
        for enemy in self.scene[LAYER_NAME_ENEMIES]:
            if (
                enemy.boundary_right
                and enemy.right > enemy.boundary_right
                and enemy.change_x > 0
            ):
                enemy.change_x *= -1

            if (
                enemy.boundary_left
                and enemy.left < enemy.boundary_left
                and enemy.change_x < 0
            ):
                enemy.change_x *= -1
```

(continues on next page)

(continued from previous page)

```

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

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

            # Figure out how many points this coin is worth
            if "Points" not in coin.properties:
                print("Warning, collected a coin without a Points property.")
            else:
                points = int(coin.properties["Points"])
                self.score += points

            # Remove the coin
            coin.remove_from_sprite_lists()
            arcade.play_sound(self.collect_coin_sound)

        # Position the camera
        self.center_camera_to_player()


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


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

8.14.1 Source Code

Listing 53: Moving the enemies

```
"""
Platformer Game

python -m arcade.examples.platform_tutorial.11_animate_character
"""
import math
import os

import arcade

# Constants
SCREEN_WIDTH = 1000
```

(continues on next page)

(continued from previous page)

```
SCREEN_HEIGHT = 650
SCREEN_TITLE = "Platformer"

# Constants used to scale our sprites from their original size
TILE_SCALING = 0.5
CHARACTER_SCALING = TILE_SCALING * 2
COIN_SCALING = TILE_SCALING
SPRITE_PIXEL_SIZE = 128
GRID_PIXEL_SIZE = SPRITE_PIXEL_SIZE * TILE_SCALING

# Movement speed of player, in pixels per frame
PLAYER_MOVEMENT_SPEED = 7
GRAVITY = 1.5
PLAYER_JUMP_SPEED = 30

# How many pixels to keep as a minimum margin between the character
# and the edge of the screen.
LEFT_VIEWPORT_MARGIN = 200
RIGHT_VIEWPORT_MARGIN = 200
BOTTOM_VIEWPORT_MARGIN = 150
TOP_VIEWPORT_MARGIN = 100

PLAYER_START_X = 2
PLAYER_START_Y = 1

# Constants used to track if the player is facing left or right
RIGHT_FACING = 0
LEFT_FACING = 1

LAYER_NAME_MOVING_PLATFORMS = "Moving Platforms"
LAYER_NAME_PLATFORMS = "Platforms"
LAYER_NAME_COINS = "Coins"
LAYER_NAME_BACKGROUND = "Background"
LAYER_NAME_LADDERS = "Ladders"
LAYER_NAME_PLAYER = "Player"
LAYER_NAME_ENEMIES = "Enemies"


def load_texture_pair(filename):
    """
    Load a texture pair, with the second being a mirror image.
    """
    return [
        arcade.load_texture(filename),
        arcade.load_texture(filename, flipped_horizontally=True),
    ]


class Entity(arcade.Sprite):
    def __init__(self, name_folder, name_file):
        super().__init__()

```

(continues on next page)

(continued from previous page)

```
        # Default to facing right
        self.facing_direction = RIGHT_FACING

        # Used for image sequences
        self.cur_texture = 0
        self.scale = CHARACTER_SCALING

        main_path = f":resources:images/animated_characters/{name_folder}/{name_file}"

        self.idle_texture_pair = load_texture_pair(f"{main_path}_idle.png")
        self.jump_texture_pair = load_texture_pair(f"{main_path}_jump.png")
        self.fall_texture_pair = load_texture_pair(f"{main_path}_fall.png")

        # Load textures for walking
        self.walk_textures = []
        for i in range(8):
            texture = 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]

        # Hit box will be set based on the first image used. If you want to specify
        # a different hit box, you can do it like the code below.
        # self.set_hit_box([[-22, -64], [22, -64], [22, 28], [-22, 28]])
        self.set_hit_box(self.texture.hit_box_points)


class Enemy(Entity):
    def __init__(self, name_folder, name_file):

        # Setup parent class
        super().__init__(name_folder, name_file)

        self.should_update_walk = 0

    def update_animation(self, delta_time: float = 1 / 60):

        # Figure out if we need to flip face left or right
        if self.change_x < 0 and self.facing_direction == RIGHT_FACING:
            self.facing_direction = LEFT_FACING
        elif self.change_x > 0 and self.facing_direction == LEFT_FACING:
            self.facing_direction = RIGHT_FACING

        # Idle animation
```

(continues on next page)

(continued from previous page)

```
        if self.change_x == 0:
            self.texture = self.idle_texture_pair[self.facing_direction]
            return

        # Walking animation
        if self.should_update_walk == 3:
            self.cur_texture += 1
            if self.cur_texture > 7:
                self.cur_texture = 0
            self.texture = self.walk_textures[self.cur_texture][self.facing_direction]
            self.should_update_walk = 0
            return

        self.should_update_walk += 1


class RobotEnemy(Enemy):
    def __init__(self):

        # Set up parent class
        super().__init__("robot", "robot")


class ZombieEnemy(Enemy):
    def __init__(self):

        # Set up parent class
        super().__init__("zombie", "zombie")


class PlayerCharacter(Entity):
    """Player Sprite"""

    def __init__(self):

        # Set up parent class
        super().__init__("male_person", "malePerson")

        # Track our state
        self.jumping = False
        self.climbing = False
        self.is_on_ladder = False

    def update_animation(self, delta_time: float = 1 / 60):

        # Figure out if we need to flip face left or right
        if self.change_x < 0 and self.facing_direction == RIGHT_FACING:
            self.facing_direction = LEFT_FACING
        elif self.change_x > 0 and self.facing_direction == LEFT_FACING:
            self.facing_direction = RIGHT_FACING

        # Climbing animation
```

(continues on next page)

(continued from previous page)

```python
        if self.is_on_ladder:
            self.climbing = True
        if not self.is_on_ladder and self.climbing:
            self.climbing = False
        if self.climbing and abs(self.change_y) > 1:
            self.cur_texture += 1
            if self.cur_texture > 7:
                self.cur_texture = 0
        if self.climbing:
            self.texture = self.climbing_textures[self.cur_texture // 4]
            return

        # Jumping animation
        if self.change_y > 0 and not self.is_on_ladder:
            self.texture = self.jump_texture_pair[self.facing_direction]
            return
        elif self.change_y < 0 and not self.is_on_ladder:
            self.texture = self.fall_texture_pair[self.facing_direction]
            return

        # Idle animation
        if self.change_x == 0:
            self.texture = self.idle_texture_pair[self.facing_direction]
            return

        # Walking animation
        self.cur_texture += 1
        if self.cur_texture > 7:
            self.cur_texture = 0
        self.texture = self.walk_textures[self.cur_texture][self.facing_direction]


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

    def __init__(self):
        """
        Initializer for the game
        """

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

        # Set the path to start with this program
        file_path = os.path.dirname(os.path.abspath(__file__))
        os.chdir(file_path)

        # Track the current state of what key is pressed
        self.left_pressed = False
        self.right_pressed = False
```

(continues on next page)

(continued from previous page)

```
        self.up_pressed = False
        self.down_pressed = False
        self.jump_needs_reset = False

        # Our TileMap Object
        self.tile_map = None

        # Our Scene Object
        self.scene = None

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

        # Our 'physics' engine
        self.physics_engine = None

        # A Camera that can be used for scrolling the screen
        self.camera = None

        # A Camera that can be used to draw GUI elements
        self.gui_camera = None

        self.end_of_map = 0

        # Keep track of the score
        self.score = 0

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

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

        # Set up the Cameras
        self.camera = arcade.Camera(self.width, self.height)
        self.gui_camera = arcade.Camera(self.width, self.height)

        # Map name
        map_name = ":resources:tiled_maps/map_with_ladders.json"

        # Layer Specific Options for the Tilemap
        layer_options = {
            LAYER_NAME_PLATFORMS: {
                "use_spatial_hash": True,
            },
            LAYER_NAME_MOVING_PLATFORMS: {
                "use_spatial_hash": False,
            },
            LAYER_NAME_LADDERS: {
                "use_spatial_hash": True,
```

(continues on next page)

(continued from previous page)

```
            },
            LAYER_NAME_COINS: {
                "use_spatial_hash": True,
            },
        }

        # Load in TileMap
        self.tile_map = arcade.load_tilemap(map_name, TILE_SCALING, layer_options)

        # Initiate New Scene with our TileMap, this will automatically add all layers
        # from the map as SpriteLists in the scene in the proper order.
        self.scene = arcade.Scene.from_tilemap(self.tile_map)

        # Keep track of the score
        self.score = 0

        # Set up the player, specifically placing it at these coordinates.
        self.player_sprite = PlayerCharacter()
        self.player_sprite.center_x = (
            self.tile_map.tile_width * TILE_SCALING * PLAYER_START_X
        )
        self.player_sprite.center_y = (
            self.tile_map.tile_height * TILE_SCALING * PLAYER_START_Y
        )
        self.scene.add_sprite(LAYER_NAME_PLAYER, self.player_sprite)

        # Calculate the right edge of the my_map in pixels
        self.end_of_map = self.tile_map.width * GRID_PIXEL_SIZE

        # -- Enemies
        enemies_layer = self.tile_map.object_lists[LAYER_NAME_ENEMIES]

        for my_object in enemies_layer:
            cartesian = self.tile_map.get_cartesian(
                my_object.shape[0], my_object.shape[1]
            )
            enemy_type = my_object.properties["type"]
            if enemy_type == "robot":
                enemy = RobotEnemy()
            elif enemy_type == "zombie":
                enemy = ZombieEnemy()
            enemy.center_x = math.floor(
                cartesian[0] * TILE_SCALING * self.tile_map.tile_width
            )
            enemy.center_y = math.floor(
                (cartesian[1] + 1) * (self.tile_map.tile_height * TILE_SCALING)
            )
            if "boundary_left" in my_object.properties:
                enemy.boundary_left = my_object.properties["boundary_left"]
            if "boundary_right" in my_object.properties:
                enemy.boundary_right = my_object.properties["boundary_right"]
            if "change_x" in my_object.properties:
```

(continues on next page)

(continued from previous page)

```
                enemy.change_x = my_object.properties["change_x"]
            self.scene.add_sprite(LAYER_NAME_ENEMIES, enemy)

        # --- Other stuff
        # Set the background color
        if self.tile_map.background_color:
            arcade.set_background_color(self.tile_map.background_color)

        # Create the 'physics engine'
        self.physics_engine = arcade.PhysicsEnginePlatformer(
            self.player_sprite,
            platforms=self.scene[LAYER_NAME_MOVING_PLATFORMS],
            gravity_constant=GRAVITY,
            ladders=self.scene[LAYER_NAME_LADDERS],
            walls=self.scene[LAYER_NAME_PLATFORMS]
        )

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

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

        # Activate the game camera
        self.camera.use()

        # Draw our Scene
        self.scene.draw()

        # Activate the GUI camera before drawing GUI elements
        self.gui_camera.use()

        # Draw our score on the screen, scrolling it with the viewport
        score_text = f"Score: {self.score}"
        arcade.draw_text(
            score_text,
            10,
            10,
            arcade.csscolor.BLACK,
            18,
        )

        # Draw hit boxes.
        # for wall in self.wall_list:
        #     wall.draw_hit_box(arcade.color.BLACK, 3)
        #
        # self.player_sprite.draw_hit_box(arcade.color.RED, 3)

    def process_keychange(self):
        """
        Called when we change a key up/down or we move on/off a ladder.
        """
```

(continues on next page)

(continued from previous page)

```
        # Process up/down
        if self.up_pressed and not self.down_pressed:
            if self.physics_engine.is_on_ladder():
                self.player_sprite.change_y = PLAYER_MOVEMENT_SPEED
            elif (
                self.physics_engine.can_jump(y_distance=10)
                and not self.jump_needs_reset
            ):
                self.player_sprite.change_y = PLAYER_JUMP_SPEED
                self.jump_needs_reset = True
                arcade.play_sound(self.jump_sound)
        elif self.down_pressed and not self.up_pressed:
            if self.physics_engine.is_on_ladder():
                self.player_sprite.change_y = -PLAYER_MOVEMENT_SPEED

        # Process up/down when on a ladder and no movement
        if self.physics_engine.is_on_ladder():
            if not self.up_pressed and not self.down_pressed:
                self.player_sprite.change_y = 0
            elif self.up_pressed and self.down_pressed:
                self.player_sprite.change_y = 0

        # Process left/right
        if self.right_pressed and not self.left_pressed:
            self.player_sprite.change_x = PLAYER_MOVEMENT_SPEED
        elif self.left_pressed and not self.right_pressed:
            self.player_sprite.change_x = -PLAYER_MOVEMENT_SPEED
        else:
            self.player_sprite.change_x = 0

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

        if key == arcade.key.UP or key == arcade.key.W:
            self.up_pressed = True
        elif key == arcade.key.DOWN or key == arcade.key.S:
            self.down_pressed = True
        elif key == arcade.key.LEFT or key == arcade.key.A:
            self.left_pressed = True
        elif key == arcade.key.RIGHT or key == arcade.key.D:
            self.right_pressed = True

        self.process_keychange()

    def on_key_release(self, key, modifiers):
        """Called when the user releases a key."""

        if key == arcade.key.UP or key == arcade.key.W:
            self.up_pressed = False
            self.jump_needs_reset = False
        elif key == arcade.key.DOWN or key == arcade.key.S:
            self.down_pressed = False
```

(continues on next page)

(continued from previous page)

```python
        elif key == arcade.key.LEFT or key == arcade.key.A:
            self.left_pressed = False
        elif key == arcade.key.RIGHT or key == arcade.key.D:
            self.right_pressed = False

        self.process_keychange()

    def center_camera_to_player(self):
        screen_center_x = self.player_sprite.center_x - (self.camera.viewport_width / 2)
        screen_center_y = self.player_sprite.center_y - (
            self.camera.viewport_height / 2
        )
        if screen_center_x < 0:
            screen_center_x = 0
        if screen_center_y < 0:
            screen_center_y = 0
        player_centered = screen_center_x, screen_center_y

        self.camera.move_to(player_centered, 0.2)

    def on_update(self, delta_time):
        """Movement and game logic"""

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

        # Update animations
        if self.physics_engine.can_jump():
            self.player_sprite.can_jump = False
        else:
            self.player_sprite.can_jump = True

        if self.physics_engine.is_on_ladder() and not self.physics_engine.can_jump():
            self.player_sprite.is_on_ladder = True
            self.process_keychange()
        else:
            self.player_sprite.is_on_ladder = False
            self.process_keychange()

        # Update Animations
        self.scene.update_animation(
            delta_time,
            [
                LAYER_NAME_COINS,
                LAYER_NAME_BACKGROUND,
                LAYER_NAME_PLAYER,
                LAYER_NAME_ENEMIES,
            ],
        )

        # Update moving platforms and enemies
        self.scene.update([LAYER_NAME_MOVING_PLATFORMS, LAYER_NAME_ENEMIES])
```

(continues on next page)

(continued from previous page)

```python

        # See if the enemy hit a boundary and needs to reverse direction.
        for enemy in self.scene[LAYER_NAME_ENEMIES]:
            if (
                enemy.boundary_right
                and enemy.right > enemy.boundary_right
                and enemy.change_x > 0
            ):
                enemy.change_x *= -1

            if (
                enemy.boundary_left
                and enemy.left < enemy.boundary_left
                and enemy.change_x < 0
            ):
                enemy.change_x *= -1

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

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

            # Figure out how many points this coin is worth
            if "Points" not in coin.properties:
                print("Warning, collected a coin without a Points property.")
            else:
                points = int(coin.properties["Points"])
                self.score += points

            # Remove the coin
            coin.remove_from_sprite_lists()
            arcade.play_sound(self.collect_coin_sound)

        # Position the camera
        self.center_camera_to_player()


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


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

8.15 Step 15 - Collision with Enemies

Listing 54: Collision with Enemies

```
"""
Platformer Game

python -m arcade.examples.platform_tutorial.11_animate_character
"""
import math
import os

import arcade

# Constants
SCREEN_WIDTH = 1000
SCREEN_HEIGHT = 650
SCREEN_TITLE = "Platformer"

# Constants used to scale our sprites from their original size
TILE_SCALING = 0.5
CHARACTER_SCALING = TILE_SCALING * 2
COIN_SCALING = TILE_SCALING
SPRITE_PIXEL_SIZE = 128
GRID_PIXEL_SIZE = SPRITE_PIXEL_SIZE * TILE_SCALING

# Movement speed of player, in pixels per frame
PLAYER_MOVEMENT_SPEED = 7
GRAVITY = 1.5
PLAYER_JUMP_SPEED = 30

# How many pixels to keep as a minimum margin between the character
# and the edge of the screen.
LEFT_VIEWPORT_MARGIN = 200
RIGHT_VIEWPORT_MARGIN = 200
BOTTOM_VIEWPORT_MARGIN = 150
TOP_VIEWPORT_MARGIN = 100

PLAYER_START_X = 2
PLAYER_START_Y = 1

# Constants used to track if the player is facing left or right
RIGHT_FACING = 0
LEFT_FACING = 1

LAYER_NAME_MOVING_PLATFORMS = "Moving Platforms"
LAYER_NAME_PLATFORMS = "Platforms"
LAYER_NAME_COINS = "Coins"
LAYER_NAME_BACKGROUND = "Background"
LAYER_NAME_LADDERS = "Ladders"
LAYER_NAME_PLAYER = "Player"
LAYER_NAME_ENEMIES = "Enemies"
```

(continues on next page)

(continued from previous page)

```python


def load_texture_pair(filename):
    """
    Load a texture pair, with the second being a mirror image.
    """
    return [
        arcade.load_texture(filename),
        arcade.load_texture(filename, flipped_horizontally=True),
    ]


class Entity(arcade.Sprite):
    def __init__(self, name_folder, name_file):
        super().__init__()

        # Default to facing right
        self.facing_direction = RIGHT_FACING

        # Used for image sequences
        self.cur_texture = 0
        self.scale = CHARACTER_SCALING

        main_path = f":resources:images/animated_characters/{name_folder}/{name_file}"

        self.idle_texture_pair = load_texture_pair(f"{main_path}_idle.png")
        self.jump_texture_pair = load_texture_pair(f"{main_path}_jump.png")
        self.fall_texture_pair = load_texture_pair(f"{main_path}_fall.png")

        # Load textures for walking
        self.walk_textures = []
        for i in range(8):
            texture = 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]

        # Hit box will be set based on the first image used. If you want to specify
        # a different hit box, you can do it like the code below.
        # self.set_hit_box([[-22, -64], [22, -64], [22, 28], [-22, 28]])
        self.set_hit_box(self.texture.hit_box_points)


class Enemy(Entity):
```

(continues on next page)

(continued from previous page)

```python
    def __init__(self, name_folder, name_file):

        # Setup parent class
        super().__init__(name_folder, name_file)

        self.should_update_walk = 0

    def update_animation(self, delta_time: float = 1 / 60):

        # Figure out if we need to flip face left or right
        if self.change_x < 0 and self.facing_direction == RIGHT_FACING:
            self.facing_direction = LEFT_FACING
        elif self.change_x > 0 and self.facing_direction == LEFT_FACING:
            self.facing_direction = RIGHT_FACING

        # Idle animation
        if self.change_x == 0:
            self.texture = self.idle_texture_pair[self.facing_direction]
            return

        # Walking animation
        if self.should_update_walk == 3:
            self.cur_texture += 1
            if self.cur_texture > 7:
                self.cur_texture = 0
            self.texture = self.walk_textures[self.cur_texture][self.facing_direction]
            self.should_update_walk = 0
            return

        self.should_update_walk += 1


class RobotEnemy(Enemy):
    def __init__(self):

        # Set up parent class
        super().__init__("robot", "robot")


class ZombieEnemy(Enemy):
    def __init__(self):

        # Set up parent class
        super().__init__("zombie", "zombie")


class PlayerCharacter(Entity):
    """Player Sprite"""

    def __init__(self):

        # Set up parent class
```

(continues on next page)

(continued from previous page)

```
        super().__init__("male_person", "malePerson")

        # Track our state
        self.jumping = False
        self.climbing = False
        self.is_on_ladder = False

    def update_animation(self, delta_time: float = 1 / 60):

        # Figure out if we need to flip face left or right
        if self.change_x < 0 and self.facing_direction == RIGHT_FACING:
            self.facing_direction = LEFT_FACING
        elif self.change_x > 0 and self.facing_direction == LEFT_FACING:
            self.facing_direction = RIGHT_FACING

        # Climbing animation
        if self.is_on_ladder:
            self.climbing = True
        if not self.is_on_ladder and self.climbing:
            self.climbing = False
        if self.climbing and abs(self.change_y) > 1:
            self.cur_texture += 1
            if self.cur_texture > 7:
                self.cur_texture = 0
        if self.climbing:
            self.texture = self.climbing_textures[self.cur_texture // 4]
            return

        # Jumping animation
        if self.change_y > 0 and not self.is_on_ladder:
            self.texture = self.jump_texture_pair[self.facing_direction]
            return
        elif self.change_y < 0 and not self.is_on_ladder:
            self.texture = self.fall_texture_pair[self.facing_direction]
            return

        # Idle animation
        if self.change_x == 0:
            self.texture = self.idle_texture_pair[self.facing_direction]
            return

        # Walking animation
        self.cur_texture += 1
        if self.cur_texture > 7:
            self.cur_texture = 0
        self.texture = self.walk_textures[self.cur_texture][self.facing_direction]


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

(continues on next page)

(continued from previous page)

```
    def __init__(self):
        """
        Initializer for the game
        """

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

        # Set the path to start with this program
        file_path = os.path.dirname(os.path.abspath(__file__))
        os.chdir(file_path)

        # Track the current state of what key is pressed
        self.left_pressed = False
        self.right_pressed = False
        self.up_pressed = False
        self.down_pressed = False
        self.jump_needs_reset = False

        # Our TileMap Object
        self.tile_map = None

        # Our Scene Object
        self.scene = None

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

        # Our 'physics' engine
        self.physics_engine = None

        # A Camera that can be used for scrolling the screen
        self.camera = None

        # A Camera that can be used to draw GUI elements
        self.gui_camera = None

        self.end_of_map = 0

        # Keep track of the score
        self.score = 0

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

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

        # Set up the Cameras
```

(continues on next page)

(continued from previous page)

```python
        self.camera = arcade.Camera(self.width, self.height)
        self.gui_camera = arcade.Camera(self.width, self.height)

        # Map name
        map_name = ":resources:tiled_maps/map_with_ladders.json"

        # Layer Specific Options for the Tilemap
        layer_options = {
            LAYER_NAME_PLATFORMS: {
                "use_spatial_hash": True,
            },
            LAYER_NAME_MOVING_PLATFORMS: {
                "use_spatial_hash": False,
            },
            LAYER_NAME_LADDERS: {
                "use_spatial_hash": True,
            },
            LAYER_NAME_COINS: {
                "use_spatial_hash": True,
            },
        }

        # Load in TileMap
        self.tile_map = arcade.load_tilemap(map_name, TILE_SCALING, layer_options)

        # Initiate New Scene with our TileMap, this will automatically add all layers
        # from the map as SpriteLists in the scene in the proper order.
        self.scene = arcade.Scene.from_tilemap(self.tile_map)

        # Keep track of the score
        self.score = 0

        # Set up the player, specifically placing it at these coordinates.
        self.player_sprite = PlayerCharacter()
        self.player_sprite.center_x = (
            self.tile_map.tile_width * TILE_SCALING * PLAYER_START_X
        )
        self.player_sprite.center_y = (
            self.tile_map.tile_height * TILE_SCALING * PLAYER_START_Y
        )
        self.scene.add_sprite(LAYER_NAME_PLAYER, self.player_sprite)

        # Calculate the right edge of the my_map in pixels
        self.end_of_map = self.tile_map.width * GRID_PIXEL_SIZE

        # -- Enemies
        enemies_layer = self.tile_map.object_lists[LAYER_NAME_ENEMIES]

        for my_object in enemies_layer:
            cartesian = self.tile_map.get_cartesian(
                my_object.shape[0], my_object.shape[1]
            )
```

(continues on next page)

(continued from previous page)

```
            enemy_type = my_object.properties["type"]
            if enemy_type == "robot":
                enemy = RobotEnemy()
            elif enemy_type == "zombie":
                enemy = ZombieEnemy()
            enemy.center_x = math.floor(
                cartesian[0] * TILE_SCALING * self.tile_map.tile_width
            )
            enemy.center_y = math.floor(
                (cartesian[1] + 1) * (self.tile_map.tile_height * TILE_SCALING)
            )
            if "boundary_left" in my_object.properties:
                enemy.boundary_left = my_object.properties["boundary_left"]
            if "boundary_right" in my_object.properties:
                enemy.boundary_right = my_object.properties["boundary_right"]
            if "change_x" in my_object.properties:
                enemy.change_x = my_object.properties["change_x"]
            self.scene.add_sprite(LAYER_NAME_ENEMIES, enemy)

        # --- Other stuff
        # Set the background color
        if self.tile_map.background_color:
            arcade.set_background_color(self.tile_map.background_color)

        # Create the 'physics engine'
        self.physics_engine = arcade.PhysicsEnginePlatformer(
            self.player_sprite,
            platforms=self.scene[LAYER_NAME_MOVING_PLATFORMS],
            gravity_constant=GRAVITY,
            ladders=self.scene[LAYER_NAME_LADDERS],
            walls=self.scene[LAYER_NAME_PLATFORMS]
        )

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

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

        # Activate the game camera
        self.camera.use()

        # Draw our Scene
        self.scene.draw()

        # Activate the GUI camera before drawing GUI elements
        self.gui_camera.use()

        # Draw our score on the screen, scrolling it with the viewport
        score_text = f"Score: {self.score}"
        arcade.draw_text(
            score_text,
```

(continues on next page)

