OOP in Arcade

Videogames Technology
Asignatura transversal

Departamento de Automática

Universidad de Alcalá

Objectives

1. Understand the OO API in Arcade
2. Use sprites and sprites sheets with Arcade
3. Handle user input (mouse, keyboard and joysticks)
4. Understand some multimedia file formats
5. Introduce the `Window`, `View` and `Sprite` classes

Bibliography

1. Paul Craven. The Arcade Book. Chapter 18: Using the Window class. (link).
2. Paul Craven. The Arcade Book. Chapter 19: User control. (link).
3. Paul Craven. The Arcade Book. Chapter 21: Spriters and collisions. (link).
4. Paul Craven. Using Views for Start/End Screens. (link).

Table of Contents

1. The Window class
   - Introduction
   - Extending the `Window` class
   - Main methods and attributes
   - Basic methods
   - Other methods
2. User control
   - Mouse
   - Keyboard
   - Game controller
3. The `View` class
   - The need for views
   - Changing views
   - Examples
   - Graphs
   - Finite States Machines
4. Sprites
   - Introduction
   - Spritesheets
   - Data formats
   - The `Sprite` class
   - Examples
   - Collision detection
   - Other classes
   - Sprite lists
   - Locating sprites
5. Resources management

The Window class

Introduction

Arcade has an OOP API

- More features than structured API
- Easy to use API

This API is based on the `arcade.Window` class

```
1  import arcade
2
3
4  def main():
5      window = arcade.Window(640, 480, "Example")
6
7      arcade.run()
8
9
10 main()
```

The Window class

Extending the Window class

```
1  import arcade
2
3
4  class MyGame(arcade.Window):
5
6      def __init__(self, width, height, title):
7
8          # Call the parent class's init function
9          super().__init__(width, height, title)
10
11
12 def main():
13     window = MyGame(640, 480, "Drawing Example")
14
15     arcade.run()
16
17
18 main()
```

The Window class

Main methods and attributes

arcade.Window

Methods

- `on_draw()`.
  Drawing
- `on_update(delta_time: float)`.
  Move everything. Perform collision checks. Do all the game logic here
- `clear()`.
  Clear screen
- `set_background_color(color)`.
  Set color used by `clear()`

Attributes

- `background_color`.
  Color used by `clear()`

Ball

```
import arcade


class MyGame(arcade.Window):
    def __init__(self, width, height, title):
        # Call the parent class's init function
        super().__init__(width, height, title)

        # Set the background color
        #arcade.set_background_color(arcade.color.ASH_GREY)
        self.background_color = arcade.color.ASH_GREY

    def on_draw(self):
        """ Called whenever we need to draw the window. """
        self.clear()

        arcade.draw_circle_filled(50, 50, 15, arcade.color.
            AUBURN)

def main():
    window = MyGame(640, 480, "Drawing Example")

    arcade.run()

main()
```

Animated ball

```
import arcade

class MyGame(arcade.Window):

    def __init__(self, width, height, title):
        super().__init__(width, height, title)

        arcade.set_background_color(arcade.color.ASH_GREY)

        self.ball_x = 50
        self.ball_y = 50

    def on_draw(self):
        """ Called whenever we need to draw the window. """
        self.clear()

        arcade.draw_circle_filled(self.ball_x, self.ball_y, 15,
            arcade.color.AUBURN)

    def on_update(self, delta_time):
        """ Called to update our objects."""
        self.ball_x += 1
        self.ball_y += 1
```

It violates encapsulation, (link to better solution)

The Window class

Constructor

Constructor

```
class arcade.Window(
        width: int = 1280,
        height: int = 720,
        title: Optional[str] = 'Arcade Window',
        fullscreen: bool = False,
        resizable: bool = False,
        center_window: bool = False,
        draw_rate: Optional[float] = 0.016666666666666666,
        update_rate: Optional[float] = 0.016666666666666666)
```

Remember to use reference documentation

- (`arcade.Window` reference)

$$\frac{1}{0{,}016666666666666666} = 60\text{Hz}$$

The Window class

Other methods

Other methods

- activate()
- center_window()
- close()
- get_location()
- get_size()
- maximize() and minimize()
- set_fullscreen(fullscreen: bool = True)
- set_location(x, y)

Adding a setup() method is recommended

User control

Introduction

How does the player interact with the game?

- Mouse
- Keyboard
- Game controller (joystick or gamepad)

The key is to override Window methods

User control

Mouse (I)

Overriding the following `arcade.Window` methods

- `on_mouse_motion(x, y, delta_x, delta_y)`.
  Called whenever the mouse moves.
- `on_mouse_press(x: float, y: float, button: int, modifiers: int)`.
  Called when the user presses a mouse button.
- `on_mouse_release(x: float, y: float, button: int, modifiers: int)`.
  Called when the user releases a mouse button.