(continued from previous page)

```
            10,
            10,
            arcade.csscolor.BLACK,
            18,
        )

        # Draw hit boxes.
        # for wall in self.wall_list:
        #     wall.draw_hit_box(arcade.color.BLACK, 3)
        #
        # self.player_sprite.draw_hit_box(arcade.color.RED, 3)

    def process_keychange(self):
        """
        Called when we change a key up/down, or we move on/off a ladder.
        """
        # Process up/down
        if self.up_pressed and not self.down_pressed:
            if self.physics_engine.is_on_ladder():
                self.player_sprite.change_y = PLAYER_MOVEMENT_SPEED
            elif (
                self.physics_engine.can_jump(y_distance=10)
                and not self.jump_needs_reset
            ):
                self.player_sprite.change_y = PLAYER_JUMP_SPEED
                self.jump_needs_reset = True
                arcade.play_sound(self.jump_sound)
        elif self.down_pressed and not self.up_pressed:
            if self.physics_engine.is_on_ladder():
                self.player_sprite.change_y = -PLAYER_MOVEMENT_SPEED

        # Process up/down when on a ladder and no movement
        if self.physics_engine.is_on_ladder():
            if not self.up_pressed and not self.down_pressed:
                self.player_sprite.change_y = 0
            elif self.up_pressed and self.down_pressed:
                self.player_sprite.change_y = 0

        # Process left/right
        if self.right_pressed and not self.left_pressed:
            self.player_sprite.change_x = PLAYER_MOVEMENT_SPEED
        elif self.left_pressed and not self.right_pressed:
            self.player_sprite.change_x = -PLAYER_MOVEMENT_SPEED
        else:
            self.player_sprite.change_x = 0

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

        if key == arcade.key.UP or key == arcade.key.W:
            self.up_pressed = True
        elif key == arcade.key.DOWN or key == arcade.key.S:
```

(continues on next page)

(continued from previous page)

```
            self.down_pressed = True
        elif key == arcade.key.LEFT or key == arcade.key.A:
            self.left_pressed = True
        elif key == arcade.key.RIGHT or key == arcade.key.D:
            self.right_pressed = True

        self.process_keychange()

    def on_key_release(self, key, modifiers):
        """Called when the user releases a key."""

        if key == arcade.key.UP or key == arcade.key.W:
            self.up_pressed = False
            self.jump_needs_reset = False
        elif key == arcade.key.DOWN or key == arcade.key.S:
            self.down_pressed = False
        elif key == arcade.key.LEFT or key == arcade.key.A:
            self.left_pressed = False
        elif key == arcade.key.RIGHT or key == arcade.key.D:
            self.right_pressed = False

        self.process_keychange()

    def center_camera_to_player(self, speed=0.2):
        screen_center_x = self.player_sprite.center_x - (self.camera.viewport_width / 2)
        screen_center_y = self.player_sprite.center_y - (
            self.camera.viewport_height / 2
        )
        if screen_center_x < 0:
            screen_center_x = 0
        if screen_center_y < 0:
            screen_center_y = 0
        player_centered = screen_center_x, screen_center_y

        self.camera.move_to(player_centered, speed)

    def on_update(self, delta_time):
        """Movement and game logic"""

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

        # Update animations
        if self.physics_engine.can_jump():
            self.player_sprite.can_jump = False
        else:
            self.player_sprite.can_jump = True

        if self.physics_engine.is_on_ladder() and not self.physics_engine.can_jump():
            self.player_sprite.is_on_ladder = True
            self.process_keychange()
        else:
```

(continues on next page)

(continued from previous page)

```
            self.player_sprite.is_on_ladder = False
            self.process_keychange()

        # Update Animations
        self.scene.update_animation(
            delta_time,
            [
                LAYER_NAME_COINS,
                LAYER_NAME_BACKGROUND,
                LAYER_NAME_PLAYER,
                LAYER_NAME_ENEMIES,
            ],
        )

        # Update moving platforms and enemies
        self.scene.update([LAYER_NAME_MOVING_PLATFORMS, LAYER_NAME_ENEMIES])

        # See if the enemy hit a boundary and needs to reverse direction.
        for enemy in self.scene[LAYER_NAME_ENEMIES]:
            if (
                enemy.boundary_right
                and enemy.right > enemy.boundary_right
                and enemy.change_x > 0
            ):
                enemy.change_x *= -1

            if (
                enemy.boundary_left
                and enemy.left < enemy.boundary_left
                and enemy.change_x < 0
            ):
                enemy.change_x *= -1

        player_collision_list = arcade.check_for_collision_with_lists(
            self.player_sprite,
            [
                self.scene[LAYER_NAME_COINS],
                self.scene[LAYER_NAME_ENEMIES],
            ],
        )

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

        # Loop through each coin we hit (if any) and remove it
        for collision in player_collision_list:

            if self.scene[LAYER_NAME_ENEMIES] in collision.sprite_lists:
                arcade.play_sound(self.game_over)
                self.setup()
```

(continues on next page)

(continued from previous page)

```
                return
            else:
                # Figure out how many points this coin is worth
                if "Points" not in collision.properties:
                    print("Warning, collected a coin without a Points property.")
                else:
                    points = int(collision.properties["Points"])
                    self.score += points

                # Remove the coin
                collision.remove_from_sprite_lists()
                arcade.play_sound(self.collect_coin_sound)

        # Position the camera
        self.center_camera_to_player()


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


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

8.16 Step 16 - Shooting Bullets

Listing 55: Shooting Bullets

```
"""
Platformer Game

python -m arcade.examples.platform_tutorial.11_animate_character
"""
import math
import os

import arcade

# Constants
SCREEN_WIDTH = 1000
SCREEN_HEIGHT = 650
SCREEN_TITLE = "Platformer"

# Constants used to scale our sprites from their original size
TILE_SCALING = 0.5
CHARACTER_SCALING = TILE_SCALING * 2
COIN_SCALING = TILE_SCALING
```

(continues on next page)

(continued from previous page)

```
SPRITE_PIXEL_SIZE = 128
GRID_PIXEL_SIZE = SPRITE_PIXEL_SIZE * TILE_SCALING

# Shooting Constants
SPRITE_SCALING_LASER = 0.8
SHOOT_SPEED = 15
BULLET_SPEED = 12
BULLET_DAMAGE = 25

# Movement speed of player, in pixels per frame
PLAYER_MOVEMENT_SPEED = 7
GRAVITY = 1.5
PLAYER_JUMP_SPEED = 30

# How many pixels to keep as a minimum margin between the character
# and the edge of the screen.
LEFT_VIEWPORT_MARGIN = 200
RIGHT_VIEWPORT_MARGIN = 200
BOTTOM_VIEWPORT_MARGIN = 150
TOP_VIEWPORT_MARGIN = 100

PLAYER_START_X = 2
PLAYER_START_Y = 1

# Constants used to track if the player is facing left or right
RIGHT_FACING = 0
LEFT_FACING = 1

LAYER_NAME_MOVING_PLATFORMS = "Moving Platforms"
LAYER_NAME_PLATFORMS = "Platforms"
LAYER_NAME_COINS = "Coins"
LAYER_NAME_BACKGROUND = "Background"
LAYER_NAME_LADDERS = "Ladders"
LAYER_NAME_PLAYER = "Player"
LAYER_NAME_ENEMIES = "Enemies"
LAYER_NAME_BULLETS = "Bullets"


def load_texture_pair(filename):
    """
    Load a texture pair, with the second being a mirror image.
    """
    return [
        arcade.load_texture(filename),
        arcade.load_texture(filename, flipped_horizontally=True),
    ]


class Entity(arcade.Sprite):
    def __init__(self, name_folder, name_file):
        super().__init__()

```

(continues on next page)

(continued from previous page)

```
        # Default to facing right
        self.facing_direction = RIGHT_FACING

        # Used for image sequences
        self.cur_texture = 0
        self.scale = CHARACTER_SCALING

        main_path = f":resources:images/animated_characters/{name_folder}/{name_file}"

        self.idle_texture_pair = load_texture_pair(f"{main_path}_idle.png")
        self.jump_texture_pair = load_texture_pair(f"{main_path}_jump.png")
        self.fall_texture_pair = load_texture_pair(f"{main_path}_fall.png")

        # Load textures for walking
        self.walk_textures = []
        for i in range(8):
            texture = 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]

        # Hit box will be set based on the first image used. If you want to specify
        # a different hit box, you can do it like the code below.
        # self.set_hit_box([[-22, -64], [22, -64], [22, 28], [-22, 28]])
        self.set_hit_box(self.texture.hit_box_points)


class Enemy(Entity):
    def __init__(self, name_folder, name_file):

        # Setup parent class
        super().__init__(name_folder, name_file)

        self.should_update_walk = 0
        self.health = 0

    def update_animation(self, delta_time: float = 1 / 60):

        # Figure out if we need to flip face left or right
        if self.change_x < 0 and self.facing_direction == RIGHT_FACING:
            self.facing_direction = LEFT_FACING
        elif self.change_x > 0 and self.facing_direction == LEFT_FACING:
            self.facing_direction = RIGHT_FACING

```

(continues on next page)

(continued from previous page)

```python
        # Idle animation
        if self.change_x == 0:
            self.texture = self.idle_texture_pair[self.facing_direction]
            return

        # Walking animation
        if self.should_update_walk == 3:
            self.cur_texture += 1
            if self.cur_texture > 7:
                self.cur_texture = 0
            self.texture = self.walk_textures[self.cur_texture][self.facing_direction]
            self.should_update_walk = 0
            return

        self.should_update_walk += 1


class RobotEnemy(Enemy):
    def __init__(self):

        # Set up parent class
        super().__init__("robot", "robot")

        self.health = 100


class ZombieEnemy(Enemy):
    def __init__(self):

        # Set up parent class
        super().__init__("zombie", "zombie")

        self.health = 50


class PlayerCharacter(Entity):
    """Player Sprite"""

    def __init__(self):

        # Set up parent class
        super().__init__("male_person", "malePerson")

        # Track our state
        self.jumping = False
        self.climbing = False
        self.is_on_ladder = False

    def update_animation(self, delta_time: float = 1 / 60):

        # Figure out if we need to flip face left or right
        if self.change_x < 0 and self.facing_direction == RIGHT_FACING:
```

(continues on next page)

(continued from previous page)

```
            self.facing_direction = LEFT_FACING
        elif self.change_x > 0 and self.facing_direction == LEFT_FACING:
            self.facing_direction = RIGHT_FACING

        # Climbing animation
        if self.is_on_ladder:
            self.climbing = True
        if not self.is_on_ladder and self.climbing:
            self.climbing = False
        if self.climbing and abs(self.change_y) > 1:
            self.cur_texture += 1
            if self.cur_texture > 7:
                self.cur_texture = 0
        if self.climbing:
            self.texture = self.climbing_textures[self.cur_texture // 4]
            return

        # Jumping animation
        if self.change_y > 0 and not self.is_on_ladder:
            self.texture = self.jump_texture_pair[self.facing_direction]
            return
        elif self.change_y < 0 and not self.is_on_ladder:
            self.texture = self.fall_texture_pair[self.facing_direction]
            return

        # Idle animation
        if self.change_x == 0:
            self.texture = self.idle_texture_pair[self.facing_direction]
            return

        # Walking animation
        self.cur_texture += 1
        if self.cur_texture > 7:
            self.cur_texture = 0
        self.texture = self.walk_textures[self.cur_texture][self.facing_direction]


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

    def __init__(self):
        """
        Initializer for the game
        """

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

        # Set the path to start with this program
        file_path = os.path.dirname(os.path.abspath(__file__))
```

(continues on next page)

(continued from previous page)

```
        os.chdir(file_path)

        # Track the current state of what key is pressed
        self.left_pressed = False
        self.right_pressed = False
        self.up_pressed = False
        self.down_pressed = False
        self.shoot_pressed = False
        self.jump_needs_reset = False

        # Our TileMap Object
        self.tile_map = None

        # Our Scene Object
        self.scene = None

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

        # Our 'physics' engine
        self.physics_engine = None

        # A Camera that can be used for scrolling the screen
        self.camera = None

        # A Camera that can be used to draw GUI elements
        self.gui_camera = None

        self.end_of_map = 0

        # Keep track of the score
        self.score = 0

        # Shooting mechanics
        self.can_shoot = False
        self.shoot_timer = 0

        # Load sounds
        self.collect_coin_sound = arcade.load_sound(":resources:sounds/coin1.wav")
        self.jump_sound = arcade.load_sound(":resources:sounds/jump1.wav")
        self.game_over = arcade.load_sound(":resources:sounds/gameover1.wav")
        self.shoot_sound = arcade.load_sound(":resources:sounds/hurt5.wav")
        self.hit_sound = arcade.load_sound(":resources:sounds/hit5.wav")

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

        # Setup the Cameras
        self.camera = arcade.Camera(self.width, self.height)
        self.gui_camera = arcade.Camera(self.width, self.height)

        # Map name
```

(continues on next page)

(continued from previous page)

```
        map_name = ":resources:tiled_maps/map_with_ladders.json"

        # Layer Specific Options for the Tilemap
        layer_options = {
            LAYER_NAME_PLATFORMS: {
                "use_spatial_hash": True,
            },
            LAYER_NAME_MOVING_PLATFORMS: {
                "use_spatial_hash": False,
            },
            LAYER_NAME_LADDERS: {
                "use_spatial_hash": True,
            },
            LAYER_NAME_COINS: {
                "use_spatial_hash": True,
            },
        }

        # Load in TileMap
        self.tile_map = arcade.load_tilemap(map_name, TILE_SCALING, layer_options)

        # Initiate New Scene with our TileMap, this will automatically add all layers
        # from the map as SpriteLists in the scene in the proper order.
        self.scene = arcade.Scene.from_tilemap(self.tile_map)

        # Keep track of the score
        self.score = 0

        # Shooting mechanics
        self.can_shoot = True
        self.shoot_timer = 0

        # Set up the player, specifically placing it at these coordinates.
        self.player_sprite = PlayerCharacter()
        self.player_sprite.center_x = (
            self.tile_map.tile_width * TILE_SCALING * PLAYER_START_X
        )
        self.player_sprite.center_y = (
            self.tile_map.tile_height * TILE_SCALING * PLAYER_START_Y
        )
        self.scene.add_sprite(LAYER_NAME_PLAYER, self.player_sprite)

        # Calculate the right edge of the my_map in pixels
        self.end_of_map = self.tile_map.width * GRID_PIXEL_SIZE

        # -- Enemies
        enemies_layer = self.tile_map.object_lists[LAYER_NAME_ENEMIES]

        for my_object in enemies_layer:
            cartesian = self.tile_map.get_cartesian(
                my_object.shape[0], my_object.shape[1]
            )
```

(continues on next page)

(continued from previous page)

```python
            enemy_type = my_object.properties["type"]
            if enemy_type == "robot":
                enemy = RobotEnemy()
            elif enemy_type == "zombie":
                enemy = ZombieEnemy()
            enemy.center_x = math.floor(
                cartesian[0] * TILE_SCALING * self.tile_map.tile_width
            )
            enemy.center_y = math.floor(
                (cartesian[1] + 1) * (self.tile_map.tile_height * TILE_SCALING)
            )
            if "boundary_left" in my_object.properties:
                enemy.boundary_left = my_object.properties["boundary_left"]
            if "boundary_right" in my_object.properties:
                enemy.boundary_right = my_object.properties["boundary_right"]
            if "change_x" in my_object.properties:
                enemy.change_x = my_object.properties["change_x"]
            self.scene.add_sprite(LAYER_NAME_ENEMIES, enemy)

        # Add bullet spritelist to Scene
        self.scene.add_sprite_list(LAYER_NAME_BULLETS)

        # --- Other stuff
        # Set the background color
        if self.tile_map.background_color:
            arcade.set_background_color(self.tile_map.background_color)

        # Create the 'physics engine'
        self.physics_engine = arcade.PhysicsEnginePlatformer(
            self.player_sprite,
            platforms=self.scene[LAYER_NAME_MOVING_PLATFORMS],
            gravity_constant=GRAVITY,
            ladders=self.scene[LAYER_NAME_LADDERS],
            walls=self.scene[LAYER_NAME_PLATFORMS]
        )

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

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

        # Activate the game camera
        self.camera.use()

        # Draw our Scene
        self.scene.draw()

        # Activate the GUI camera before drawing GUI elements
        self.gui_camera.use()

        # Draw our score on the screen, scrolling it with the viewport
```

(continues on next page)

(continued from previous page)

```
        score_text = f"Score: {self.score}"
        arcade.draw_text(
            score_text,
            10,
            10,
            arcade.csscolor.BLACK,
            18,
        )

        # Draw hit boxes.
        # for wall in self.wall_list:
        #     wall.draw_hit_box(arcade.color.BLACK, 3)
        #
        # self.player_sprite.draw_hit_box(arcade.color.RED, 3)

    def process_keychange(self):
        """
        Called when we change a key up/down, or we move on/off a ladder.
        """
        # Process up/down
        if self.up_pressed and not self.down_pressed:
            if self.physics_engine.is_on_ladder():
                self.player_sprite.change_y = PLAYER_MOVEMENT_SPEED
            elif (
                self.physics_engine.can_jump(y_distance=10)
                and not self.jump_needs_reset
            ):
                self.player_sprite.change_y = PLAYER_JUMP_SPEED
                self.jump_needs_reset = True
                arcade.play_sound(self.jump_sound)
        elif self.down_pressed and not self.up_pressed:
            if self.physics_engine.is_on_ladder():
                self.player_sprite.change_y = -PLAYER_MOVEMENT_SPEED

        # Process up/down when on a ladder and no movement
        if self.physics_engine.is_on_ladder():
            if not self.up_pressed and not self.down_pressed:
                self.player_sprite.change_y = 0
            elif self.up_pressed and self.down_pressed:
                self.player_sprite.change_y = 0

        # Process left/right
        if self.right_pressed and not self.left_pressed:
            self.player_sprite.change_x = PLAYER_MOVEMENT_SPEED
        elif self.left_pressed and not self.right_pressed:
            self.player_sprite.change_x = -PLAYER_MOVEMENT_SPEED
        else:
            self.player_sprite.change_x = 0

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

```

(continues on next page)

(continued from previous page)

```
        if key == arcade.key.UP or key == arcade.key.W:
            self.up_pressed = True
        elif key == arcade.key.DOWN or key == arcade.key.S:
            self.down_pressed = True
        elif key == arcade.key.LEFT or key == arcade.key.A:
            self.left_pressed = True
        elif key == arcade.key.RIGHT or key == arcade.key.D:
            self.right_pressed = True

        if key == arcade.key.Q:
            self.shoot_pressed = True

        self.process_keychange()

    def on_key_release(self, key, modifiers):
        """Called when the user releases a key."""

        if key == arcade.key.UP or key == arcade.key.W:
            self.up_pressed = False
            self.jump_needs_reset = False
        elif key == arcade.key.DOWN or key == arcade.key.S:
            self.down_pressed = False
        elif key == arcade.key.LEFT or key == arcade.key.A:
            self.left_pressed = False
        elif key == arcade.key.RIGHT or key == arcade.key.D:
            self.right_pressed = False

        if key == arcade.key.Q:
            self.shoot_pressed = False

        self.process_keychange()

    def center_camera_to_player(self, speed=0.2):
        screen_center_x = self.player_sprite.center_x - (self.camera.viewport_width / 2)
        screen_center_y = self.player_sprite.center_y - (
            self.camera.viewport_height / 2
        )
        if screen_center_x < 0:
            screen_center_x = 0
        if screen_center_y < 0:
            screen_center_y = 0
        player_centered = screen_center_x, screen_center_y

        self.camera.move_to(player_centered, speed)

    def on_update(self, delta_time):
        """Movement and game logic"""

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

        # Update animations
```

(continues on next page)

(continued from previous page)

```
        if self.physics_engine.can_jump():
            self.player_sprite.can_jump = False
        else:
            self.player_sprite.can_jump = True

        if self.physics_engine.is_on_ladder() and not self.physics_engine.can_jump():
            self.player_sprite.is_on_ladder = True
            self.process_keychange()
        else:
            self.player_sprite.is_on_ladder = False
            self.process_keychange()

        if self.can_shoot:
            if self.shoot_pressed:
                arcade.play_sound(self.shoot_sound)
                bullet = arcade.Sprite(
                    ":resources:images/space_shooter/laserBlue01.png",
                    SPRITE_SCALING_LASER,
                )

                if self.player_sprite.facing_direction == RIGHT_FACING:
                    bullet.change_x = BULLET_SPEED
                else:
                    bullet.change_x = -BULLET_SPEED

                bullet.center_x = self.player_sprite.center_x
                bullet.center_y = self.player_sprite.center_y

                self.scene.add_sprite(LAYER_NAME_BULLETS, bullet)

                self.can_shoot = False
        else:
            self.shoot_timer += 1
            if self.shoot_timer == SHOOT_SPEED:
                self.can_shoot = True
                self.shoot_timer = 0

        # Update Animations
        self.scene.update_animation(
            delta_time,
            [
                LAYER_NAME_COINS,
                LAYER_NAME_BACKGROUND,
                LAYER_NAME_PLAYER,
                LAYER_NAME_ENEMIES,
            ],
        )

        # Update moving platforms, enemies, and bullets
        self.scene.update(
            [LAYER_NAME_MOVING_PLATFORMS, LAYER_NAME_ENEMIES, LAYER_NAME_BULLETS]
        )
```

(continues on next page)

(continued from previous page)

```python

        # See if the enemy hit a boundary and needs to reverse direction.
        for enemy in self.scene[LAYER_NAME_ENEMIES]:
            if (
                enemy.boundary_right
                and enemy.right > enemy.boundary_right
                and enemy.change_x > 0
            ):
                enemy.change_x *= -1

            if (
                enemy.boundary_left
                and enemy.left < enemy.boundary_left
                and enemy.change_x < 0
            ):
                enemy.change_x *= -1

        for bullet in self.scene[LAYER_NAME_BULLETS]:
            hit_list = arcade.check_for_collision_with_lists(
                bullet,
                [
                    self.scene[LAYER_NAME_ENEMIES],
                    self.scene[LAYER_NAME_PLATFORMS],
                    self.scene[LAYER_NAME_MOVING_PLATFORMS],
                ],
            )

            if hit_list:
                bullet.remove_from_sprite_lists()

                for collision in hit_list:
                    if (
                        self.scene[LAYER_NAME_ENEMIES]
                        in collision.sprite_lists
                    ):
                        # The collision was with an enemy
                        collision.health -= BULLET_DAMAGE

                        if collision.health <= 0:
                            collision.remove_from_sprite_lists()
                            self.score += 100

                        # Hit sound
                        arcade.play_sound(self.hit_sound)

                return

            if (bullet.right < 0) or (
                bullet.left
                > (self.tile_map.width * self.tile_map.tile_width) * TILE_SCALING
            ):
                bullet.remove_from_sprite_lists()
```

(continues on next page)

(continued from previous page)

```python

        player_collision_list = arcade.check_for_collision_with_lists(
            self.player_sprite,
            [
                self.scene[LAYER_NAME_COINS],
                self.scene[LAYER_NAME_ENEMIES],
            ],
        )

        # Loop through each coin we hit (if any) and remove it
        for collision in player_collision_list:

            if self.scene[LAYER_NAME_ENEMIES] in collision.sprite_lists:
                arcade.play_sound(self.game_over)
                self.setup()
                return
            else:
                # Figure out how many points this coin is worth
                if "Points" not in collision.properties:
                    print("Warning, collected a coin without a Points property.")
                else:
                    points = int(collision.properties["Points"])
                    self.score += points

                # Remove the coin
                collision.remove_from_sprite_lists()
                arcade.play_sound(self.collect_coin_sound)

        # Position the camera
        self.center_camera_to_player()


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


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

8.17 Step 17 - Views

Listing 56: Shooting Bullets

```python
"""
Platformer Game

python -m arcade.examples.platform_tutorial.11_animate_character
"""
import math
import os

import arcade

# Constants
SCREEN_WIDTH = 1000
SCREEN_HEIGHT = 650
SCREEN_TITLE = "Platformer"

# Constants used to scale our sprites from their original size
TILE_SCALING = 0.5
CHARACTER_SCALING = TILE_SCALING * 2
COIN_SCALING = TILE_SCALING
SPRITE_PIXEL_SIZE = 128
GRID_PIXEL_SIZE = SPRITE_PIXEL_SIZE * TILE_SCALING

# Shooting Constants
SPRITE_SCALING_LASER = 0.8
SHOOT_SPEED = 15
BULLET_SPEED = 12
BULLET_DAMAGE = 25

# Movement speed of player, in pixels per frame
PLAYER_MOVEMENT_SPEED = 7
GRAVITY = 1.5
PLAYER_JUMP_SPEED = 30

# How many pixels to keep as a minimum margin between the character
# and the edge of the screen.
LEFT_VIEWPORT_MARGIN = 200
RIGHT_VIEWPORT_MARGIN = 200
BOTTOM_VIEWPORT_MARGIN = 150
TOP_VIEWPORT_MARGIN = 100

PLAYER_START_X = 2
PLAYER_START_Y = 1

# Constants used to track if the player is facing left or right
RIGHT_FACING = 0
LEFT_FACING = 1

LAYER_NAME_MOVING_PLATFORMS = "Moving Platforms"
```

(continues on next page)

(continued from previous page)

```
LAYER_NAME_PLATFORMS = "Platforms"
LAYER_NAME_COINS = "Coins"
LAYER_NAME_BACKGROUND = "Background"
LAYER_NAME_LADDERS = "Ladders"
LAYER_NAME_PLAYER = "Player"
LAYER_NAME_ENEMIES = "Enemies"
LAYER_NAME_BULLETS = "Bullets"


def load_texture_pair(filename):
    """
    Load a texture pair, with the second being a mirror image.
    """
    return [
        arcade.load_texture(filename),
        arcade.load_texture(filename, flipped_horizontally=True),
    ]


class Entity(arcade.Sprite):
    def __init__(self, name_folder, name_file):
        super().__init__()

        # Default to facing right
        self.facing_direction = RIGHT_FACING

        # Used for image sequences
        self.cur_texture = 0
        self.scale = CHARACTER_SCALING

        main_path = f":resources:images/animated_characters/{name_folder}/{name_file}"

        self.idle_texture_pair = load_texture_pair(f"{main_path}_idle.png")
        self.jump_texture_pair = load_texture_pair(f"{main_path}_jump.png")
        self.fall_texture_pair = load_texture_pair(f"{main_path}_fall.png")

        # Load textures for walking
        self.walk_textures = []
        for i in range(8):
            texture = 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]

```

(continues on next page)

(continued from previous page)

```
        # Hit box will be set based on the first image used. If you want to specify
        # a different hit box, you can do it like the code below.
        # self.set_hit_box([[-22, -64], [22, -64], [22, 28], [-22, 28]])
        self.set_hit_box(self.texture.hit_box_points)


class Enemy(Entity):
    def __init__(self, name_folder, name_file):

        # Setup parent class
        super().__init__(name_folder, name_file)

        self.should_update_walk = 0
        self.health = 0

    def update_animation(self, delta_time: float = 1 / 60):

        # Figure out if we need to flip face left or right
        if self.change_x < 0 and self.facing_direction == RIGHT_FACING:
            self.facing_direction = LEFT_FACING
        elif self.change_x > 0 and self.facing_direction == LEFT_FACING:
            self.facing_direction = RIGHT_FACING

        # Idle animation
        if self.change_x == 0:
            self.texture = self.idle_texture_pair[self.facing_direction]
            return

        # Walking animation
        if self.should_update_walk == 3:
            self.cur_texture += 1
            if self.cur_texture > 7:
                self.cur_texture = 0
            self.texture = self.walk_textures[self.cur_texture][self.facing_direction]
            self.should_update_walk = 0
            return

        self.should_update_walk += 1


class RobotEnemy(Enemy):
    def __init__(self):

        # Set up parent class
        super().__init__("robot", "robot")

        self.health = 100


class ZombieEnemy(Enemy):
    def __init__(self):

```

(continues on next page)

(continued from previous page)

```python
        # Set up parent class
        super().__init__("zombie", "zombie")

        self.health = 50


class PlayerCharacter(Entity):
    """Player Sprite"""

    def __init__(self):

        # Set up parent class
        super().__init__("male_person", "malePerson")

        # Track our state
        self.jumping = False
        self.climbing = False
        self.is_on_ladder = False

    def update_animation(self, delta_time: float = 1 / 60):

        # Figure out if we need to flip face left or right
        if self.change_x < 0 and self.facing_direction == RIGHT_FACING:
            self.facing_direction = LEFT_FACING
        elif self.change_x > 0 and self.facing_direction == LEFT_FACING:
            self.facing_direction = RIGHT_FACING

        # Climbing animation
        if self.is_on_ladder:
            self.climbing = True
        if not self.is_on_ladder and self.climbing:
            self.climbing = False
        if self.climbing and abs(self.change_y) > 1:
            self.cur_texture += 1
            if self.cur_texture > 7:
                self.cur_texture = 0
        if self.climbing:
            self.texture = self.climbing_textures[self.cur_texture // 4]
            return

        # Jumping animation
        if self.change_y > 0 and not self.is_on_ladder:
            self.texture = self.jump_texture_pair[self.facing_direction]
            return
        elif self.change_y < 0 and not self.is_on_ladder:
            self.texture = self.fall_texture_pair[self.facing_direction]
            return

        # Idle animation
        if self.change_x == 0:
            self.texture = self.idle_texture_pair[self.facing_direction]
            return
```

(continues on next page)

(continued from previous page)

```python

        # Walking animation
        self.cur_texture += 1
        if self.cur_texture > 7:
            self.cur_texture = 0
        self.texture = self.walk_textures[self.cur_texture][self.facing_direction]


class MainMenu(arcade.View):
    """Class that manages the 'menu' view."""

    def on_show_view(self):
        """Called when switching to this view."""
        arcade.set_background_color(arcade.color.WHITE)

    def on_draw(self):
        """Draw the menu"""
        self.clear()
        arcade.draw_text(
            "Main Menu - Click to play",
            SCREEN_WIDTH / 2,
            SCREEN_HEIGHT / 2,
            arcade.color.BLACK,
            font_size=30,
            anchor_x="center",
        )

    def on_mouse_press(self, _x, _y, _button, _modifiers):
        """Use a mouse press to advance to the 'game' view."""
        game_view = GameView()
        self.window.show_view(game_view)


class GameView(arcade.View):
    """
    Main application class.
    """

    def __init__(self):
        """
        Initializer for the game
        """
        super().__init__()

        # Set the path to start with this program
        file_path = os.path.dirname(os.path.abspath(__file__))
        os.chdir(file_path)

        # Track the current state of what key is pressed
        self.left_pressed = False
        self.right_pressed = False
        self.up_pressed = False
```

(continues on next page)

(continued from previous page)

```
        self.down_pressed = False
        self.shoot_pressed = False
        self.jump_needs_reset = False

        # Our TileMap Object
        self.tile_map = None

        # Our Scene Object
        self.scene = None

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

        # Our 'physics' engine
        self.physics_engine = None

        # A Camera that can be used for scrolling the screen
        self.camera = None

        # A Camera that can be used to draw GUI elements
        self.gui_camera = None

        self.end_of_map = 0

        # Keep track of the score
        self.score = 0

        # Shooting mechanics
        self.can_shoot = False
        self.shoot_timer = 0

        # Load sounds
        self.collect_coin_sound = arcade.load_sound(":resources:sounds/coin1.wav")
        self.jump_sound = arcade.load_sound(":resources:sounds/jump1.wav")
        self.game_over = arcade.load_sound(":resources:sounds/gameover1.wav")
        self.shoot_sound = arcade.load_sound(":resources:sounds/hurt5.wav")
        self.hit_sound = arcade.load_sound(":resources:sounds/hit5.wav")

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

        # Set up the Cameras
        self.camera = arcade.Camera(self.window.width, self.window.height)
        self.gui_camera = arcade.Camera(self.window.width, self.window.height)

        # Map name
        map_name = ":resources:tiled_maps/map_with_ladders.json"

        # Layer Specific Options for the Tilemap
        layer_options = {
            LAYER_NAME_PLATFORMS: {
                "use_spatial_hash": True,
```

(continues on next page)

(continued from previous page)

```
            },
            LAYER_NAME_MOVING_PLATFORMS: {
                "use_spatial_hash": False,
            },
            LAYER_NAME_LADDERS: {
                "use_spatial_hash": True,
            },
            LAYER_NAME_COINS: {
                "use_spatial_hash": True,
            },
        }

        # Load in TileMap
        self.tile_map = arcade.load_tilemap(map_name, TILE_SCALING, layer_options)

        # Initiate New Scene with our TileMap, this will automatically add all layers
        # from the map as SpriteLists in the scene in the proper order.
        self.scene = arcade.Scene.from_tilemap(self.tile_map)

        # Keep track of the score
        self.score = 0

        # Shooting mechanics
        self.can_shoot = True
        self.shoot_timer = 0

        # Set up the player, specifically placing it at these coordinates.
        self.player_sprite = PlayerCharacter()
        self.player_sprite.center_x = (
            self.tile_map.tile_width * TILE_SCALING * PLAYER_START_X
        )
        self.player_sprite.center_y = (
            self.tile_map.tile_height * TILE_SCALING * PLAYER_START_Y
        )
        self.scene.add_sprite(LAYER_NAME_PLAYER, self.player_sprite)

        # Calculate the right edge of the my_map in pixels
        self.end_of_map = self.tile_map.width * GRID_PIXEL_SIZE

        # -- Enemies
        enemies_layer = self.tile_map.object_lists[LAYER_NAME_ENEMIES]

        for my_object in enemies_layer:
            cartesian = self.tile_map.get_cartesian(
                my_object.shape[0], my_object.shape[1]
            )
            enemy_type = my_object.properties["type"]
            if enemy_type == "robot":
                enemy = RobotEnemy()
            elif enemy_type == "zombie":
                enemy = ZombieEnemy()
            enemy.center_x = math.floor(
```

(continues on next page)

(continued from previous page)

```
                cartesian[0] * TILE_SCALING * self.tile_map.tile_width
            )
            enemy.center_y = math.floor(
                (cartesian[1] + 1) * (self.tile_map.tile_height * TILE_SCALING)
            )
            if "boundary_left" in my_object.properties:
                enemy.boundary_left = my_object.properties["boundary_left"]
            if "boundary_right" in my_object.properties:
                enemy.boundary_right = my_object.properties["boundary_right"]
            if "change_x" in my_object.properties:
                enemy.change_x = my_object.properties["change_x"]
            self.scene.add_sprite(LAYER_NAME_ENEMIES, enemy)

        # Add bullet spritelist to Scene
        self.scene.add_sprite_list(LAYER_NAME_BULLETS)

        # --- Other stuff
        # Set the background color
        if self.tile_map.background_color:
            arcade.set_background_color(self.tile_map.background_color)

        # Create the 'physics engine'
        self.physics_engine = arcade.PhysicsEnginePlatformer(
            self.player_sprite,
            platforms=self.scene[LAYER_NAME_MOVING_PLATFORMS],
            gravity_constant=GRAVITY,
            ladders=self.scene[LAYER_NAME_LADDERS],
            walls=self.scene[LAYER_NAME_PLATFORMS]
        )

    def on_show_view(self):
        self.setup()

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

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

        # Activate the game camera
        self.camera.use()

        # Draw our Scene
        self.scene.draw()

        # Activate the GUI camera before drawing GUI elements
        self.gui_camera.use()

        # Draw our score on the screen, scrolling it with the viewport
        score_text = f"Score: {self.score}"
        arcade.draw_text(
            score_text,
```

(continues on next page)

(continued from previous page)

```
            10,
            10,
            arcade.csscolor.BLACK,
            18,
        )

        # Draw hit boxes.
        # for wall in self.wall_list:
        #     wall.draw_hit_box(arcade.color.BLACK, 3)
        #
        # self.player_sprite.draw_hit_box(arcade.color.RED, 3)

    def process_keychange(self):
        """
        Called when we change a key up/down or we move on/off a ladder.
        """
        # Process up/down
        if self.up_pressed and not self.down_pressed:
            if self.physics_engine.is_on_ladder():
                self.player_sprite.change_y = PLAYER_MOVEMENT_SPEED
            elif (
                self.physics_engine.can_jump(y_distance=10)
                and not self.jump_needs_reset
            ):
                self.player_sprite.change_y = PLAYER_JUMP_SPEED
                self.jump_needs_reset = True
                arcade.play_sound(self.jump_sound)
        elif self.down_pressed and not self.up_pressed:
            if self.physics_engine.is_on_ladder():
                self.player_sprite.change_y = -PLAYER_MOVEMENT_SPEED

        # Process up/down when on a ladder and no movement
        if self.physics_engine.is_on_ladder():
            if not self.up_pressed and not self.down_pressed:
                self.player_sprite.change_y = 0
            elif self.up_pressed and self.down_pressed:
                self.player_sprite.change_y = 0

        # Process left/right
        if self.right_pressed and not self.left_pressed:
            self.player_sprite.change_x = PLAYER_MOVEMENT_SPEED
        elif self.left_pressed and not self.right_pressed:
            self.player_sprite.change_x = -PLAYER_MOVEMENT_SPEED
        else:
            self.player_sprite.change_x = 0

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

        if key == arcade.key.UP or key == arcade.key.W:
            self.up_pressed = True
        elif key == arcade.key.DOWN or key == arcade.key.S:
```

(continues on next page)

(continued from previous page)

```
            self.down_pressed = True
        elif key == arcade.key.LEFT or key == arcade.key.A:
            self.left_pressed = True
        elif key == arcade.key.RIGHT or key == arcade.key.D:
            self.right_pressed = True

        if key == arcade.key.Q:
            self.shoot_pressed = True

        if key == arcade.key.PLUS:
            self.camera.zoom(0.01)
        elif key == arcade.key.MINUS:
            self.camera.zoom(-0.01)

        self.process_keychange()

    def on_key_release(self, key, modifiers):
        """Called when the user releases a key."""

        if key == arcade.key.UP or key == arcade.key.W:
            self.up_pressed = False
            self.jump_needs_reset = False
        elif key == arcade.key.DOWN or key == arcade.key.S:
            self.down_pressed = False
        elif key == arcade.key.LEFT or key == arcade.key.A:
            self.left_pressed = False
        elif key == arcade.key.RIGHT or key == arcade.key.D:
            self.right_pressed = False

        if key == arcade.key.Q:
            self.shoot_pressed = False

        self.process_keychange()

    def on_mouse_scroll(self, x, y, scroll_x, scroll_y):
        self.camera.zoom(-0.01 * scroll_y)

    def center_camera_to_player(self, speed=0.2):
        screen_center_x = self.camera.scale * (self.player_sprite.center_x - (self.
↪camera.viewport_width / 2))
        screen_center_y = self.camera.scale * (self.player_sprite.center_y - (self.
↪camera.viewport_height / 2))
        if screen_center_x < 0:
            screen_center_x = 0
        if screen_center_y < 0:
            screen_center_y = 0
        player_centered = (screen_center_x, screen_center_y)

        self.camera.move_to(player_centered, speed)

    def on_update(self, delta_time):
        """Movement and game logic"""
```

(continues on next page)

(continued from previous page)

```python

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

        # Update animations
        if self.physics_engine.can_jump():
            self.player_sprite.can_jump = False
        else:
            self.player_sprite.can_jump = True

        if self.physics_engine.is_on_ladder() and not self.physics_engine.can_jump():
            self.player_sprite.is_on_ladder = True
            self.process_keychange()
        else:
            self.player_sprite.is_on_ladder = False
            self.process_keychange()

        if self.can_shoot:
            if self.shoot_pressed:
                arcade.play_sound(self.shoot_sound)
                bullet = arcade.Sprite(
                    ":resources:images/space_shooter/laserBlue01.png",
                    SPRITE_SCALING_LASER,
                )

                if self.player_sprite.facing_direction == RIGHT_FACING:
                    bullet.change_x = BULLET_SPEED
                else:
                    bullet.change_x = -BULLET_SPEED

                bullet.center_x = self.player_sprite.center_x
                bullet.center_y = self.player_sprite.center_y

                self.scene.add_sprite(LAYER_NAME_BULLETS, bullet)

                self.can_shoot = False
        else:
            self.shoot_timer += 1
            if self.shoot_timer == SHOOT_SPEED:
                self.can_shoot = True
                self.shoot_timer = 0

        # Update Animations
        self.scene.update_animation(
            delta_time,
            [
                LAYER_NAME_COINS,
                LAYER_NAME_BACKGROUND,
                LAYER_NAME_PLAYER,
                LAYER_NAME_ENEMIES,
            ],
        )
```

(continues on next page)

(continued from previous page)

```

        # Update moving platforms, enemies, and bullets
        self.scene.update(
            [LAYER_NAME_MOVING_PLATFORMS, LAYER_NAME_ENEMIES, LAYER_NAME_BULLETS]
        )

        # See if the enemy hit a boundary and needs to reverse direction.
        for enemy in self.scene[LAYER_NAME_ENEMIES]:
            if (
                enemy.boundary_right
                and enemy.right > enemy.boundary_right
                and enemy.change_x > 0
            ):
                enemy.change_x *= -1

            if (
                enemy.boundary_left
                and enemy.left < enemy.boundary_left
                and enemy.change_x < 0
            ):
                enemy.change_x *= -1

        for bullet in self.scene[LAYER_NAME_BULLETS]:
            hit_list = arcade.check_for_collision_with_lists(
                bullet,
                [
                    self.scene[LAYER_NAME_ENEMIES],
                    self.scene[LAYER_NAME_PLATFORMS],
                    self.scene[LAYER_NAME_MOVING_PLATFORMS],
                ],
            )

            if hit_list:
                bullet.remove_from_sprite_lists()

                for collision in hit_list:
                    if (
                        self.scene[LAYER_NAME_ENEMIES]
                        in collision.sprite_lists
                    ):
                        # The collision was with an enemy
                        collision.health -= BULLET_DAMAGE

                        if collision.health <= 0:
                            collision.remove_from_sprite_lists()
                            self.score += 100

                        # Hit sound
                        arcade.play_sound(self.hit_sound)

                return

```

(continues on next page)

(continued from previous page)

```
            if (bullet.right < 0) or (
                bullet.left
                > (self.tile_map.width * self.tile_map.tile_width) * TILE_SCALING
            ):
                bullet.remove_from_sprite_lists()

        player_collision_list = arcade.check_for_collision_with_lists(
            self.player_sprite,
            [
                self.scene[LAYER_NAME_COINS],
                self.scene[LAYER_NAME_ENEMIES],
            ],
        )

        # Loop through each coin we hit (if any) and remove it
        for collision in player_collision_list:

            if self.scene[LAYER_NAME_ENEMIES] in collision.sprite_lists:
                arcade.play_sound(self.game_over)
                game_over = GameOverView()
                self.window.show_view(game_over)
                return
            else:
                # Figure out how many points this coin is worth
                if "Points" not in collision.properties:
                    print("Warning, collected a coin without a Points property.")
                else:
                    points = int(collision.properties["Points"])
                    self.score += points

                # Remove the coin
                collision.remove_from_sprite_lists()
                arcade.play_sound(self.collect_coin_sound)

        # Position the camera
        self.center_camera_to_player()


class GameOverView(arcade.View):
    """Class to manage the game overview"""

    def on_show_view(self):
        """Called when switching to this view"""
        arcade.set_background_color(arcade.color.BLACK)

    def on_draw(self):
        """Draw the game overview"""
        self.clear()
        arcade.draw_text(
            "Game Over - Click to restart",
            SCREEN_WIDTH / 2,
            SCREEN_HEIGHT / 2,
```

(continues on next page)

(continued from previous page)

```
            arcade.color.WHITE,
            30,
            anchor_x="center",
        )

    def on_mouse_press(self, _x, _y, _button, _modifiers):
        """Use a mouse press to advance to the 'game' view."""
        game_view = GameView()
        self.window.show_view(game_view)


def main():
    """Main function"""
    window = arcade.Window(SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_TITLE)
    menu_view = MainMenu()
    window.show_view(menu_view)
    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

This tutorial is also being expanded into a fully featured game developed by the Arcade community. You can check out that project on Github at https://github.com/pythonarcade/community-platformer

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

```python
"""
Example of Pymunk Physics Engine Platformer
"""
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.

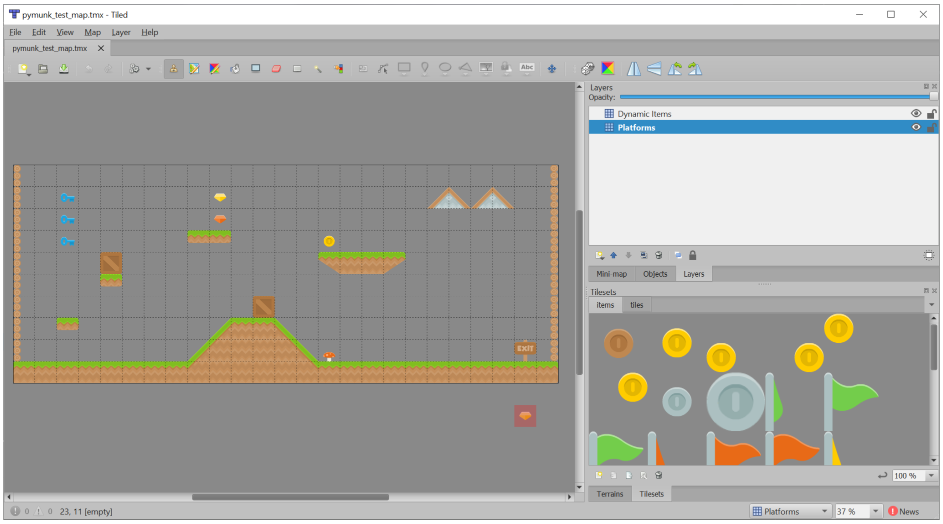
Listing 3: Create instance variables

```
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
        arcade.set_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 - Display The Score*.

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

```
    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)

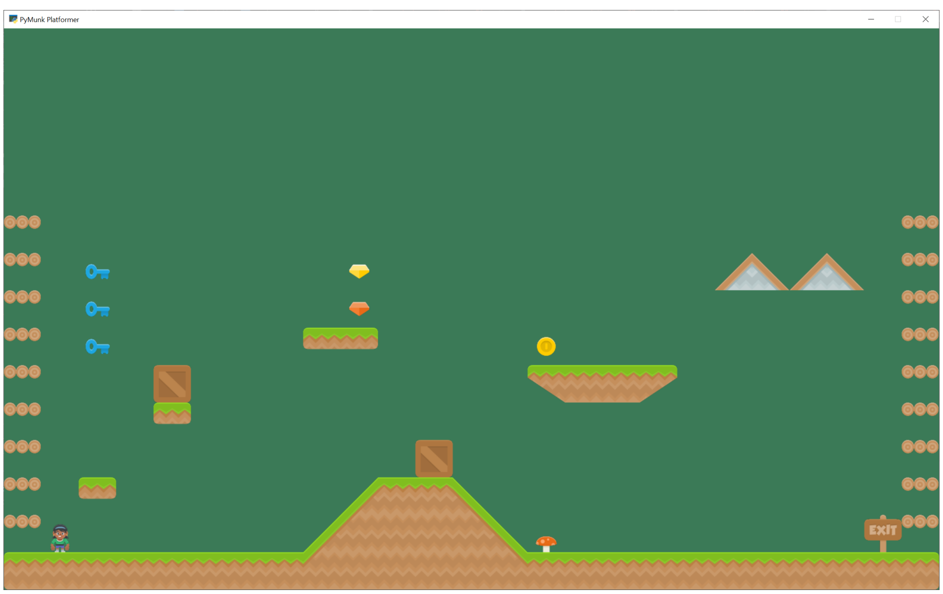
```
        self.wall_list = tile_map.sprite_lists["Platforms"]
        self.item_list = tile_map.sprite_lists["Dynamic Items"]

        # 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

```
    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)

```python

        # 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 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=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

```
                                              collision_type="item")

    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
```

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

```
            self.right_pressed = False

    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)

```

- 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

```
# 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

```
                                              friction=DYNAMIC_ITEM_FRICTION,
                                              collision_type="item")

    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):
```

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

```
            self.right_pressed = False

    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:
```

(continues on next page)

(continued from previous page)

```
            # Create a force to the left. 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)
        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.
```

- 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

```
# Close enough to not-moving to have the animation go to idle.
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

```
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]

        # Hit box will be set based on the first image used.
        self.hit_box = self.texture.hit_box_points

        # Default to face-right
        self.character_face_direction = RIGHT_FACING
```

(continues on next page)

(continued from previous page)

```python

        # Index of our current texture
        self.cur_texture = 0

        # 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

```
        # Pull the sprite layers out of the tile map
        self.wall_list = tile_map.sprite_lists["Platforms"]
```

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

```
        super().__init__(width, height, title)

```

- 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(20, 5, 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,
                                       friction=0.6,
```

(continues on next page)

(continued from previous page)

```
                                       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

```
        if key == arcade.key.LEFT:
            self.left_pressed = False
        elif key == arcade.key.RIGHT:
            self.right_pressed = False

```

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

```

        # Create the physics engine
        self.physics_engine = arcade.PymunkPhysicsEngine(damping=damping,
                                                         gravity=gravity)

        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):
```

- 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
- right_boundary

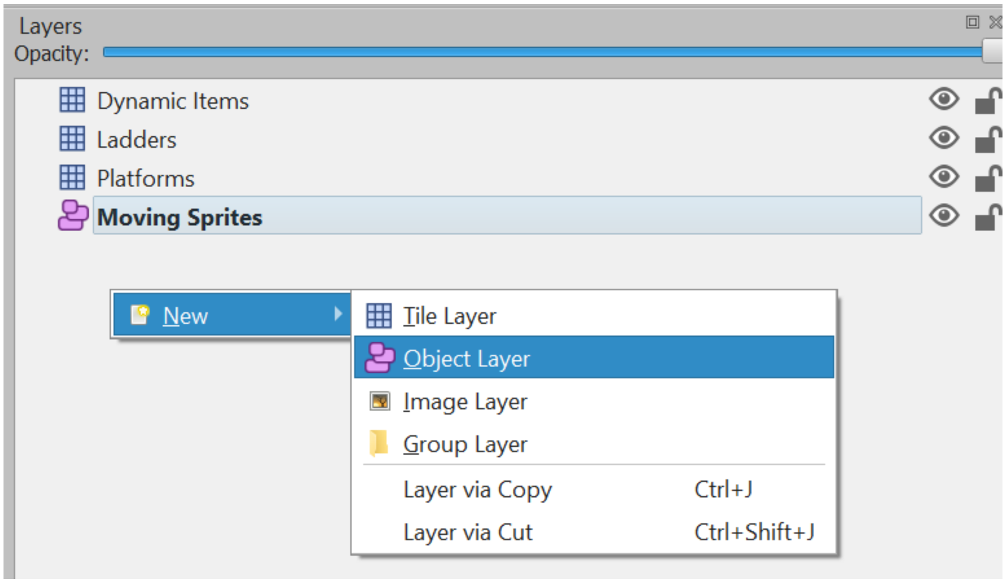


Fig. 1: Adding an object layer.

- `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.wall_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.player_sprite.center_y = SPRITE_SIZE * grid_y + SPRITE_SIZE / 2
        # Add to player sprite list
        self.player_list.append(self.player_sprite)
```

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

```
        # Create the items
        self.physics_engine.add_sprite_list(self.item_list,
                                            friction=DYNAMIC_ITEM_FRICTION,
```

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.

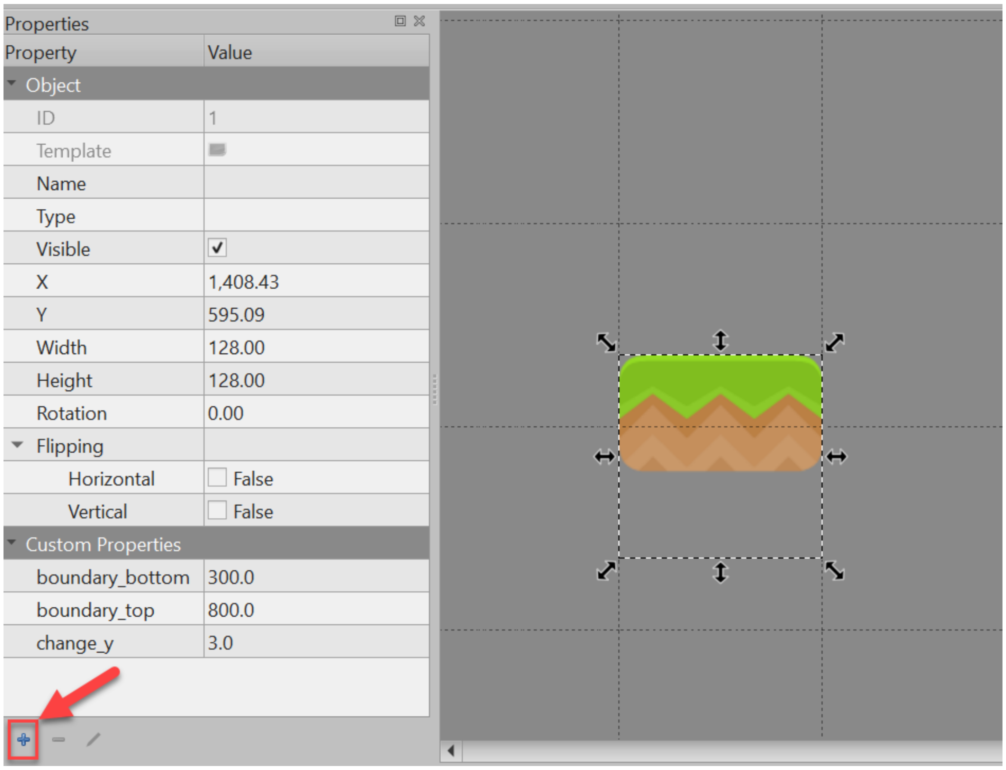


Fig. 2: Adding custom properties.

Listing 28: Moving Platforms - Draw the sprites

```
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

```
            self.physics_engine.set_friction(self.player_sprite, 1.0)

        # Move items in the physics engine
        self.physics_engine.step()

        # 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
```

- 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):
        """ Init """
        # Let parent initialize
        super().__init__()

        # Set our scale
        self.scale = SPRITE_SCALING_PLAYER
```

(continues on next page)

(continued from previous page)

```python

        # 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]

        # Hit box will be set based on the first image used.
        self.hit_box = self.texture.hit_box_points

        # 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

```
    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:
```

(continues on next page)

(continued from previous page)

```
                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]
```

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
```

(continues on next page)

(continued from previous page)

```
        self.physics_engine: Optional[arcade.PymunkPhysicsEngine] = None

        # Set background color
        arcade.set_background_color(arcade.color.AMAZON)
```

Then load the ladder layer in `setup`:

Listing 33: Add Ladders - Game Window Setup

```
        tile_map = arcade.load_tilemap(map_name, SPRITE_SCALING_TILES)

        # Pull the sprite layers out of the tile map
        self.wall_list = tile_map.sprite_lists["Platforms"]
```

Also, pass the ladder list to the player class:

Listing 34: Add Ladders - Game Window Setup

```
        self.ladder_list = tile_map.sprite_lists["Ladders"]
        self.moving_sprites_list = tile_map.sprite_lists['Moving Platforms']
```

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

```
    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

```
    def on_key_release(self, key, modifiers):
        """Called when the user releases a key. """

        if key == arcade.key.LEFT:
            self.left_pressed = False
```

(continues on next page)

(continued from previous page)

```
        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

```

        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)

        else:
```

And, of course, don't forget to draw the ladders:

Listing 38: Add Ladders - Game Window Key Down

```
    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:

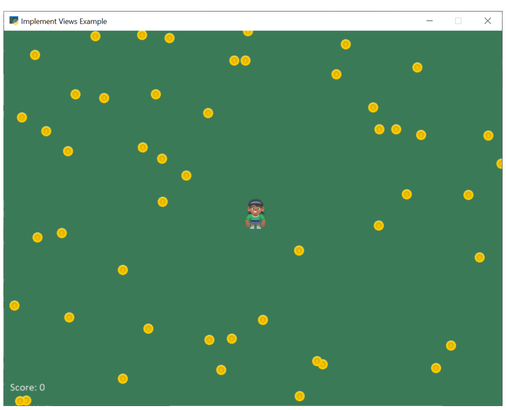
- Start screens
- Instruction screens
- Game over screens
- Pause screens

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 *Instruction Screens and Game Over Screens*. 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.

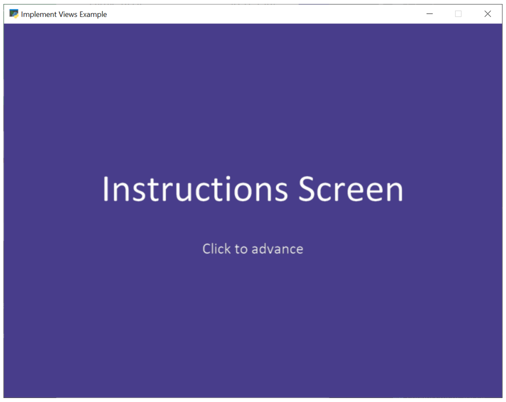
Listing 1: Add views - Main function

```
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:

```
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

```
    def on_show_view(self):
        """ This is run once when we switch to this view """
        arcade.set_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

```
    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)
```

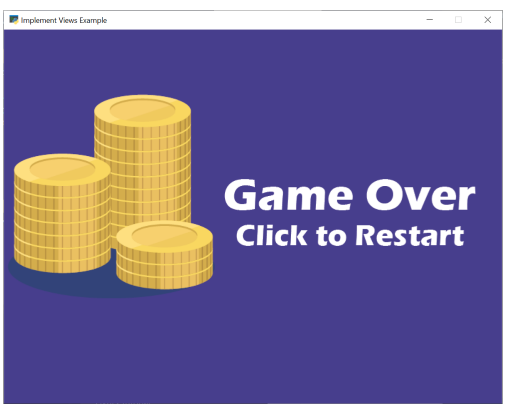
Now we need to go back to the main function. Instead of creating a GameView it needs to now create an InstructionView.

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

```
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

```
    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 TUTORIAL

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

```
"""
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)

        arcade.set_background_color(arcade.color.AMAZON)
```

(continues on next page)

(continued from previous page)

```python

    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

```python
# Constants for sizing
CARD_SCALE = 0.6

```

(continues on next page)

(continued from previous page)

```
# How big are the cards?
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

```
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

```
    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

        arcade.set_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

```
    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)
```

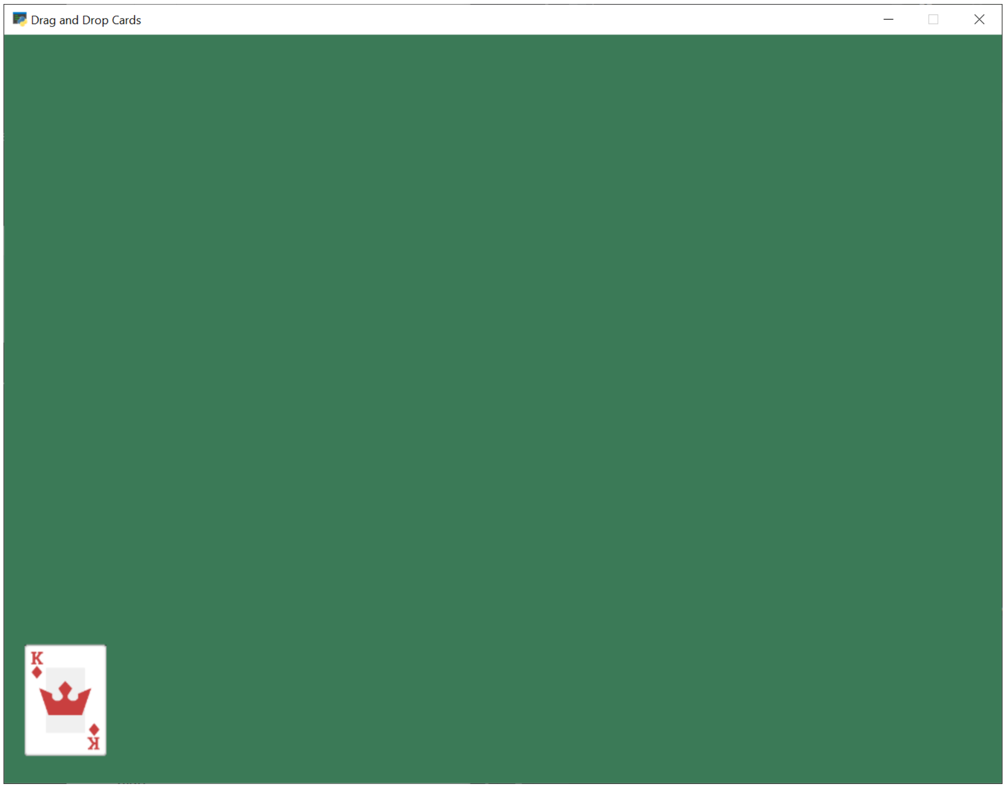
11.2.4 Drawing Cards

Finally, draw the cards:

Listing 6: Create card sprites

```
    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

        arcade.set_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
```

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

```python
    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

```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]

            # 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:
            card.center_x += dx
            card.center_y += dy
```

11.3.5 Mouse Released

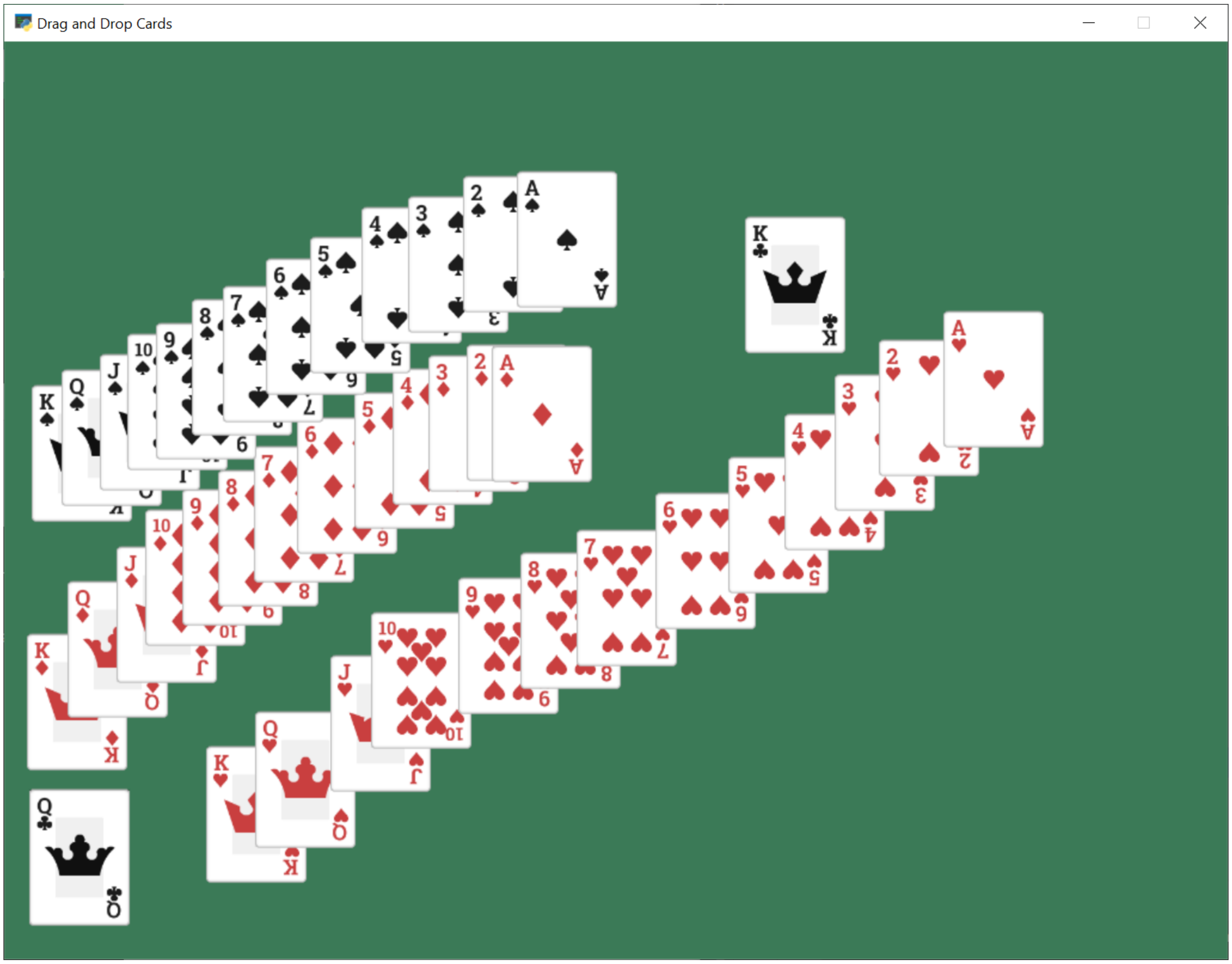
When the user releases the mouse button, we'll clear the held card list.

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)

```
# 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

```
    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

        arcade.set_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

```
    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_
↪GREEN)
```

(continues on next page)

(continued from previous page)

```
        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

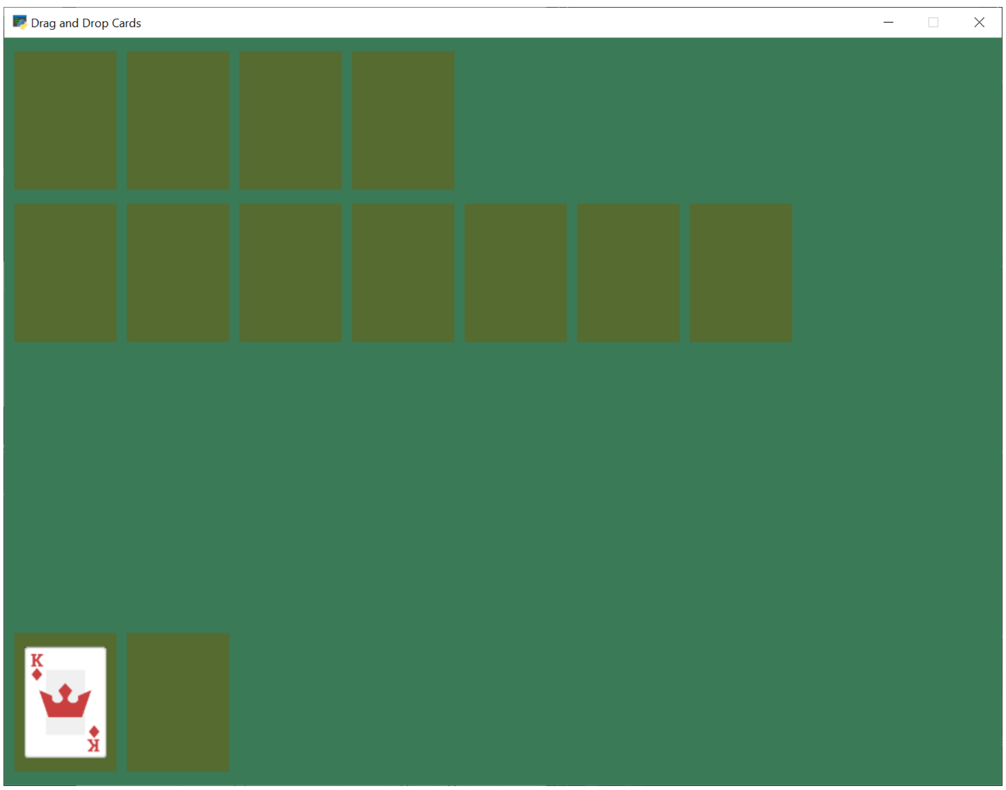
Finally, the mats aren't going to display if we don't draw them:

Listing 16: Draw the mat sprites

```
    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

```
    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

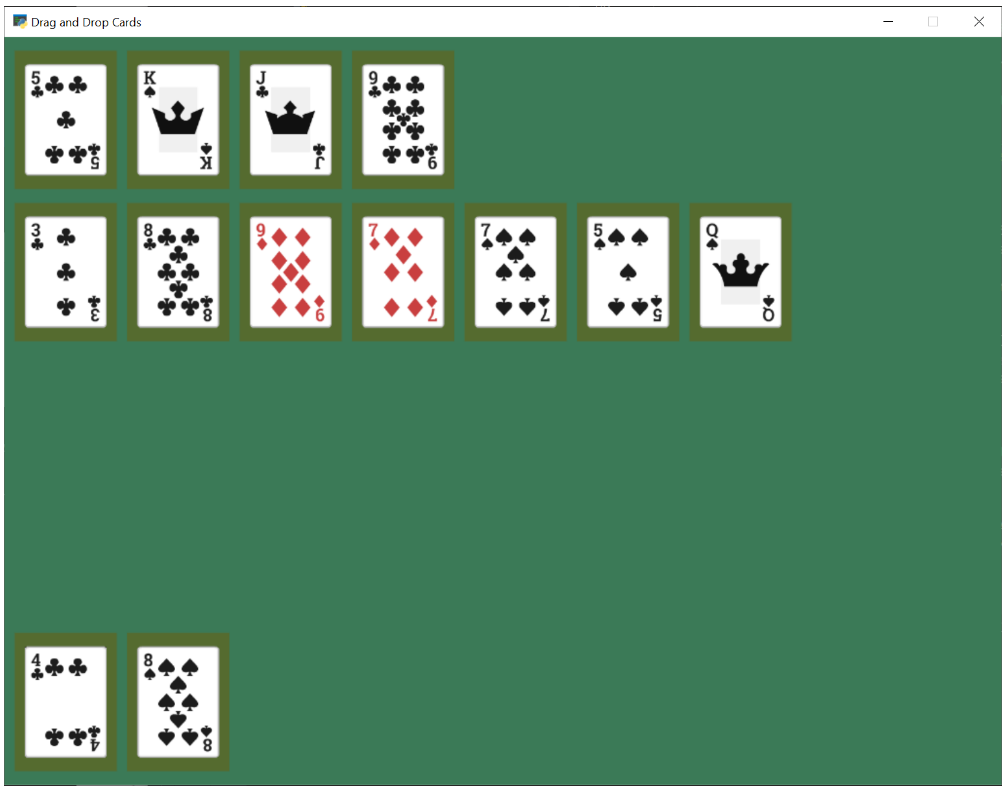
Having all the cards in order is boring. Let's shuffle them in the `setup` method:

Listing 18: Shuffle Cards

```
        # 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

```
# 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

```
        # 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

```python
    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

```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):

            # 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

```python
            # 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

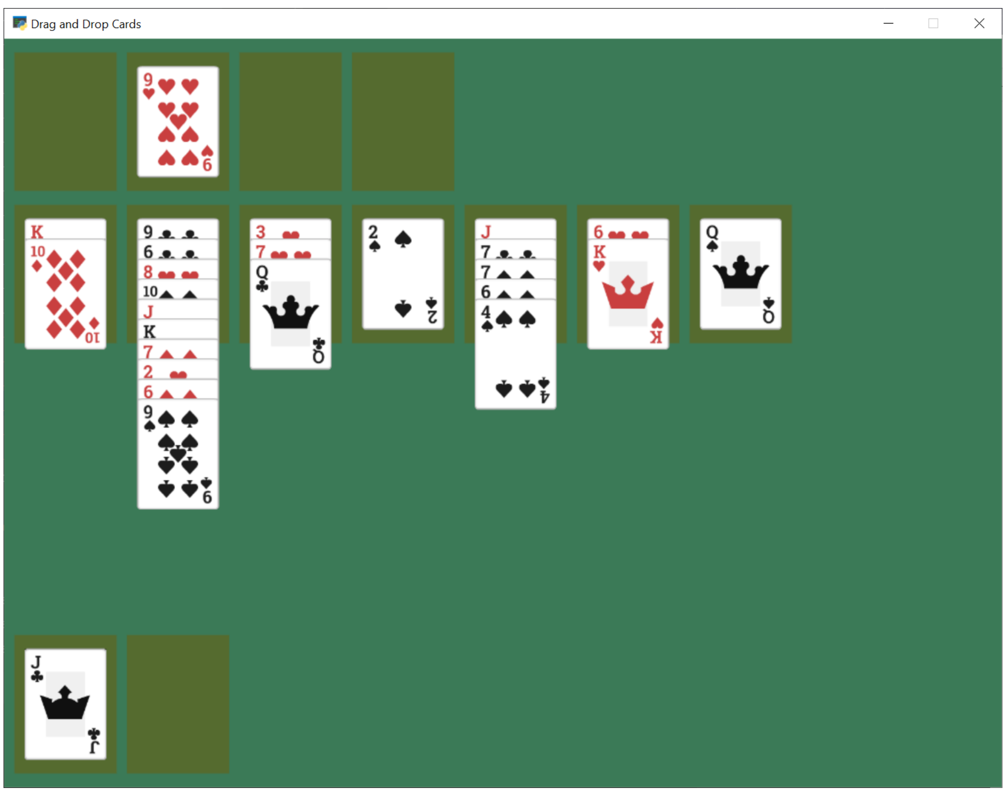
```
        if reset_position:
            # Where-ever we were dropped, it wasn't valid. Reset the each card's position
            # to its original spot.
            for pile_index, card in enumerate(self.held_cards):
                card.position = self.held_cards_original_position[pile_index]

        # 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)

```python
        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

```python
# 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

```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"
        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:
```

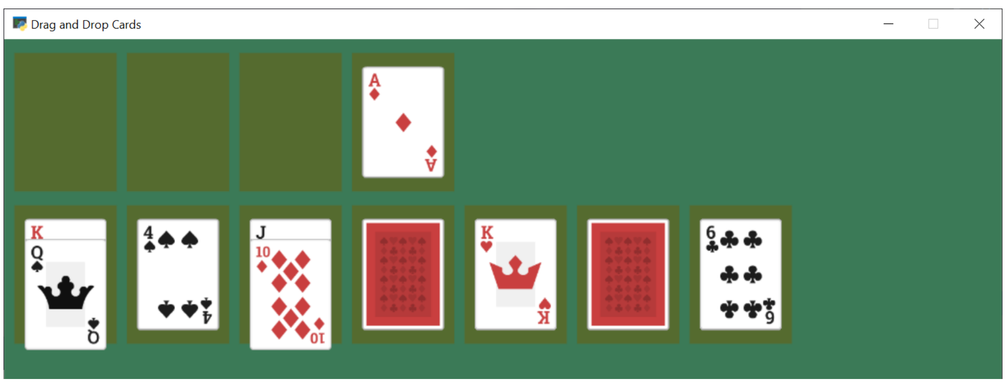
(continues on next page)

(continued from previous page)

```
                # 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

```
    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

```
    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
```

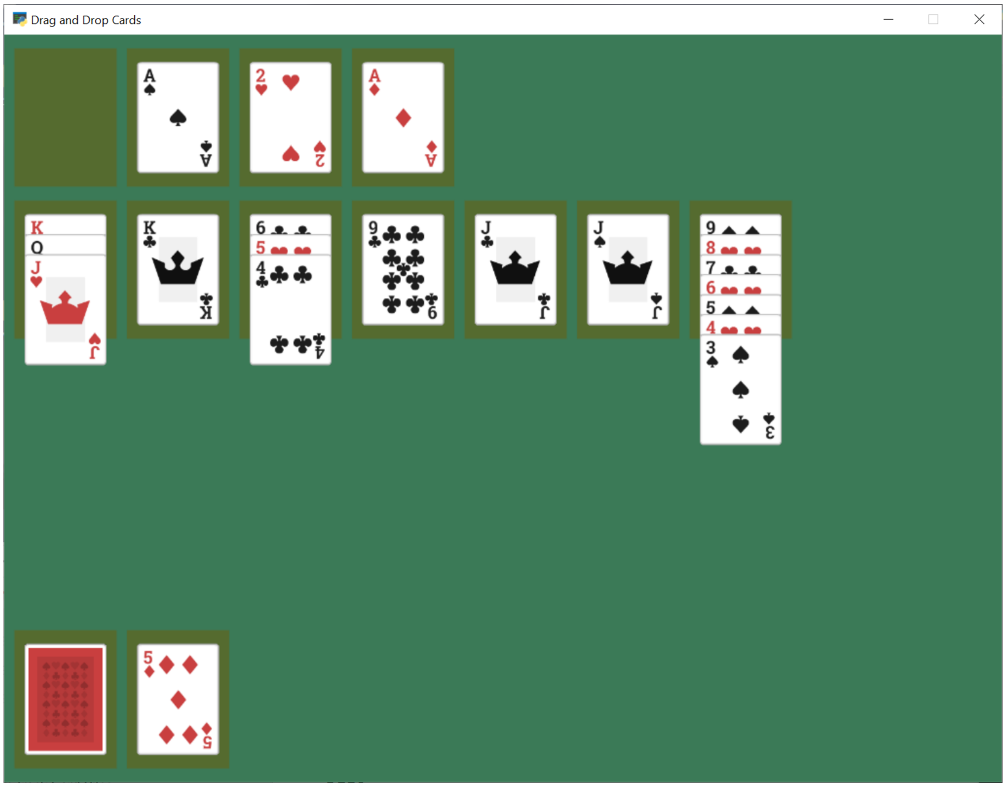
(continues on next page)

(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

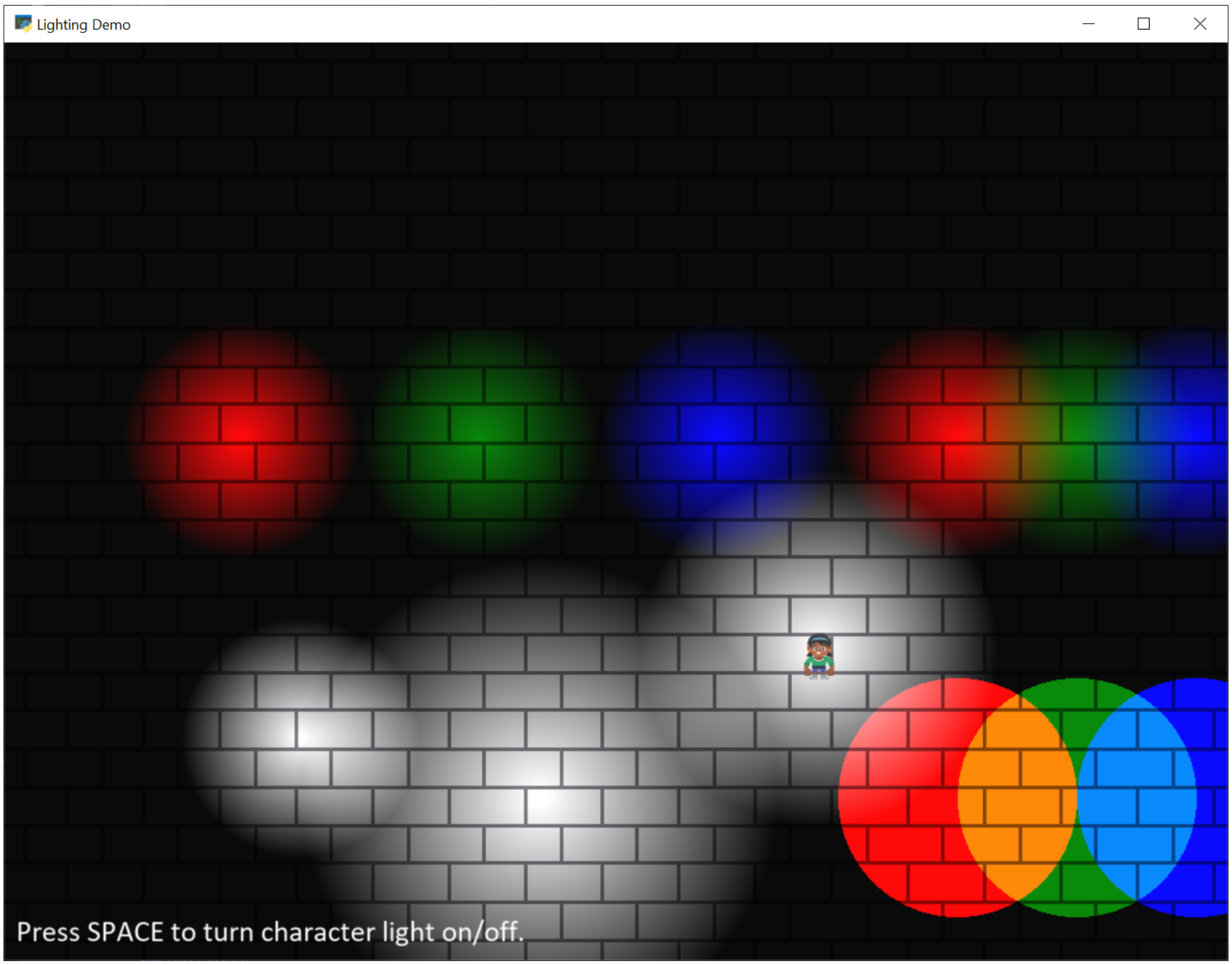
11.13 Conclusion

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 TUTORIAL



(To be done.)

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)

```
          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)

```
        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

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.

13.1 Step 1: Open a Blank Window

First, let's start with a blank window.

Listing 1: gpu_particle_burst_01.py

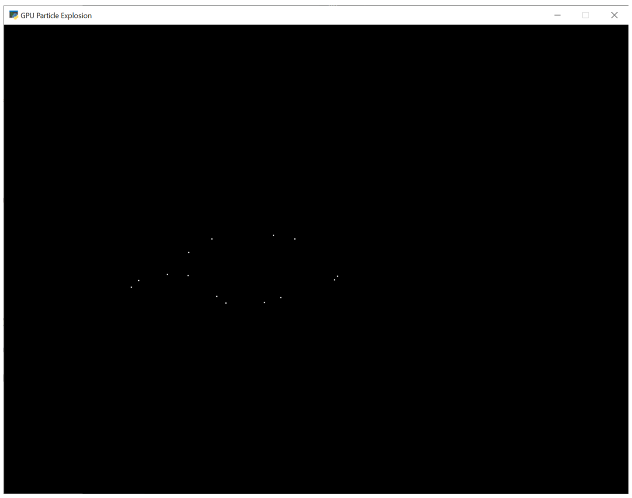
```python
"""
Example showing how to create particle explosions via the GPU.
"""
import arcade

SCREEN_WIDTH = 1024
SCREEN_HEIGHT = 768
SCREEN_TITLE = "GPU Particle Explosion"


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


```

(continues on next page)

(continued from previous page)

```
if __name__ == "__main__":
    window = MyWindow()
    window.center_window()
    arcade.run()
```

13.2 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.

13.2.1 Imports

First, we'll import some more items for our program:

```
from array import array
from dataclasses import dataclass
import arcade
import arcade.gl
```

13.2.2 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.

```
@dataclass
class Burst:
    """ Track for each burst. """
    buffer: arcade.gl.Buffer
    vao: arcade.gl.Geometry
```

13.2.3 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 2: 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()
```

13.2.4 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 3: 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
```

(continues on next page)

(continued from previous page)

```
    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 4: fragment_shader.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);
}
```

13.2.5 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 5: MyWindow.on_mouse_press

```
    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.
```

(continues on next page)

(continued from previous page)

```
        # 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 that says how the buffer data is formatted.
        buffer_description = arcade.gl.BufferDescription(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)
```

13.2.6 Drawing

Finally, draw it.

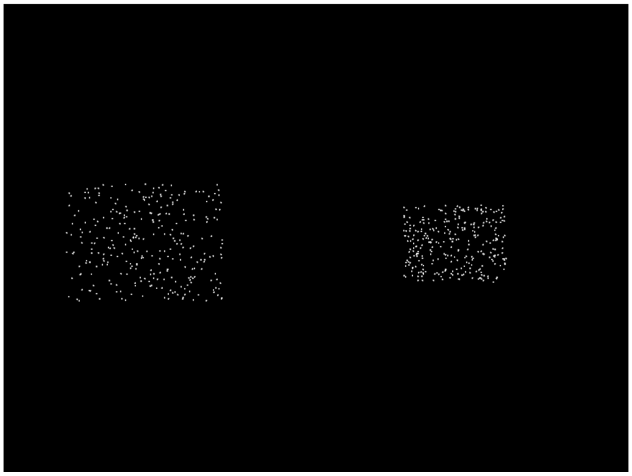
Listing 6: MyWindow.on_draw

```
    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)
```

13.2.7 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

13.3 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.

13.3.1 Imports

First, we'll import both the random and time libraries:

```
import random
import time
```

13.3.2 Constants

Then we need to create a constant that contains the number of particles to create:

```
PARTICLE_COUNT = 300
```

13.3.3 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
```

13.3.4 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 that says how the buffer data is formatted.
        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)
```

13.3.5 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.

```
    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)
```

13.3.6 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 7: vertex_shader_v2.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);

```

(continues on next page)

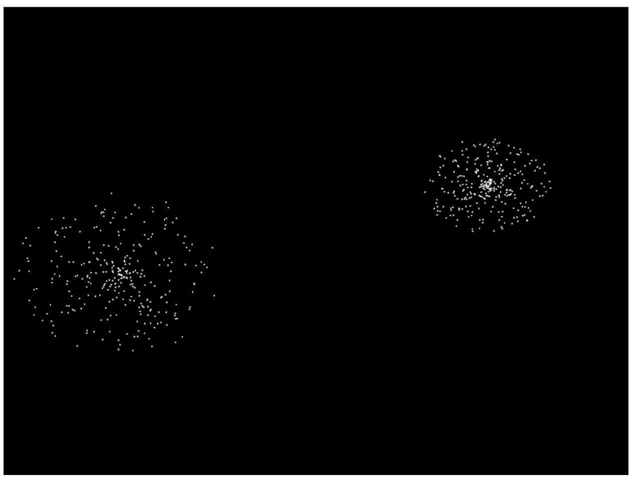
(continued from previous page)

```
    // 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);
}
```

13.3.7 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

13.4 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.

13.4.1 Update Imports

Import the math library so we can do some trig:

```
import math
```

13.4.2 Update Burst Creation

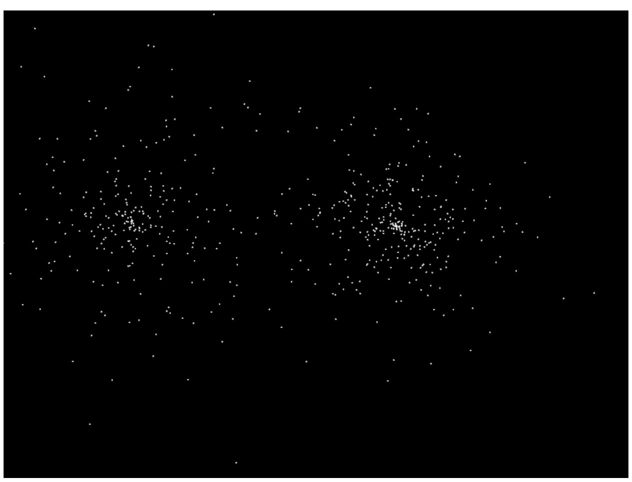
Now, pick a random direction from zero to 2 pi radians. Also, pick a random speed. Then use sine and cosine to calculate the delta x and y.

```
    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

```

13.4.3 Program Listings

- gpu_particle_burst_04 ← Where we are right now
- gpu_particle_burst_04_diff ← What we changed to get here

13.5 Step 5: Gaussian Distribution



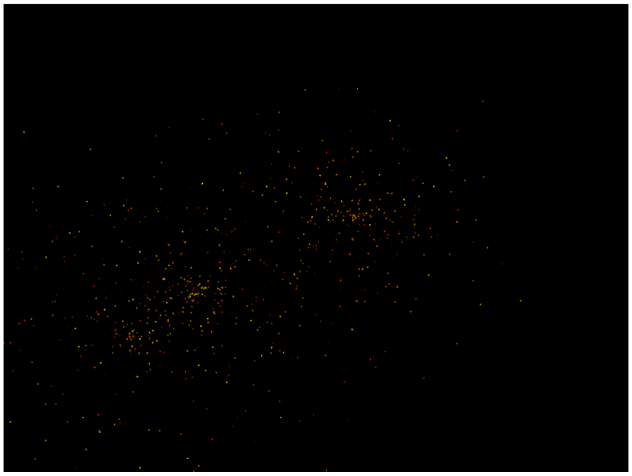
Setting speed to a random amount makes for an expanding circle. Another option is to use a gaussian function to produce more of a 'splat' look:

```
            for i in range(PARTICLE_COUNT):
```

13.5.1 Program Listings

- gpu_particle_burst_05 ← Where we are right now
- gpu_particle_burst_05_diff ← What we changed to get here

13.6 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, pass in the three floats as `in_color` to the shader buffer (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
                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 that says how the buffer data is formatted.
        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:

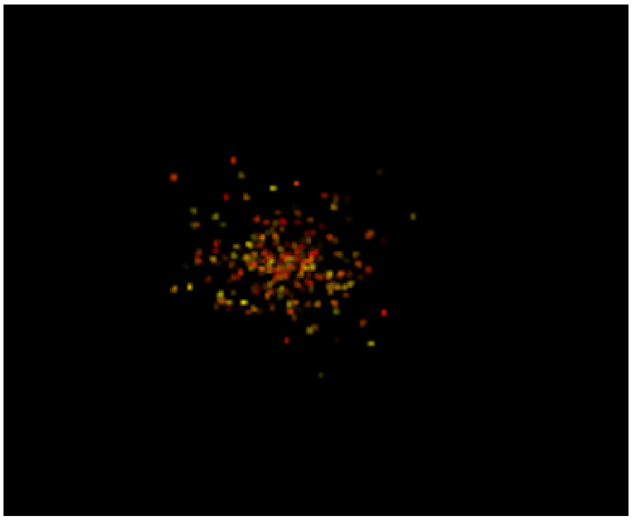
Listing 8: vertex_shader_v3.glsl

```
#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)
    gl_Position = vec4(new_pos, 0.0, 1);
}
```

13.6.1 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

13.7 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`.

13.7.1 Constants

First, let's add a couple constants to control the minimum and maximum tile to fade a particle:

```
MIN_FADE_TIME = 0.25
MAX_FADE_TIME = 1.5
```

13.7.2 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)
```

13.7.3 Add Fade Rate to Buffer

Next, add the fade rate 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 that says how the buffer data is formatted.
        buffer_description = arcade.gl.BufferDescription(buffer,
                                                         '2f 2f 3f f',
                                                         ['in_pos',
                                                          'in_vel',
                                                          'in_color',
                                                          'in_fade_rate'])
        # 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)
```

13.7.4 Update Shader

Update the shader. Calculate the alpha. If it is less that 0, just use 0.

Listing 9: 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);
}
```

13.7.5 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.

```
    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
```

(continues on next page)

(continued from previous page)

```
        # 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)
```

13.7.6 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

13.8 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.)

```
// 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);
```

13.8.1 Program Listings

- vertex_shader_v5 ← Where we are right now
- vertex_shader_v5_diff ← What we changed to get here

CHAPTER

FOURTEEN

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.

14.1 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

```
import arcade

arcade.open_window(400, 400, "My Game")

self.clear()
arcade.draw_circle_filled(200, 200, 100, arcade.color.BLUE)
arcade.finish_render()
```

(continues on next page)

(continued from previous page)

```
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.

14.2 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.

14.2.1 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;."
```

14.2.2 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"
```

14.2.3 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"
```

14.2.4 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
```

(continues on next page)

(continued from previous page)

```
    |--- 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!

14.3 Troubleshooting

14.3.1 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.

14.3.2 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.

14.4 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.

14.5 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

FIFTEEN

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. **Note that the build is platform dependent, meaning a Windows build will not work out-of-the-box on another OS.**

15.1 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 upto 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!

15.2 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.

15.3 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!

15.4 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

15.5 Additional Information

- This tutorial was tested with Nutika 0.7.x. Later releases are likely to work.

CHAPTER

SIXTEEN

SHADER TOY TUTORIAL - GLOW

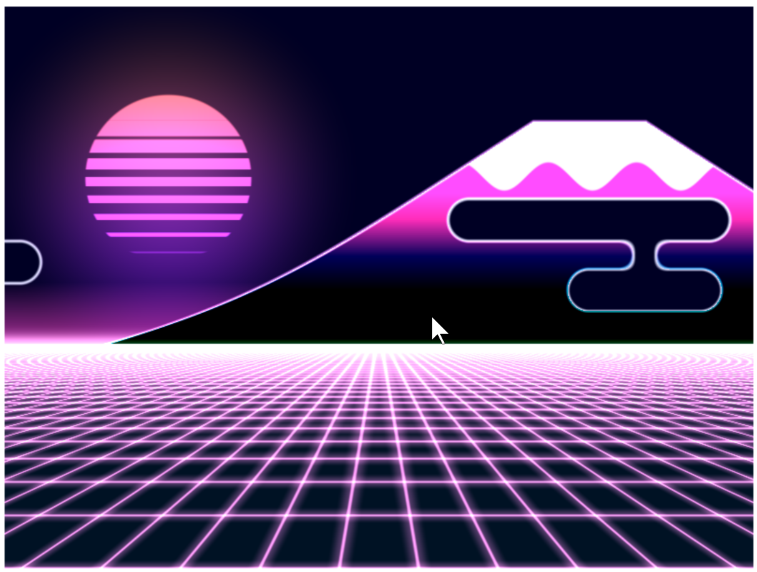


Fig. 1: cyber_fuji_2020

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.

16.1 PyCon 2022 Slides

This tutorial is scheduled to be presented at 2022 PyCon US. Here are the slides for that presentation:

16.2 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 1: Open a window

```
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()
```

16.3 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 2: Run a shader

```
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()
```

(continues on next page)

(continued from previous page)

```
        self.shadertoy = Shadertoy.create_from_file(window_size, shader_file_path)

    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:

```
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)
```

16.4 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 3: GLSL code for creating a shader.

```
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
```

(continues on next page)

(continued from previous page)

```
        fragColor = vec4(1.0, 1.0, 1.0, 1.0);
    }
}
```

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.

16.5 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.

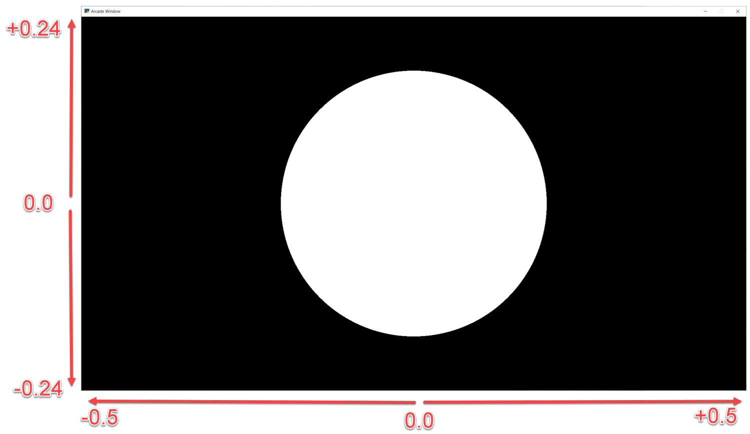
Listing 4: Center the origin

```
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;
```

(continues on next page)

(continued from previous page)

```

    // How far is the current pixel from the origin (0, 0)
    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);
    }
}
```



16.6 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 5: 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
```

(continues on next page)

(continued from previous page)

```
    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 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);
}
```

16.7 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.

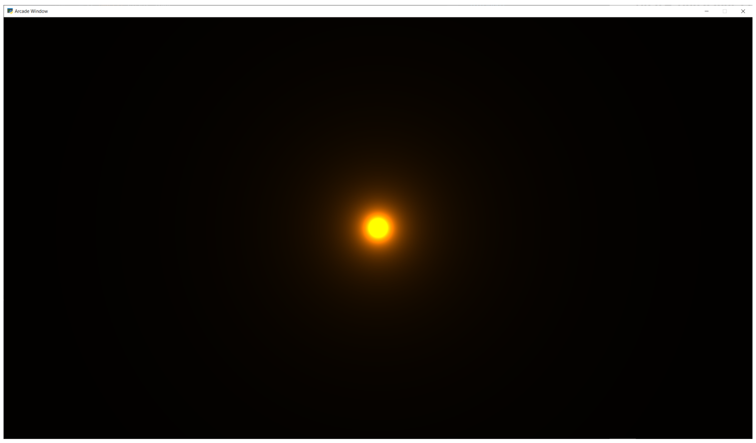
Listing 6: Adjusts fade speed

```
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);
```

(continues on next page)

(continued from previous page)

```
    // 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);

    // Fade our orange color
    vec3 color = strength * vec3(1.0, 0.5, 0.0);

    // Output to the screen
    fragColor = vec4(color, 1.0);
}
```



16.8 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.

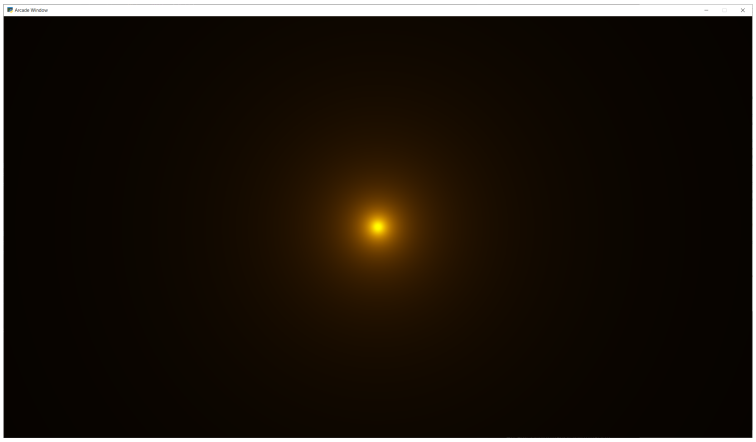
Listing 7: Tone mapping

```
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
```

(continues on next page)

(continued from previous page)

```
    vec3 color = strength * vec3(1.0, 0.5, 0);

    // Tone mapping
    color = 1.0 - exp( -color );

    // Output to the screen
    fragColor = vec4(color, 1.0);
}
```



16.9 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 8: Run a shader

```
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_6.glsl"
        window_size = self.get_size()
        self.shadertoy = Shadertoy.create_from_file(window_size, shader_file_path)

    def on_draw(self):
        # Set uniform data to send to the GLSL shader
```

(continues on next page)

(continued from previous page)

```
        self.shadertoy.program['pos'] = self.mouse["x"], self.mouse["y"]
        self.shadertoy.program['color'] = arcade.get_three_float_color(arcade.color.
↪LIGHT_BLUE)
        # Run the GLSL code
        self.shadertoy.render()

if __name__ == "__main__":
    MyGame()
    arcade.run()
```

Then we can use those uniforms in our shader:

Listing 9: 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.

16.10 Other examples

Here's another Python program that loads a GLSL file and displays it:

Listing 10: Shader Toy Demo

```
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:

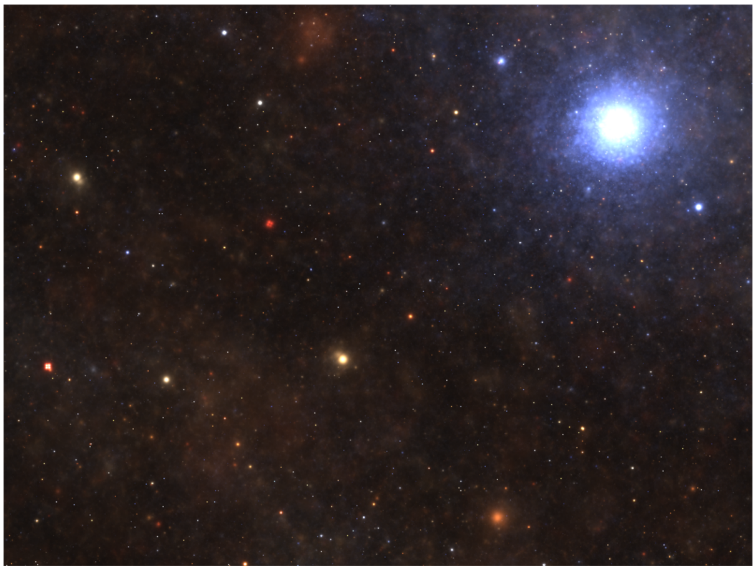


Fig. 2: star_nest

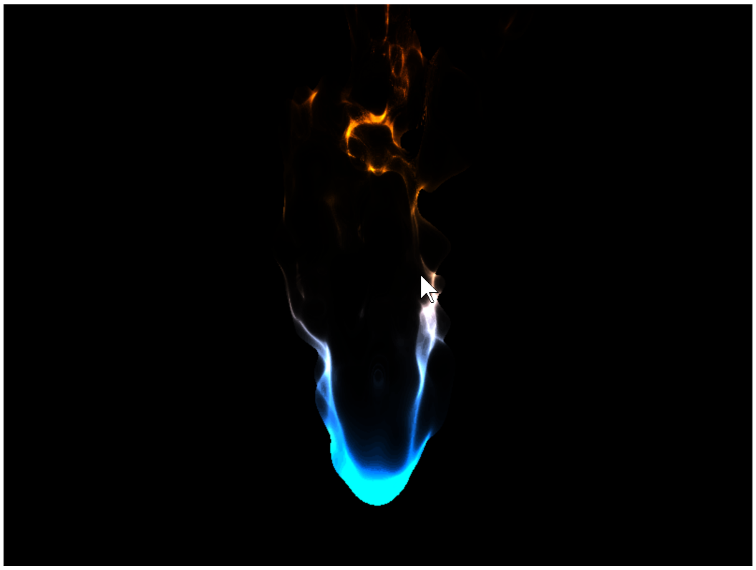


Fig. 3: flame

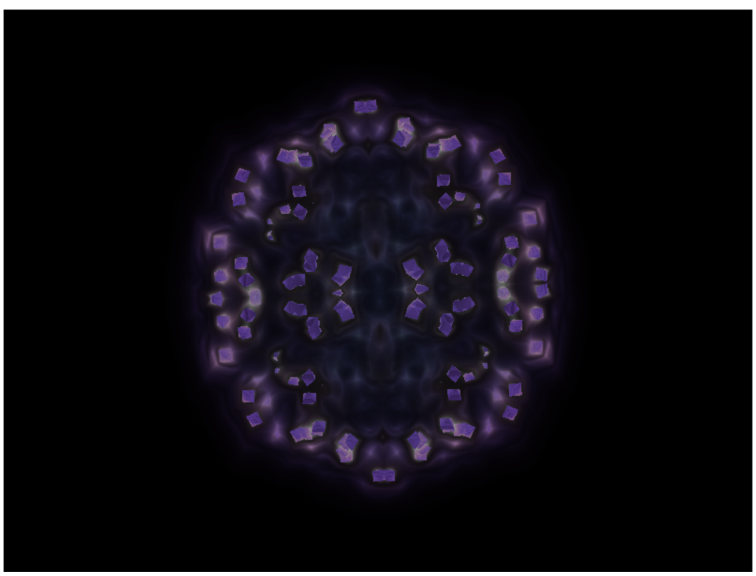


Fig. 4: fractal_pyramid

16.11 Additional learning

On this site:

- Learn a method of creating particles in *Shader Toy Tutorial - 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 Notes*.
- Learn to do a compute shader in *Compute Shader Tutorial*.

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

CHAPTER

SEVENTEEN

SHADER TOY TUTORIAL - PARTICLES

Contents

This tutorial assumes you are already familiar with the material in *Shader Toy Tutorial - 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.

17.1 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.

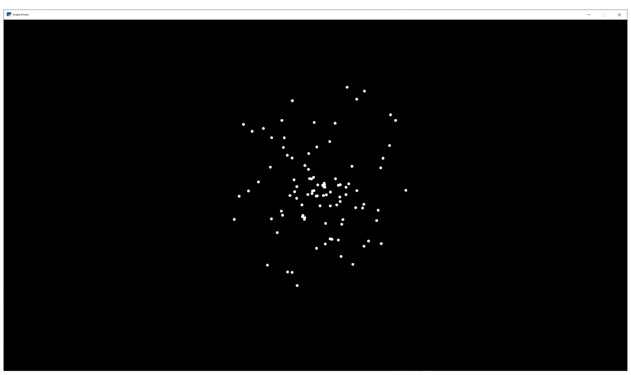
```
import arcade
from arcade.experimental import Shadertoy


# 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)

        # Used to track run-time
        self.time = 0.0
```

(continues on next page)

(continued from previous page)

```python

        # 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()
```

17.2 Initial shader with particles



```glsl
// 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.
```

(continues on next page)

(continued from previous page)

```
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);
}
```

17.3 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
```

(continues on next page)

(continued from previous page)

```
        // 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);
}
```

17.4 Fade-out

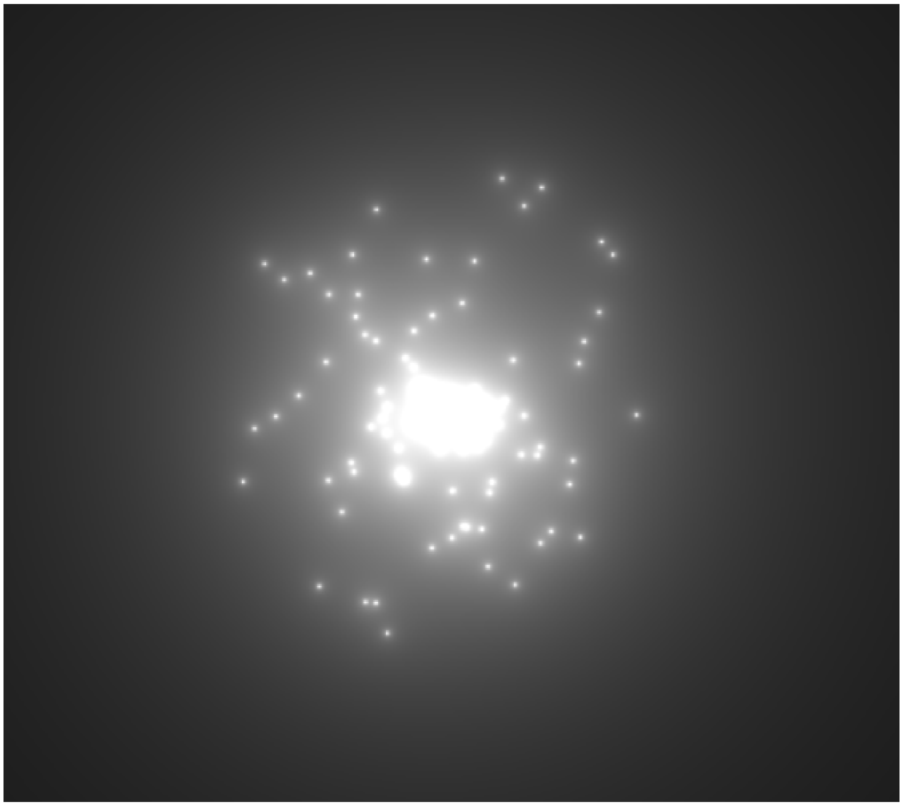
```
// Origin of the particles
uniform vec2 pos;

// 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;

```

(continues on next page)

(continued from previous page)

```
    // 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);
        // 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));
}
```

17.5 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;
```

(continues on next page)

(continued from previous page)

```
  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)
        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));
}
```

17.6 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.)
```

(continues on next page)

(continued from previous page)

```
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;

    // 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);
```

(continues on next page)

(continued from previous page)

```
        alpha += brightness / d;
    }
    // Output to screen
    fragColor = vec4(1.0, 1.0, 1.0, alpha * (1.0 - timeFract));
}
```

CHAPTER EIGHTEEN

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)

        arcade.set_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)

```python

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()
        print(self.width / 2)
        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

```

(continues on next page)

(continued from previous page)

```
            uniform sampler2D texture0;

            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

```
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)
```

(continues on next page)

(continued from previous page)

```python
        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()
```

CHAPTER

NINETEEN

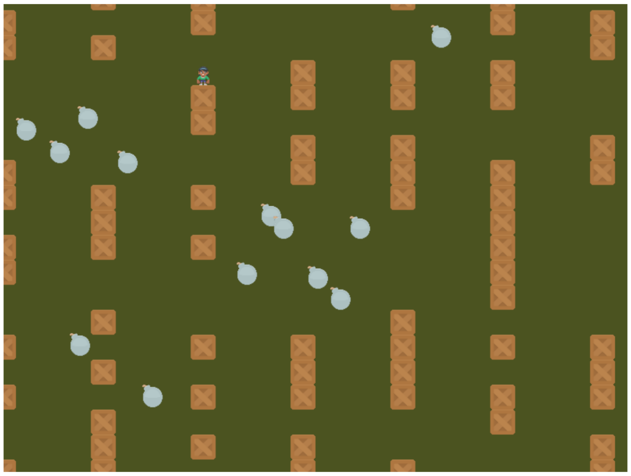
RAY-CASTING SHADOWS

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.

19.1 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.

19.2 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

```
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 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()
        arcade.set_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):
        # Where is the shader file? Must be specified as a path.
        shader_file_path = Path("step_01.glsl")

        # Size of the window
        window_size = self.get_size()

        # Create the shader toy
        self.shadertoy = Shadertoy.create_from_file(window_size, shader_file_path)

        # 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);
```

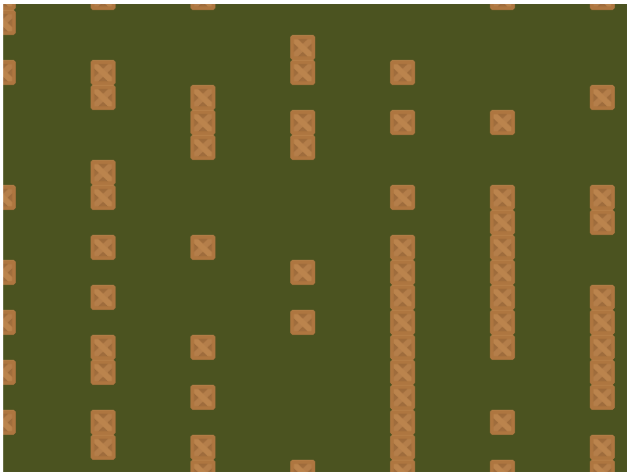
Now that we have our shader, a couple FBOs, and our initial GLSL program, we can flip back to our Python program and update the drawing code to use them:

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

19.3 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.

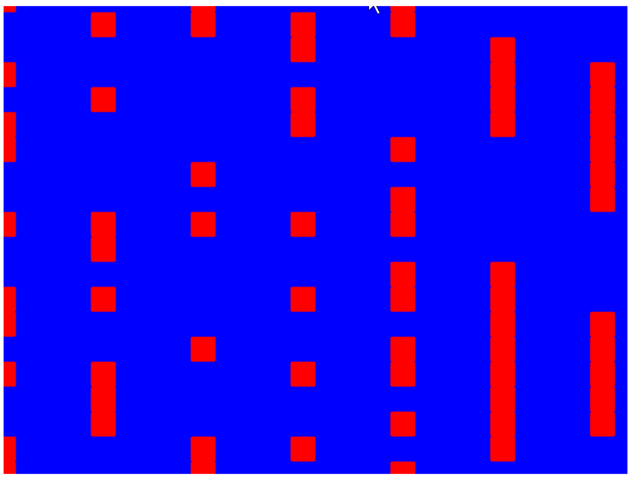
We can modify our shader to get the current texture color and store it in the variable `inColor`. A color has four components, red-green-blue and alpha. If the alpha is above zero, we can output a red color. If the alpha is zero, we output a blue color.

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

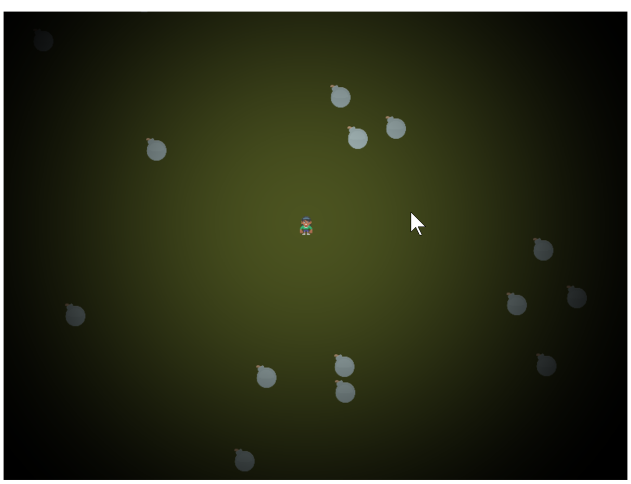
```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec2 normalizedFragCoord = fragCoord/iResolution.xy;
    vec4 inColor = texture(iChannel0, normalizedFragCoord);
    if (inColor.a > 0.0)
        // Set to a red color
        fragColor = vec4(1.0, 0.0, 0.0, 1.0);
    else
        // Set to a blue color
        fragColor = vec4(0.0, 0.0, 1.0, 1.0);
}
```

Giving us a resulting image that looks like:



19.4 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.

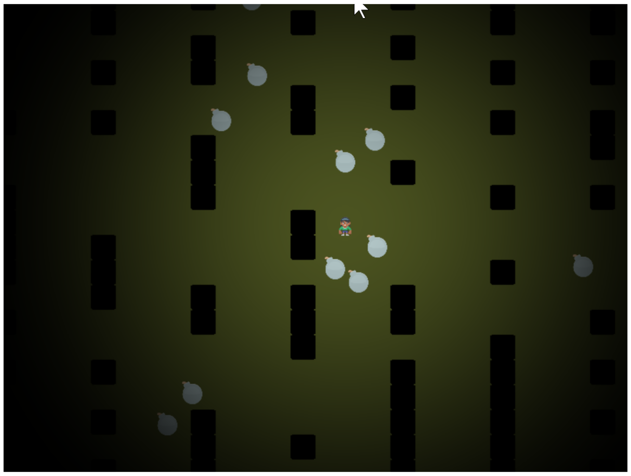
Listing 12: Drawing using the shader

```
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

19.5 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;
}
```

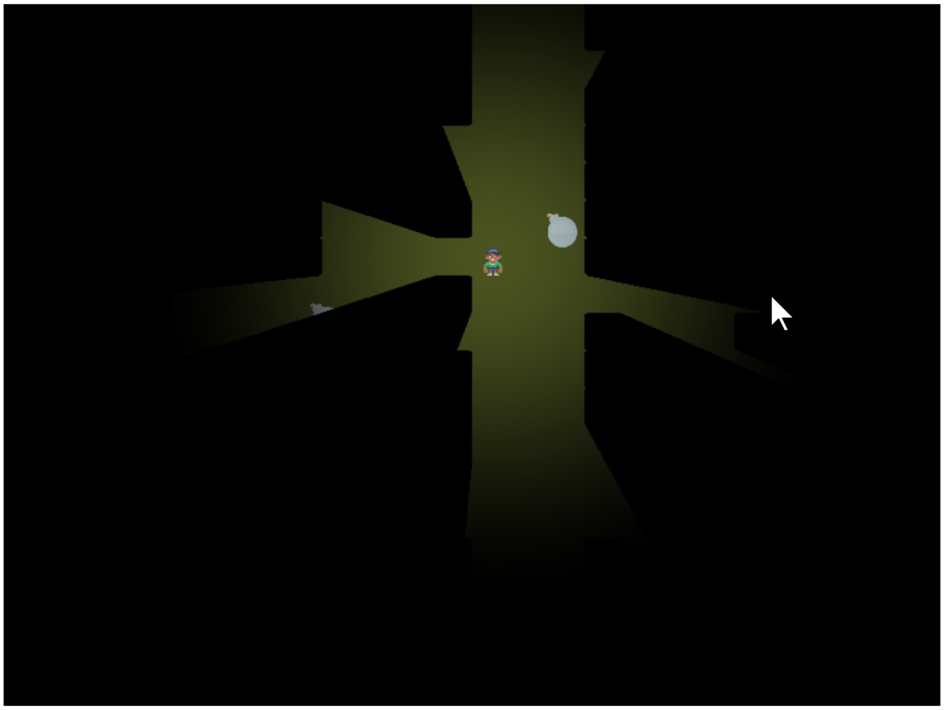
Next, we'll factor in this barrier to our light. So our light amount will be a combination of the distance from the light, and if there's a barrier object on this pixel.

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

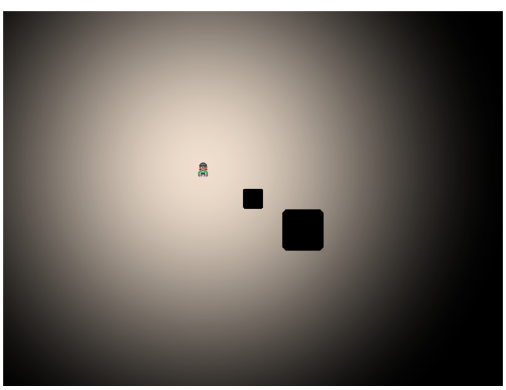
19.6 Step 5: Cast the Shadows



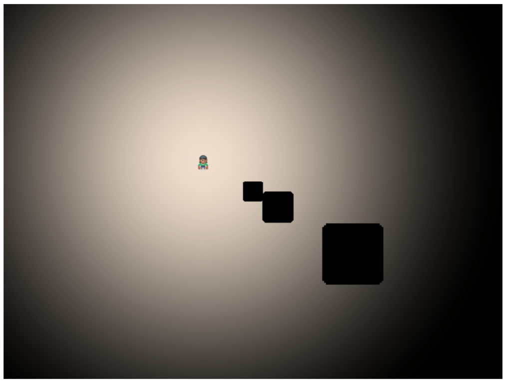
Now it is time to cast the shadows.

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.

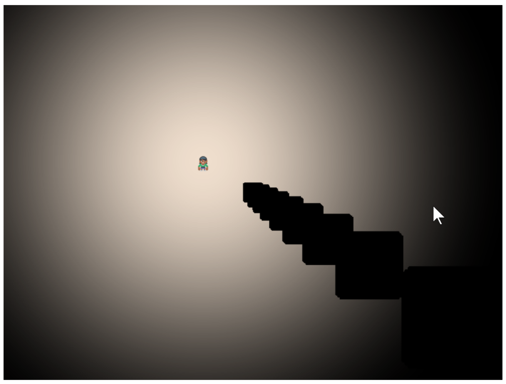
How many times do we sample? If we don't sample enough times, we miss barriers and end up with weird shadows. This first image is if we only sample twice. Once where we are, and once in the middle:



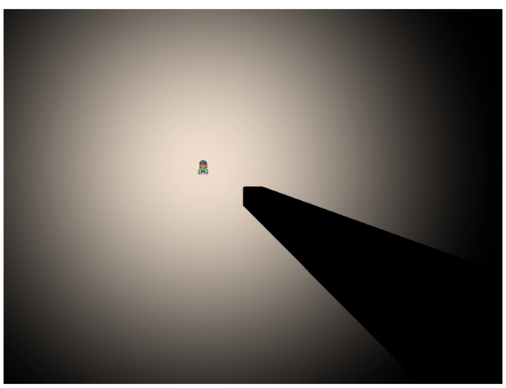
If N is three, we end up with three copies of the shadow:



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.

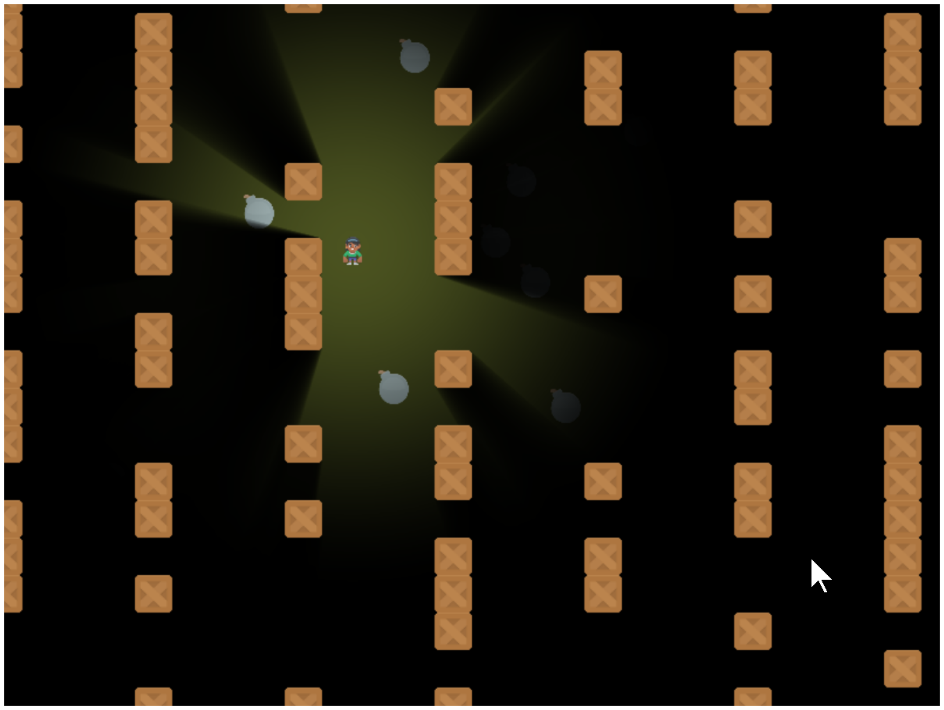


Keep in mind there is a speed trade-off. With 800x600 pixels, we have 480,000 pixels to calculate. If each of those pixels has a loop that does 500 samples, we are sampling 480,000x500 = 240,000 sample per frame, or 14.4 million samples per second, still very do-able with modern graphics cards.

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

19.7 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()
```

(continues on next page)

(continued from previous page)

```
        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

19.8 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

```
class MyGame(arcade.Window):

    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: float, height: float):
        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

19.9 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.Camera(width, height)
        self.camera_gui = arcade.Camera(width, height)

        self.generate_sprites()

        # Our sample GUI text
        self.score_text = arcade.Text("Score: 0", 10, 10, arcade.color.WHITE, 24)

        arcade.set_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 effected by the camera position. Thus we subtract it out.

Listing 20: MyGame.on_draw

```
    def on_draw(self):
        # Use our scrolled camera
```

(continues on next page)

(continued from previous page)

```
        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

```
    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 that new function:

Listing 22: MyGame.scroll_to_player

```
    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 23: MyGame.on_resize

```
    def on_resize(self, width: float, height: float):
        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

19.10 Bibliography

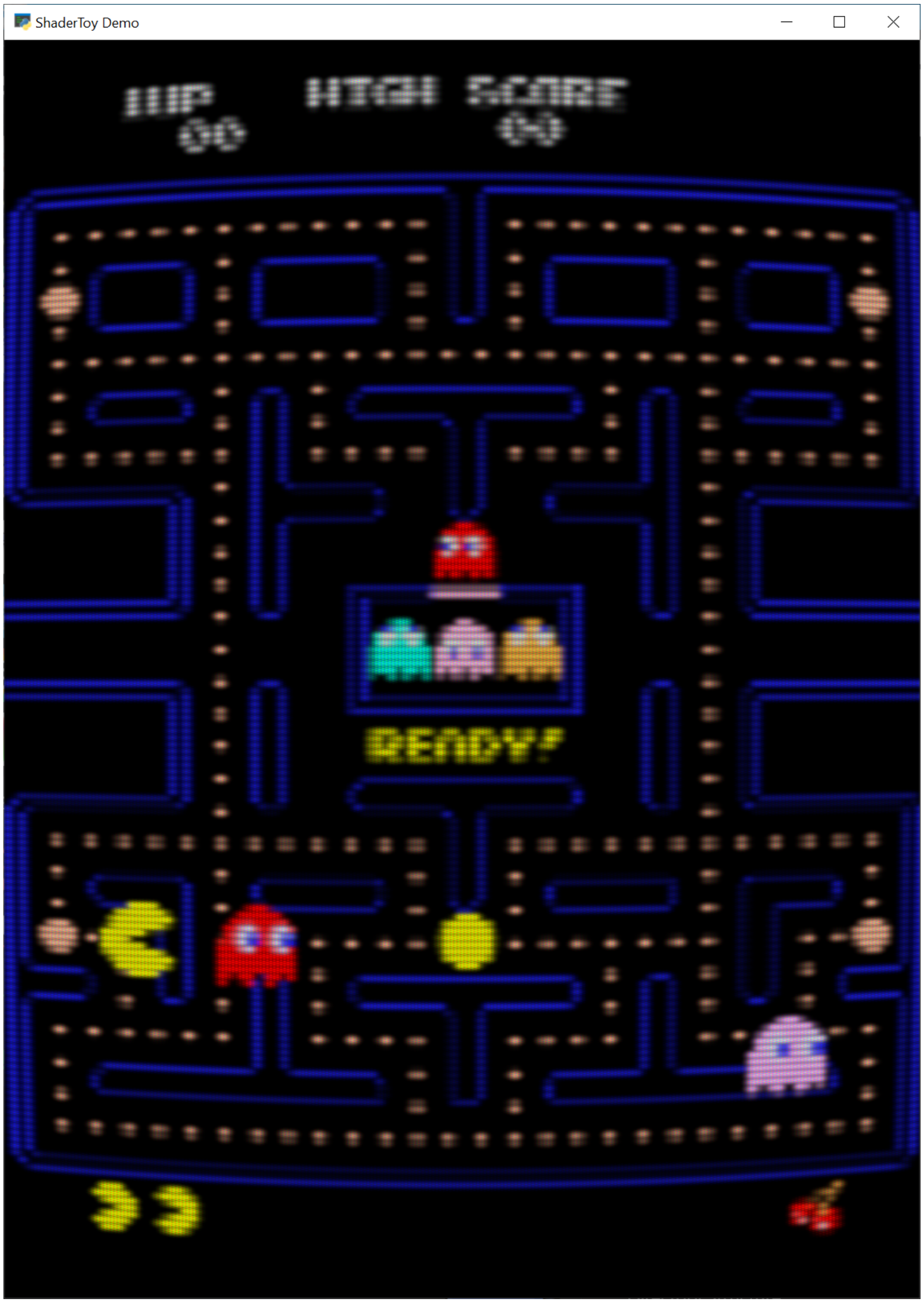
Before I wrote this tutorial I did not know how these shadows were made. I found the sample code Simple 2d Ray-Cast Shadow by jt which allowed me to very slowly figure out how to cast shadows.

CHAPTER

TWENTY

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

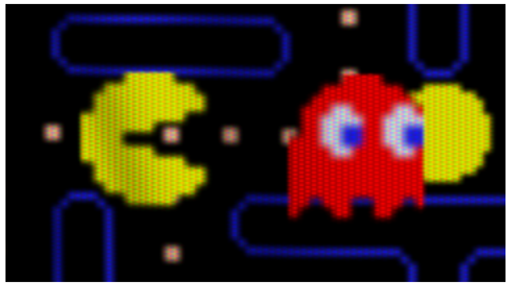


Fig. 1: resolution_down_scale = 1

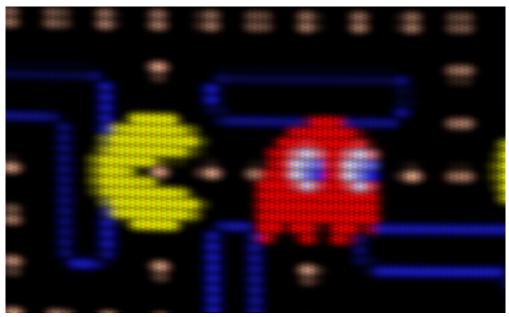


Fig. 2: resolution_down_scale = 6

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()
```

20.1 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

```
from pathlib import Path
import arcade
from arcade.experimental.crt_filter import CRTFilter
```

(continues on next page)

(continued from previous page)

```
from pyglet.math import Vec2


# Do the math to figure out our screen dimensions
SCREEN_WIDTH = 800
SCREEN_HEIGHT = 1100
SCREEN_TITLE = "ShaderToy Demo"
RESOURCE_DIR = Path(__file__).parent


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(RESOURCE_DIR / "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(RESOURCE_DIR / "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(RESOURCE_DIR / "pac_man_sprite_sheet.png",
                                  scale=5, image_x=4, image_y=81, image_width=13, image_
↪height=15)
        my_sprite.change_x = -1
        self.sprite_list.append(my_sprite)
        my_sprite.center_x = 800
        my_sprite.center_y = 200

        my_sprite = arcade.AnimatedTimeBasedSprite()
```

(continues on next page)

(continued from previous page)

```
        texture = arcade.load_texture(RESOURCE_DIR / "pac_man_sprite_sheet.png", x=4,␣
↪y=1, width=13, height=15)
        frame = arcade.AnimationKeyframe(tile_id=0,
                                         duration=150,
                                         texture=texture)
        my_sprite.frames.append(frame)
        texture = arcade.load_texture(RESOURCE_DIR / "pac_man_sprite_sheet.png", x=20,␣
↪y=1, width=13, height=15)
        frame = arcade.AnimationKeyframe(tile_id=1,
                                         duration=150,
                                         texture=texture)
        my_sprite.frames.append(frame)

        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

    def on_key_press(self, key, mod):
        if key == arcade.key.SPACE:
            self.filter_on = not self.filter_on
```

(continues on next page)

(continued from previous page)

```
if __name__ == "__main__":
    MyGame(SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_TITLE)
    arcade.run()
```

CHAPTER TWENTYONE

COMPUTE SHADER TUTORIAL

Using the compute shader, you can use the GPU to perform calculations thousands of times faster than just by using the CPU.

In this example, we will simulate a star field using an 'N-Body simulation'. Each star is effected by each other star's gravity. 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, this glues everything together.
- The visualization shaders, which let us see the data.
- The compute shader, which moves everything.

21.1 Visualization Shaders

There are multiple visualization shaders, which operate in this order:

The Python program creates a **shader storage buffer object** (SSBO) of floating point numbers. This buffer has the x, y, z and radius of each star stored in `in_vertex`. It also stores the color in `in_color`.

The **vertex shader** doesn't do much more than separate out the radius variable from the group of floats used to store position.

Listing 1: shaders/vertex_shader.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;
```

(continues on next page)

(continued from previous page)

```
    vertex_color = in_color;
}
```

The **geometry shader** converts the single point (which we can't render) to a square, which we can render. It changes the one point, to four points of a quad.

Listing 2: shaders/geometry_shader.glsl

```
#version 330

layout (points) in;
layout (triangle_strip, max_vertices = 4) out;

// Use arcade's global projection UBO
uniform Projection {
    uniform mat4 matrix;
} proj;

in vec2 vertex_pos[];
in vec4 vertex_color[];
in float vertex_radius[];

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();

    EndPrimitive();
}
```

The **fragment shader** runs for each pixel. It produces the soft glow effect of the star, and rounds off the quad into a circle.

Listing 3: shaders/fragment_shader.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);
}
```

21.2 Compute Shaders

This program runs two buffers. We have an **input buffer**, with all our current data. We perform calculations on that data and write to the **output buffer**. We then swap those buffers for the next frame, where we use the output of the previous frame as the input to the next frame.

Listing 4: shaders/compute_shader.glsl

```
#version 430

// Set up our compute groups
layout(local_size_x=COMPUTE_SIZE_X, local_size_y=COMPUTE_SIZE_Y) in;

// Input uniforms go here if you need them.
// Some examples:
//uniform vec2 screen_size;
//uniform vec2 force;
//uniform float frame_time;

// Structure of the ball data
struct Ball
{
    vec4 pos;
```

(continues on next page)

(continued from previous page)

```
    vec4 vel;
    vec4 color;
};

// Input buffer
layout(std430, binding=0) buffer balls_in
{
    Ball balls[];
} In;

// Output buffer
layout(std430, binding=1) buffer balls_out
{
    Ball balls[];
} Out;

void main()
{
    int curBallIndex = int(gl_GlobalInvocationID);

    Ball in_ball = In.balls[curBallIndex];

    vec4 p = in_ball.pos.xyzw;
    vec4 v = in_ball.vel.xyzw;

    // Move the ball 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.balls.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.balls[i].pos.xyzw.xy, p.xy);
        float distanceSquared = dist * dist;

        // If stars get too close the fling into never-never land.
        // So use a minimum distance
        float minDistance = 0.02;
        float gravityStrength = 0.3;
        float simulationSpeed = 0.002;
        float force = min(minDistance, gravityStrength / distanceSquared) * -
↪simulationSpeed;

        vec2 diff = p.xy - In.balls[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;
```

(continues on next page)

(continued from previous page)

```
    }


    Ball out_ball;
    out_ball.pos.xyzw = p.xyzw;
    out_ball.vel.xyzw = v.xyzw;

    vec4 c = in_ball.color.xyzw;
    out_ball.color.xyzw = c.xyzw;

    Out.balls[curBallIndex] = out_ball;
}
```

21.3 Python Program

Read through the code here, I've tried hard to explain all the parts in the comments.

Listing 5: main.py

```python
"""
Compute shader with buffers
"""
import random
from array import array

import arcade
from arcade.gl import BufferDescription

# Window dimensions
WINDOW_WIDTH = 2300
WINDOW_HEIGHT = 1300

# Size of performance graphs
GRAPH_WIDTH = 200
GRAPH_HEIGHT = 120
GRAPH_MARGIN = 5


class MyWindow(arcade.Window):

    def __init__(self):
        # Call parent constructor
        # Ask for OpenGL 4.3 context, as we need that for compute shader support.
        super().__init__(WINDOW_WIDTH, WINDOW_HEIGHT,
                         "Compute Shader",
                         gl_version=(4, 3),
                         resizable=True)
        self.center_window()

        # --- Class instance variables
```

(continues on next page)

(continued from previous page)

```
        # Number of balls to move
        self.num_balls = 40000

        # This has something to do with how we break the calculations up
        # and parallelize them.
        self.group_x = 256
        self.group_y = 1

        # --- Create buffers

        # Format of the 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"
        # Generate the initial data that we will put in buffer 1.
        initial_data = self.gen_initial_data()

        # Create data buffers for the compute shader
        # We ping-pong render between these two buffers
        # ssbo = shader storage buffer object
        self.ssbo_1 = self.ctx.buffer(data=array('f', initial_data))
        self.ssbo_2 = self.ctx.buffer(reserve=self.ssbo_1.size)

        # Attribute variable names for the vertex shader
        attributes = ["in_vertex", "in_color"]
        self.vao_1 = self.ctx.geometry(
            [BufferDescription(self.ssbo_1, buffer_format, attributes)],
            mode=self.ctx.POINTS,
        )
        self.vao_2 = self.ctx.geometry(
            [BufferDescription(self.ssbo_2, buffer_format, attributes)],
            mode=self.ctx.POINTS,
        )

        # --- Create shaders

        # Load in the shader source code
        file = open("shaders/compute_shader.glsl")
        compute_shader_source = file.read()
        file = open("shaders/vertex_shader.glsl")
        vertex_shader_source = file.read()
        file = open("shaders/fragment_shader.glsl")
        fragment_shader_source = file.read()
        file = open("shaders/geometry_shader.glsl")
        geometry_shader_source = file.read()

        # Create our compute shader.
        # Search/replace to set up our compute groups
        compute_shader_source = compute_shader_source.replace("COMPUTE_SIZE_X",
                                                              str(self.group_x))
```

(continues on next page)

(continued from previous page)

```
        compute_shader_source = compute_shader_source.replace("COMPUTE_SIZE_Y",
                                                              str(self.group_y))
        self.compute_shader = self.ctx.compute_shader(source=compute_shader_source)

        # Program for visualizing the balls
        self.program = self.ctx.program(
            vertex_shader=vertex_shader_source,
            geometry_shader=geometry_shader_source,
            fragment_shader=fragment_shader_source,
        )

        # --- Create 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.center_x = GRAPH_WIDTH / 2
        graph.center_y = 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_1.bind_to_storage_buffer(binding=0)
        self.ssbo_2.bind_to_storage_buffer(binding=1)

        # Set input variables for compute shader
        # These are examples, although this example doesn't use them
        # self.compute_shader["screen_size"] = self.get_size()
        # self.compute_shader["force"] = force
        # self.compute_shader["frame_time"] = self.run_time

        # Run compute shader
        self.compute_shader.run(group_x=self.group_x, group_y=self.group_y)

        # Draw the balls
        self.vao_2.render(self.program)

        # Swap the buffers around (we are ping-ping rendering between two buffers)
        self.ssbo_1, self.ssbo_2 = self.ssbo_2, self.ssbo_1
        # Swap what geometry we draw
        self.vao_1, self.vao_2 = self.vao_2, self.vao_1

```

(continues on next page)

(continued from previous page)

```
        # Draw the graphs
        self.perf_graph_list.draw()

    def gen_initial_data(self):
        for i in range(self.num_balls):
            # Position/radius
            yield random.randrange(0, self.width)
            yield random.randrange(0, self.height)
            yield 0.0  # z (padding)
            yield 6.0

            # Velocity
            yield 0.0
            yield 0.0
            yield 0.0  # vz (padding)
            yield 0.0  # vw (padding)

            # Color
            yield 1.0  # r
            yield 1.0  # g
            yield 1.0  # b
            yield 1.0  # a


app = MyWindow()
arcade.run()
```

An expanded version of this, with support for 3D, is available at: https://github.com/pvcraven/n-body

CHAPTER

TWENTYTWO

GUI

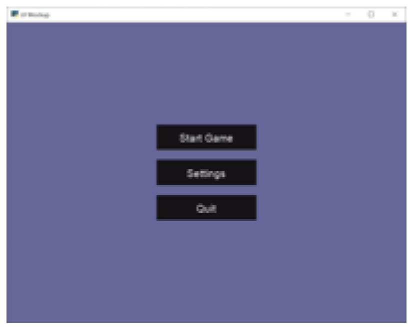


Fig. 1: gui_flat_button

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.

22.1 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 *UIEvents*.

Classes of Arcades GUI code are prefixed with UI- to make them easy to identify and search for in autocompletion.

22.1.1 UIWidget

`UIWidget` are the core of Arcades GUI. A widget represents the behaviour and graphical representation of any element (like Buttons or Text)

A `UIWidget` has following properties

rect x and y coordinates (bottom left of the widget), width and height

children Child widgets, rendered within this widget A `UIWidget` will not move or resize its children, use a `UILayout` instead.

size_hint tuple of two floats, defines how much of the parents space it would like to occupy (range: 0.0-1.0). For maximal vertical and horizontal expansion, define *size_hint* of 1 for the axis.

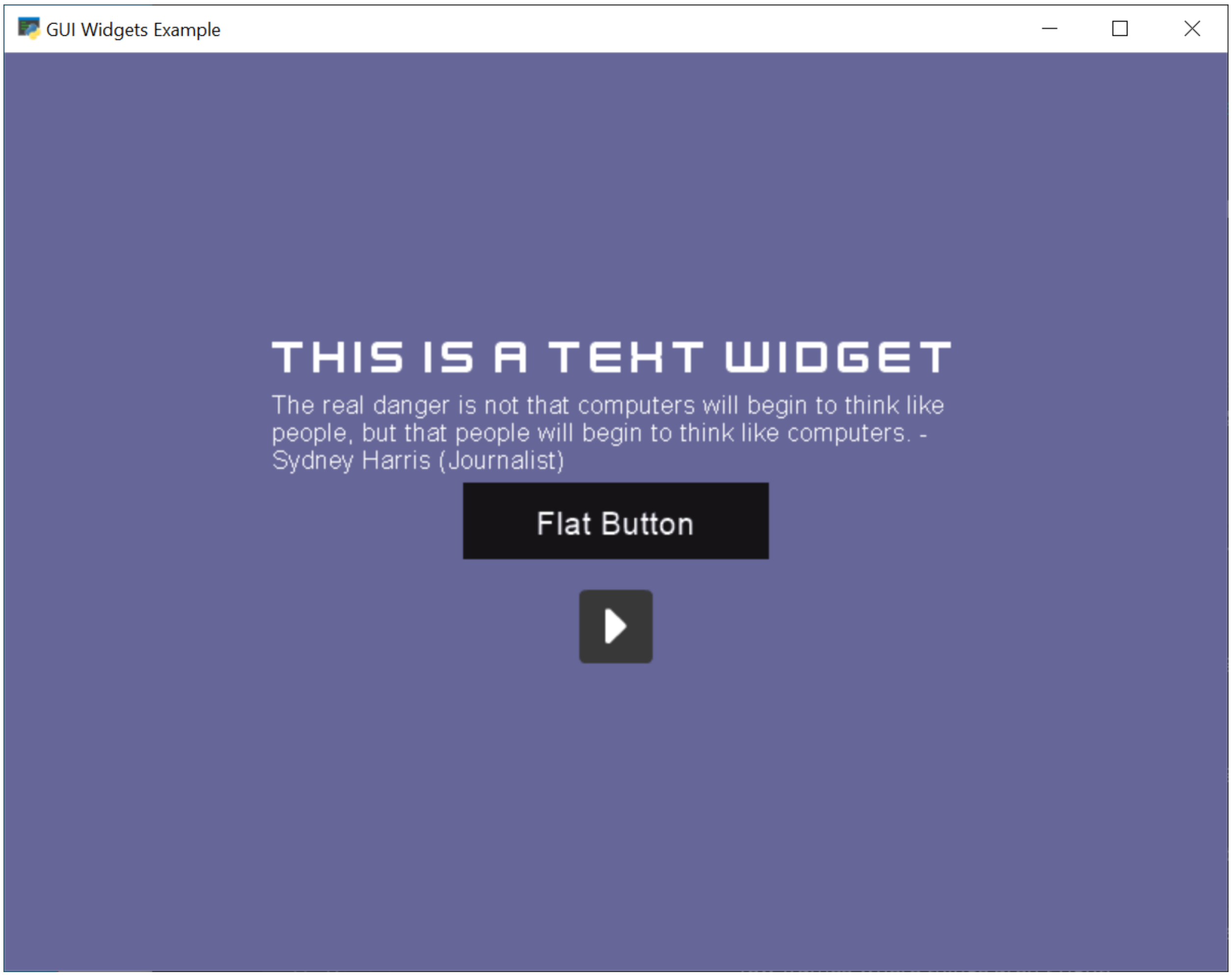


Fig. 2: gui_widgets

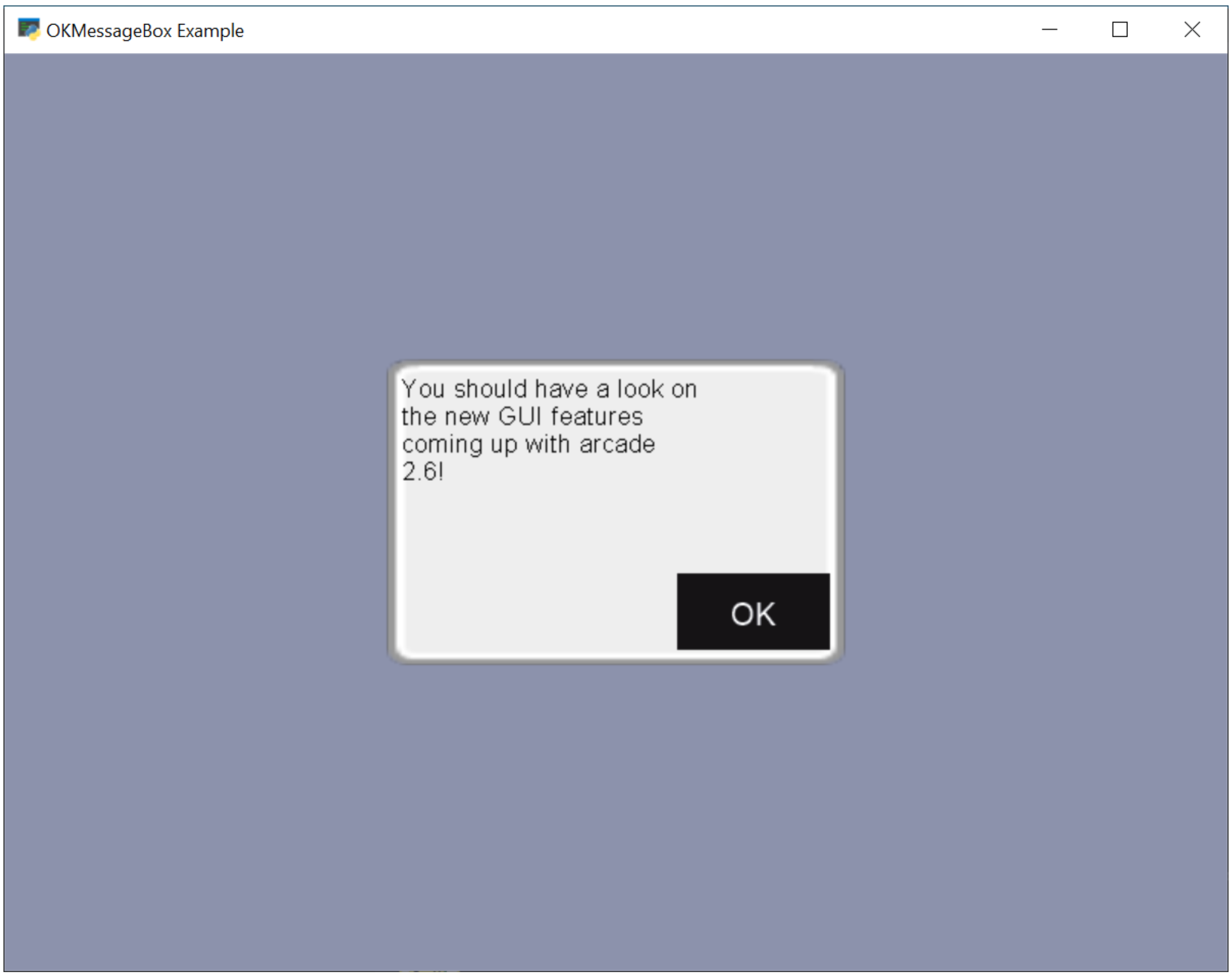


Fig. 3: gui_ok_messagebox

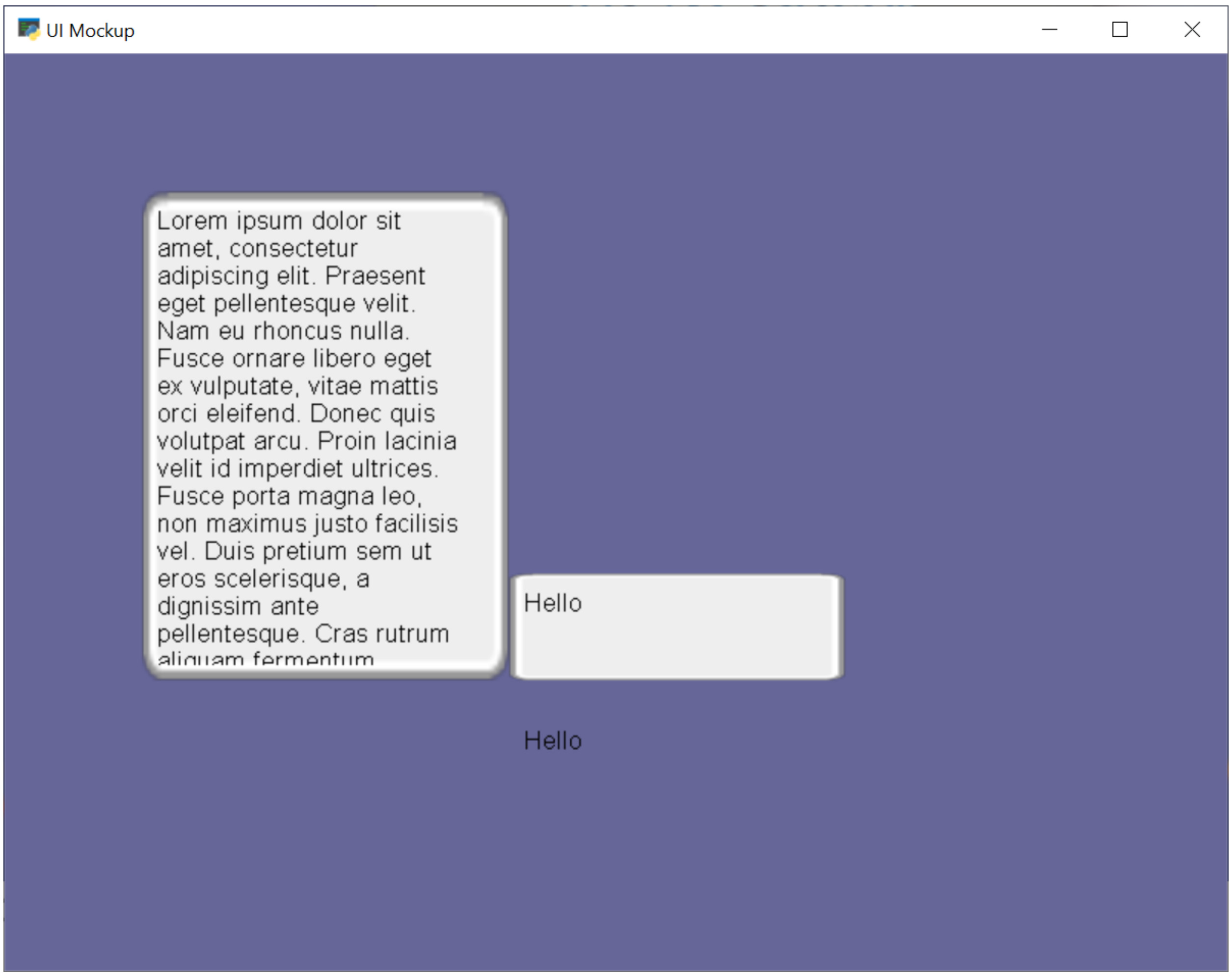


Fig. 4: gui_scrollable_text

size_hint_min tuple of two ints, defines minimal size of the widget. If set, changing the size of a widget to a lower values will use this size instead.

size_hint_max tuple of two ints, defines maximum size of the widget. If set, changing the size of a widget to a higher values will use this size instead.

size_hint, *size_hint_min*, and *size_hint_max* are values that are additional information of a widget, but do not effect the widget on its own. `UILayout` may use these information to place or resize a widget.

Rendering

`UIWidget.do_render()` is called recursively if rendering was requested via `UIWidget.trigger_render()`. In case widgets have to request their parents to render use `UIWidget.trigger_full_render()`

The widget has to draw itself and child widgets within `UIWidget.do_render()`. Due to the deferred functionality render does not have to check any dirty variables, as long as state changes use the trigger function.

For widgets, that might have transparent areas, they have to request a full rendering.

Enforced rendering of the whole GUI might be very expensive!

22.1.2 UILayout and UIWrapper

`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*.

`UIWrapper` are widgets that are used to wrap a single child widget to apply additional effects like borders or space around.

Algorithm (WIP, not fully implemented)

`UIManager` triggers the layout and render process right before the actual frame draw. This opens the possibility, to adjust to multiple changes only ones.

Executed steps within `UIBoxLayout`:

1. **`UIBoxLayout.do_layout()`**
    1. collect current size, size_hint, size_hint_min/max of children
    2. calculate the new position and sizes
    3. set position and size of children
2. recursive call *do_layout* on child layouts (done after `UIBoxLayout.do_layout()`)

```
┌─────────┐          ┌────────┐                    ┌────────┐
│UIManager│          │UILayout│                    │children│
└────┬────┘          └───┬────┘                    └───┬────┘
     │   do_layout()     │                             │
     │──────────────────>│┌┐                           │
     │                   ││                            │
     │                   ││                            │
     │                                                 │
     │      place children   │                         │
     │      ──────────────use size, size_hint, ...     │
     │                   │ │<───────────────────────────
```

(continues on next page)

(continued from previous page)

```
     │                 ││                              │
     │                 ││          set size and pos    │
     │                 ││  ─────────────────────────────>
     │                                                 
     │                  ││                              │
     │                  ││                              │
     │                                                  
     │           LOOP  │  sub layouts                  │
     │         ───────    │ │                          │
     │                 ││           do_layout()        │
     │                 ││  ─────────────────────────────>
     │                                                  
     │                  └                              │
     │                   │                              │
     │<─ ─ ─ ─ ─ ─ ─ ─ ─ │                              │
 ────────          ───────                     ───────
│UIManager│          │UILayout│                    │children│
└────────          └───────                     └───────
```

22.1.3 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.

22.1.4 Constructs

Constructs are predefined structures of widgets and layouts like a message box or (not yet available) file dialogues.

22.1.5 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, fits content
  - `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`.

22.1.6 UIEvents

UIEvents are fully typed dataclasses, which provide information about a event effecting the UI. Events are passed top down to every `UIWidget` by the UIManager.

General pyglet window events are converted by the UIManager into UIEvents and passed via dispatch_event to the `on_event` callbacks.

Widget specific UIEvents like UIOnClick are dispatched via "on_event" and are then dispatched as specific event types (like 'on_click')

- `UIEvent` - Base class for all events
- **`UIMouseEvent` - Base class for mouse related event**
  - `UIMouseMovementEvent` - Mouse moves
  - `UIMousePressEvent` - Mouse button pressed
  - `UIMouseDragEvent` - Mouse pressed and moved (drag)
  - `UIMouseReleaseEvent` - Mouse button released
  - `UIMouseScrollEvent` - Mouse scolls
- `UITextEvent` - Text input from user
- `UITextMotionEvent` - Text motion events like arrows
- `UITextMotionSelectEvent` - Text motion events for selection
- `UIOnClickEvent` - Click event of `UIInteractiveWidget` class
- `UIOnChangeEvent` - A value of a `UIWidget` has changed
- `UIOnUpdateEvent` - arcade.Window *on_update* callback

22.1.7 Different Event Systems

The GUI uses different event systems, dependent on the required flow. A game developer should mostly interact with UIEvents which are dispatched from specific UIWidgets like `on_click` of a button.

In rare cases a developer might implement some UIWidgets or wants to modify the existing GUI behavior. In those cases a developer might register own Pyglet event types on UIWidgets or overwrite the `UIWidget.on_event` method.

Pyglet Window Events

Received by UIManager, dispatched via `UIWidget.dispatch_event("on_event", UIEvent(...))`. Window Events are wrapped into subclasses of UIEvent.

Pyglet EventDispatcher - UIWidget

UIWidgets implement Pyglets EventDispatcher and register an `on_event` event type. `UIWidget.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")`).

UIEvents

UIEvents are typed representations of events that are passed within the GUI. UIWidgets might define their own UIEvents.

_Property

`_Property` is an internal, experimental, pure-Python implementation of Kivy Properties. They are used to detect attribute changes of UIWidgets and trigger rendering. They should only be used in arcade internal code.

22.2 GUI Style

> `arcade.experimental.uistyle` is an experimental component, which might change in upcoming versions.

`arcade.experimental.uistyle` provides style dicts, which are used within `UIWidget` to provide the colors for default appearance.

22.2.1 Style Parameters

`UIWidget` load style parameters from a dict like object, which can be passed as `UIWidget.style`.

Style Parameters

Following parameters are used within multiple `UIWidget`. Style parameters are prefixed with the `UIWidget` state (`normal`, `hovered` and `pressed`)

<state>_font_size Font size of any text within the `UIWidget`

<state>_font_name Font of any text within the `UIWidget`

<state>_font_color Color of any text within the `UIWidget`

<state>_bg Background color, also used as the primary color within an `UIWidget`

<state>_border Color of `UIWidget` border

<state>_border_width Width of `UIWidget` border in pixel

<state>_filled_bar Color used within bars like slider to indicate fill state

<state>_unfilled_bar Color used within bars like slider for unfilled background

22.2.2 UIWidget Style

UISlider

- `<state>_filled_bar`
- `<state>_unfilled_bar`
- `<state>_bg` - color of cursor
- `<state>_border` - outline of cursor
- `<state>_border_width`

UIFlatButton

- `<state>_font_name`
- `<state>_font_size`
- `<state>_font_color`
- `<state>_bg`
- `<state>_border`
- `<state>_border_width`

22.3 Troubleshooting & Hints

22.3.1 `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

TWENTYTHREE

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.

23.1 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
```

23.2 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.

23.3 Examples

There are two recommended approaches: *Simple headless mode* and *Headless mode while extending the Arcade Window*.

23.3.1 Simple headless mode

For simpler applications we don't need to subclass the window.

```
# 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.

23.3.2 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.

```
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.

23.4 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
```

23.5 Issues?

If you run into issues or have questions please create an issue on github or join our discord server.

CHAPTER

TWENTYFOUR

VERTICAL SYNCHRONIZATION

24.1 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:

```
# 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.

24.2 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.

24.3 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

TWENTYFIVE

TEXTURES

25.1 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.

25.2 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.

25.3 Texture Cache

Arcade is caching texture instances based on the `name` attribute to significantly speed up loading times.

```
# 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()`.

25.4 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.

```
# 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

TWENTYSIX

TEXTURE ATLAS

26.1 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 crated. 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.

26.2 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.

26.3 Resize

Atlases will resize automatically when full. It will also try to pack the textures better by sorting them by their height.

26.4 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).

26.5 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)
```

26.6 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.

26.7 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.

26.8 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.

26.9 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.

26.10 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

TWENTYSEVEN

OPENGL NOTES

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.

CHAPTER

TWENTYEIGHT

ARCADE PERFORMANCE INFORMATION

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:

```
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

DIVERSE CODERS

If you are female, trans, non-binary, POC there are a lot of coders like you! But sometimes it can be hard to find them.

- Python Software Foundation Diversity Statement
- PyCon 2022 Diversity Statement
- Arcade Library's Code of Conduct

Check out the following organizations.

- PyLadies
- Django Girls
- Girls Who Code
- Trans Game Dev

The US PyCon works hard to be open to create an inclusive conference. I highly recommend it to people who are interested in going. They have special sessions on diversity, easily identified staff trained in incident response, ribbons people put no their name tag for preferred pronouns, and more. PyLadies holds charity auctions at the US PyCon every year, raising over $20,000 every year to help support and promote coding amongst women.

If you are interested in getting started coding, or contributing to the Python community, reach out to me or any of the organizations above.

CHAPTER

THIRTY

SOCIAL MEDIA

- Discord - The most active spot
- Reddit /r/pythonarcade The next most active spot
- Twitter @ArcadeLibrary Good for announcements
- Instagram @PythonArcadeLibrary
- Facebook

CHAPTER

THIRTYONE

DEVELOPMENT INFORMATION

31.1 Release Notes

Keep up-to-date with the latest changes to the Arcade library by the release notes.

31.1.1 Version 2.6.17

Released 2022-Dec-30

- Bump Pillow to 9.3.0
- Bump PyMunk to 6.4.0
- Add explicit compatability tag for 3.11
- Drop 3.7 as part of the test suite

31.1.2 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.

31.1.3 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

31.1.4 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.

31.1.5 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.

31.1.6 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 Tutorial - 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

31.1.7 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: *Dividing a View Into Sections*
- New Example Code:
  – Add parallax example: 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.

31.1.8 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.

31.1.9 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.

31.1.10 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.

31.1.11 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.

31.1.12 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.

31.1.13 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.

31.1.14 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.

31.1.15 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

31.1.16 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

31.1.17 Version 2.6.1

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

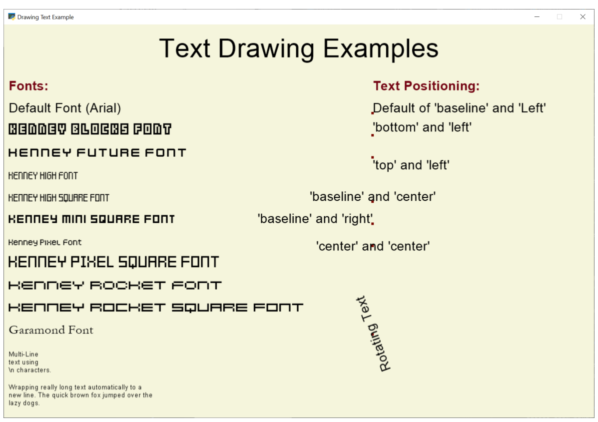
31.1.18 Version 2.6.0

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.

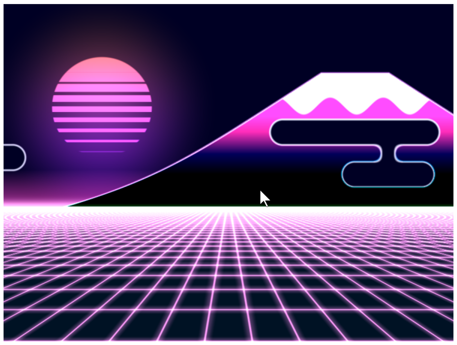
- Tiled Map Editor support has been overhauled.
  - Arcade now uses the .json file format for maps created by the Tiled Map Editor rather than the TMX format. Tile sets and other supporting files need to all be saved in .json format. The XML based formats are no longer supported by Arcade.
  - Arcade 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.

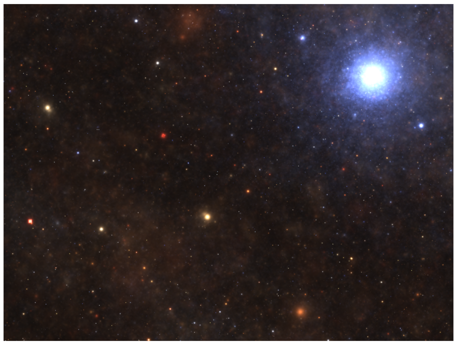
- Texture atlases have been introduced, texture management has been improved.
  - A sprite list will create and use its own texture atlas.
  - This introduces a new `arcade.TextureAtlas` class that is used internally by SpriteList.
  - Sprites with new textures can be added to a sprite list without the 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.

- 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.

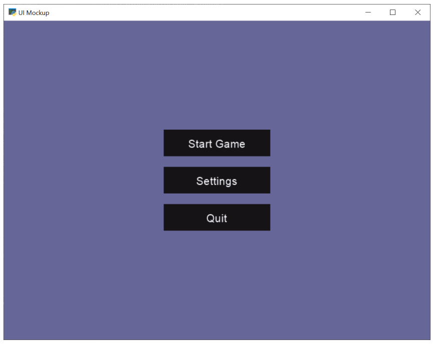


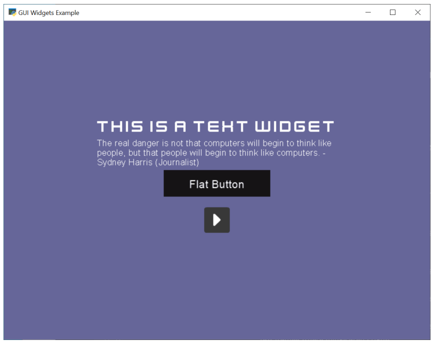
- SpriteList optimizations.
  - Sprites now draw even faster than before. On an Intel i7 with nVidia 980 Ti graphics card, 8,000+ moving sprites can be drawn while maintaining 60 FPS. The same machine can only do 2,000 sprites with Pygame before FPS drops.
- Shadertoy support.
  - Shadertoy.com is a 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 Tutorial - Glow* and Asteroids.





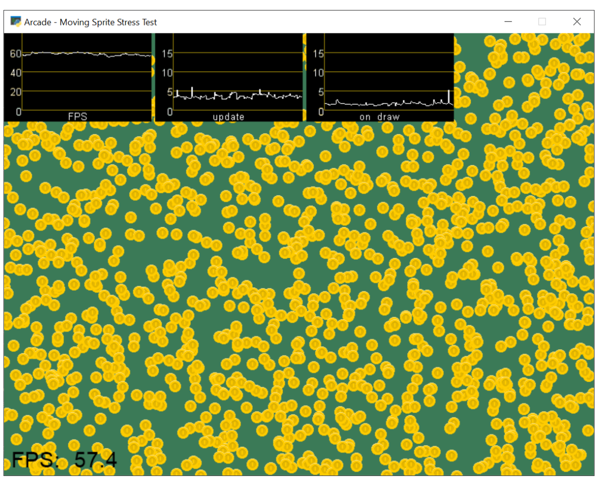
- Reworked GUI



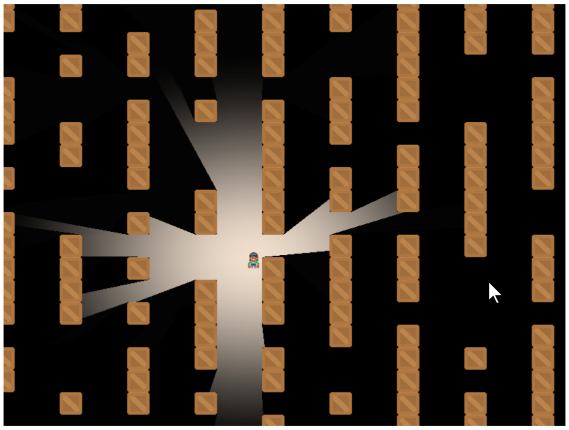


- UIElements are replaced by UIWidgets
- Option to relative pin widgets on screen to center or border (supports resizing)
- Widgets can be placed on top of each other
- Overlapping widgets properly handle mouse interaction
- Fully typed event classes
- Events contain source widget
- ScrollableText widgets (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 *Quick 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 Tutorial - 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.

31.1.19 Version 2.5.7

Released on 2021-May-25

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

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.

31.1.20 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

31.1.21 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

31.1.22 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

31.1.23 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

31.1.24 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

31.1.25 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

31.1.26 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.

31.1.27 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

31.1.28 Version 2.4.2

Version 2.4.2 was released 2020-09-08

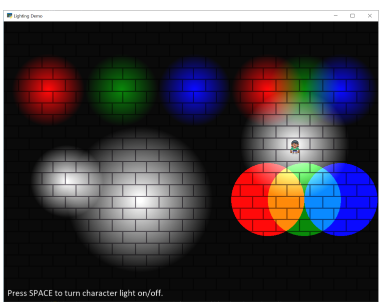
- Enhancement: `draw_hit_boxes` new method in `SpriteList`.
- Enhancement: `draw_points` now significantly faster
- Added UIToggle, on/off switch
- Add example showing how to do GPU transformations with the mouse
- Create buttons with default size/position so size can be set after creation.
- Allow checking if a sound is done playing Issue 728

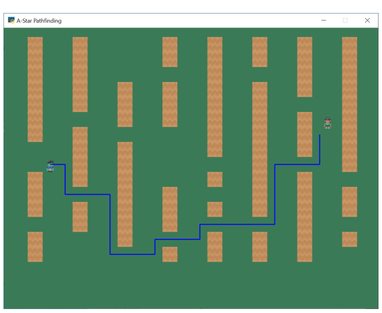
- Add an early camera mock-up
- Add `finish` method to `arcade.gl.context`.
- New example arcade.experimental.examples.3d_cube (experimental)
- New example arcade.examples.camera_example
- Improved UIManager.unregister_handlers(), improves multi view setup
- Update `preload_textures` method of `SpriteList` 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

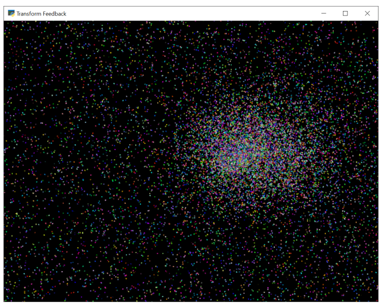
31.1.29 Version 2.4

Arcade 2.4.1 was released 2020-07-13.

Arcade version 2.4 is a major enhancement release to Arcade.







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: bloom_defender.
  - 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.

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.

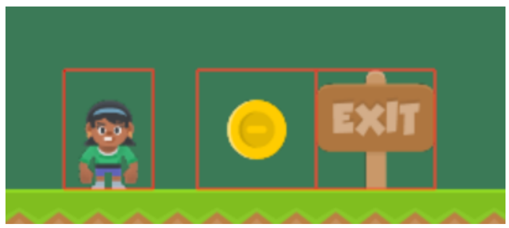


Fig. 1: hit_box_algorithm = "None"

Fig. 2: hit_box_algorithm = "Simple"

Fig. 3: hit_box_algorithm = "Detailed"

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.

Version 2.4 New Documentation

- New Tutorial: *Pymunk Platformer*
- New Tutorial: *Using Views for Start/End Screens*
- New Tutorial: *Solitaire Tutorial*
- 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

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.

Special Thanks

Special thanks to Einar Forselv and Maic Siemering for their significant work in helping put this release together.

31.1.30 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

31.1.31 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.

31.1.32 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

31.1.33 Version 2.3.12

Release Date: Apr-8-2020

- Enhancement: Support more textures in a SpriteList Issue 332

31.1.34 Version 2.3.11

Release Date: Apr-5-2020

- Bug Fix: Fix procedural_caves_bsp.py
- Bug Fix: Improve Windows install docs Issue 623

31.1.35 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

31.1.36 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

31.1.37 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.

31.1.38 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

31.1.39 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

31.1.40 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

31.1.41 Version 2.3.4

Release Date: Feb-08-2020

- Bug Fix: Sprites weren't appearing Issue 585

31.1.42 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

31.1.43 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

31.1.44 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.

31.1.45 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.)

31.1.46 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

31.1.47 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

31.1.48 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

31.1.49 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*.

31.1.50 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

31.1.51 Version 2.2.4

Release Date: Jan-15-2020

- Enhancement: Moving sprites now 20% more efficient.

31.1.52 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.

31.1.53 Version 2.2.2

Release Date: Jan-09-2020

- Bug fix: Arcade assumes tiles in tileset are same sized Issue 550

31.1.54 Version 2.2.1

Release Date: Dec-22-2019

- Bug fix: Resource folder not included in distribution Issue 541

31.1.55 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.

31.1.56 Version 2.1.7

- Enhancement: Tile set support. Issue 511
- Bug fix, search file tile images relative to tile map. Issue 480

31.1.57 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

31.1.58 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

31.1.59 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

31.1.60 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

31.1.61 Version 2.1.2

- Fix: Linked to wrong version of Pyglet Issue 467

31.1.62 Version 2.1.1

- Added pytiled-parser as a dependency in setup.py

31.1.63 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/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.

31.1.64 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

31.1.65 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

31.1.66 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

31.1.67 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

31.1.68 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

31.1.69 Version 2.0.4

- Fix for drawing with a border width of 1 Issue 352

31.1.70 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.

31.1.71 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

31.1.72 Version 2.0.1

- Turn on multi-sampling so lines could be anti-aliased Issue 325

31.1.73 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.

31.1.74 Version 1.3.7

Released 2018-10-28

- Fix for Issue 275 where sprites can get blurry.

31.1.75 Version 1.3.6

Released 2018-10-10

- Bux fix for spatial hashing
- Implement commands for getting a pixel, and image from screen

31.1.76 Version 1.3.5

Released 08-23-2018

Bug fixes for spatial hashing and sound.

31.1.77 Version 1.3.4

Released 28-May-2018

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.

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

Documentation

- Issue 217: Fix typo in doc string
- Issue 198: Add example showing start of isometric tile support

31.1.78 Version 1.3.3

Released 2018-May-05

New Features

- Issue 184: For sound, wav, mp3, and ogg should work on Linux and Windows. wav and mp3 should work on Mac.

Updated Examples

- Add happy face drawing example

31.1.79 Version 1.3.2

Released 2018-Apr-20

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

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

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

31.1.80 Version 1.3.1

Released 2018-Mar-31

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

Updated Documentation

- Update quick-links on homepage of http://arcade.academy
- Update Sprite class documentation
- Update copyright date to 2018

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

31.1.81 Version 1.3.0

Released 2018-February-11.

Enhancements

- Issue 126: Initial support for decorators.
- Issue 167: Improve audio support.
- Issue 169: Code cleanup in SpriteList.move()
- Issue 174: Support for gradients.

31.1.82 Version 1.2.5

Released 2017-December-29.

Bug Fixes

- Issue 173: JPGs not included in examples

Enhancements

- Issue 171: Clean up sprite list code

31.1.83 Version 1.2.4

Released 2017-December-23.

Bug Fixes

- Issue 170: Unusually high CPU

31.1.84 Version 1.2.3

Released 2017-December-20.

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

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

31.1.85 Version 1.2.2

Released 2017-December-02.

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

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

31.2 How to Contribute

We would love to have you contribute to the project! There are several ways that you can do so.

31.2.1 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.

31.2.2 How to contribute code

First, take some time to understand the project layout:

- *Directory Structure*
- *How to Build*
- *How to Submit Changes*

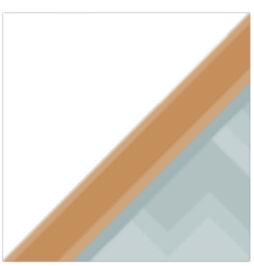
Then you can improve these parts of the project:

- **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.

A list of enhancements people have requested is available on the *Enhancement List* page.

31.3 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?

31.3.1 Why Edge Artifacts Appear

This happens when the edge of an image does not fall cleanly onto an image.

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. 4: 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.

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.)

Rotating

With rotation, it can be very difficult to get pixels lined up, and edge artifacts are common.

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.

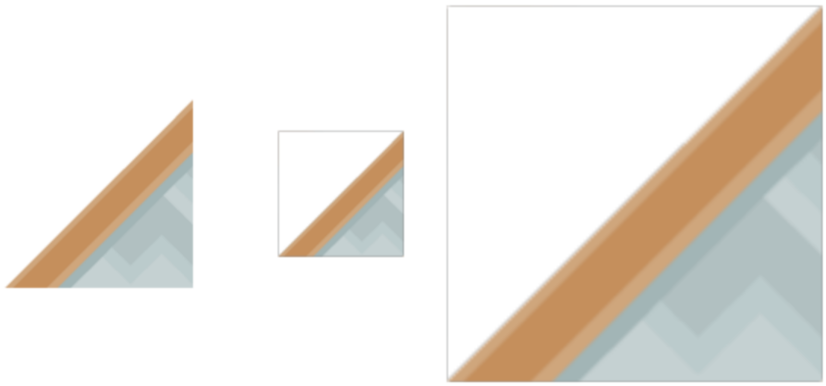


Fig. 5: Figure 2: Edge artifacts caused by scaling.

31.3.2 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.

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

```
def on_draw(self):
    self.my_sprite_list.draw(pixelated=True)
```

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.

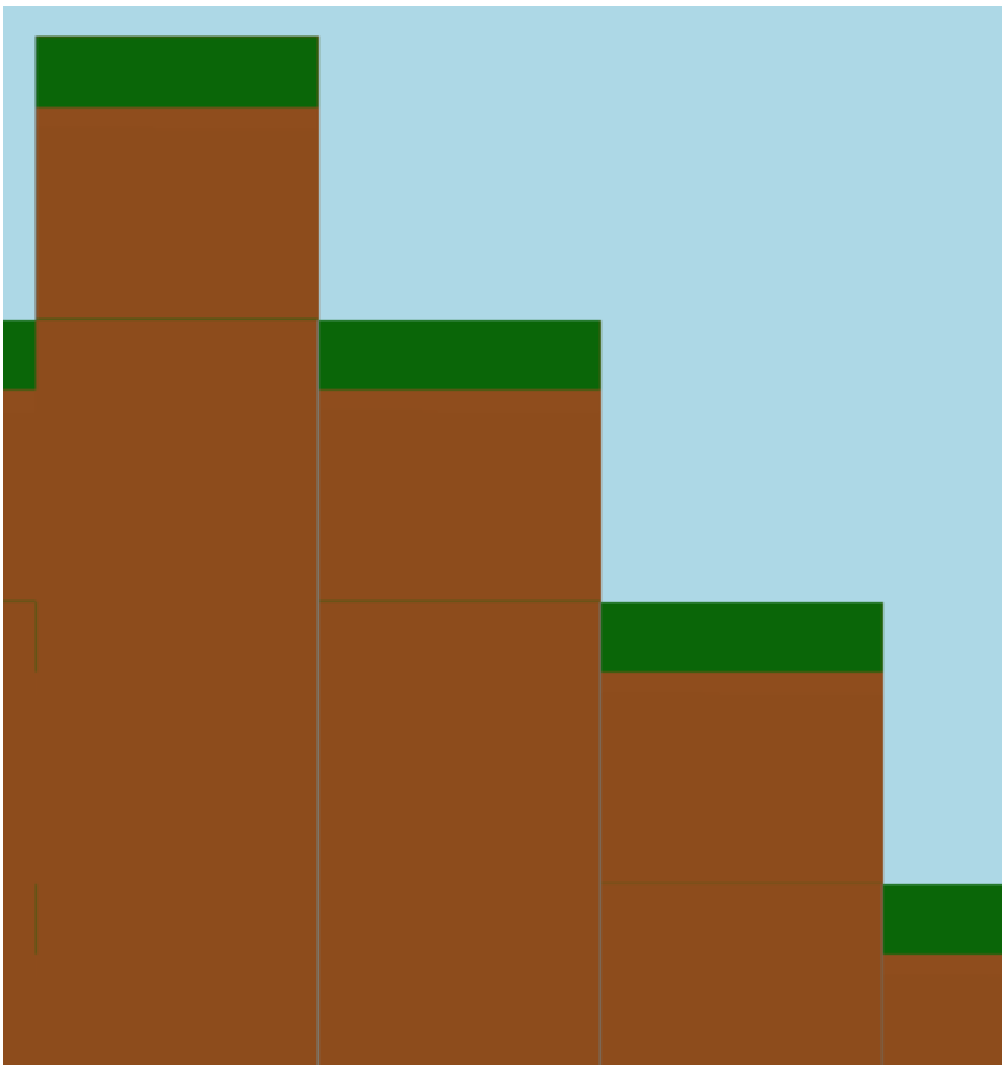


Fig. 6: Figure 3: Incorrect viewport

31.4 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 *How to Build*.

31.5 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 *How to 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.

31.6 Enhancement List

This is a list of possible enhancements opened in GitHub, but not being actively worked on. These are all good ideas. If you are thinking of helping the Arcade library by working on one of these, please re-open the issue.

31.6.1 Drawing

- Issue 433 Add support for bitmapped fonts.
- Add support for Pyglet's ImageMouseCursors

31.6.2 Sprites

- Issue 380 Add ability to specify a point the sprite rotates around.
- Issue 289 Be able to get Sprite position and velocity as vectors.
- Be able to load an Aesprite image directly (Piggy-back of Pyglet support)

31.6.3 Physics Engine

- Issue 499 Create PyMunk + TMX example.
- Issue 500 Show 'rope' effect.
- Issue 524 Add example for "push back".
- Create a simplified front-end to the PyMunk physics engine

31.6.4 Event Processing

- Issue 593 Add support for signals

31.6.5 Documentation

- Issue 452 Documentation Request - explain how delta_time works to help learners fully understand both how and why.

31.6.6 Examples

- Issue 446 Add more procedural generation examples.
- Issue 464 Add example for checkers-like game.
- Issue 523 Add sprite trigger/example for on_enter / on_exit.

This enhancement is not currently in process. Please re-open if you'd like to work on it. A full list of desired enhancements is available at:

http://arcade.academy/enhancement_list.html

31.7 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.

31.7.1 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 is_static=False
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
```

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.")
```

31.7.2 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
```

31.8 How to Build

31.8.1 Windows

Prep your system by getting the needed Python packages, listed in the `requirements.txt` file.

Create your own fork of the repository, and then clone it on your computer.

From the base directory, there is a "make" batch file that can be run with a number of different arguments, some of them listed here:

- `make test` - This runs the tests.
- `make testcov` - This runs the tests, and lists coverage
- `make dist` - Makes the distributable wheels
- `make deploy_pypi` - Uploads wheels to PyPi

Note: Placing test programs in the root of the project folder will pull from the source code in the arcade library, rather than the library installed in the Python interpreter. This is helpful because you can avoid the compile step. Just make sure not to check in your test code.

To build the docs, switch to the `doc` directory, and type `make html`.

31.8.2 Linux

Create your own fork of the repository, and then clone it on your computer.

Prep your system by downloading the needed packages:

`sudo apt-get install python-dev`

31.9 Release Checklist

1. Check for updated libraries, and if we need to pin a more recent version.
2. Run `flake8 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.

INDEX

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z