Make the mouse pointer dissapear:

- Call `self.set_mouse_visible(False)` the constructor

User control

Mouse (II)

Capturing a mouse click

```
def on_mouse_press(self, x, y, button, modifiers):
    """ Called when the user presses a mouse button. """

    if button == arcade.MOUSE_BUTTON_LEFT:
        print("Left mouse button pressed at", x, y)
    elif button == arcade.MOUSE_BUTTON_RIGHT:
        print("Right mouse button pressed at", x, y)
```

Moving a ball with a mouse

(Example)

(More info about keys)

User control

Keyboard

Two events when we press a key

- Press and release

Overriding the following `arcade.Window` methods

- `on_key_press(symbol: int, modifiers: int)`
  Called when the user presses key.
- `on_key_release(symbol: int, modifiers: int)`.
  Called when the user releases key.

Capturing a key

```
1 def on_key_press(self, key, modifiers):
2     if key == arcade.key.LEFT:
3         print("Left key hit")
4     elif key == arcade.key.A:
5         print("The 'a' key was hit")
```

(Example)

User control

Game controller (I)

A computer might not have any controller, or it might five game controllers

- `arcade.get_joysticks()`

Init controller in __init__

```
1 joysticks = arcade.get_joysticks()
2
3 if joysticks:
4     self.joystick = joysticks[0]
5     self.joystick.open()
6 else:
7     print("There are no joysticks.")
8     self.joystick = None
```

User control

Game controller (II)

Read game controller value

```
def update(self, delta_time):
    # Update the position according to the game controller
    if self.joystick:
        print(self.joystick.x, self.joystick.y)
```

Centered $(0, 0)$

Down $(0, 1)$

Up $(0, -1)$

Down-left $(-1, 1)$

(Source)

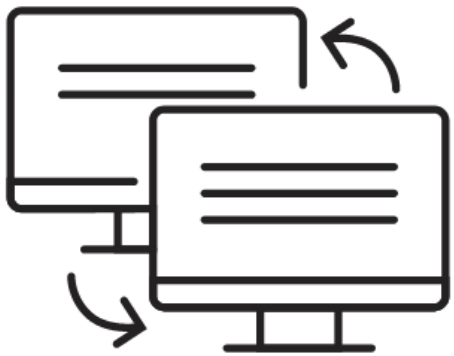
(Interesting example - Arcade 2.6.17)

The `View` class (I)

The need for views

Videogames use to show several screens

- Start screens
- Instruction screens
- Game over screens
- Pause screens



(Source)

Arcade provides the `View` class

- Very much like the Window class
- It has the `on_draw()` and `on_update()` methods

The View class

The View class (II)

Our class must inherit from `arcade.View`

```
class MyGame(arcade.Window):
```

⇒

```
class GameView(arcade.View):
```

The view does not control the window size, so

```
super().__init__(WIDTH, HEIGHT, TITLE)
```

⇒

```
super().__init__()
```

The view does not control the window size, so

```
self.set_mouse_visible(False)
```

⇒

```
self.window.set_mouse_visible(False)
```

The View class

The View class (III)

Finally, we need to create a window, a view and show that view

```
def main():
    """ Main function """

    window = arcade.Window(WIDTH, HEIGHT, TITLE)
    start_view = GameView()
    window.show_view(start_view)
    start_view.setup()
    arcade.run()
```

The View class

Changing views

`on_view_draw()`

- Run once when we switch to the view

We can switch the view any time

From a view

```
def on_update(self, delta_time):

    [...]

    if len(self.coin_list) == 0:
        view = GameOverView()
        self.window.show_view(view)
```

The `View` class

Example: Game Over screen

GameOverView 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")

    def on_draw(self):
        """ Draw this view """
        self.clear()
        self.texture.draw_sized(WIDTH / 2, HEIGHT / 2, WIDTH,
            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 View class

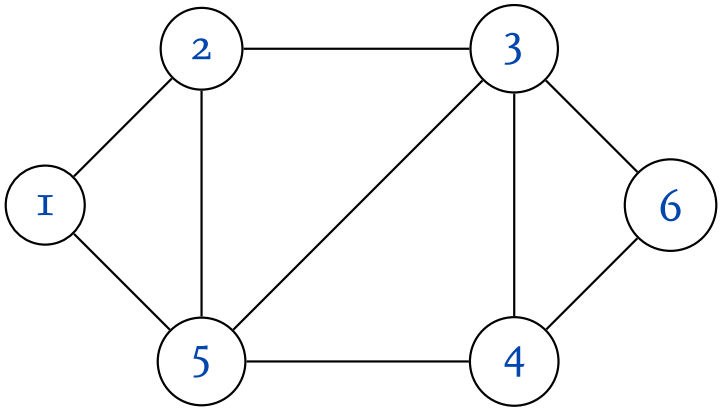
Graphs

Graph: Data structure with **nodes** and **edges**

- Widely used in programming, AI and videogames
- Huge number of applications

Central role in path planning

- (Video)
- Navigation mesh

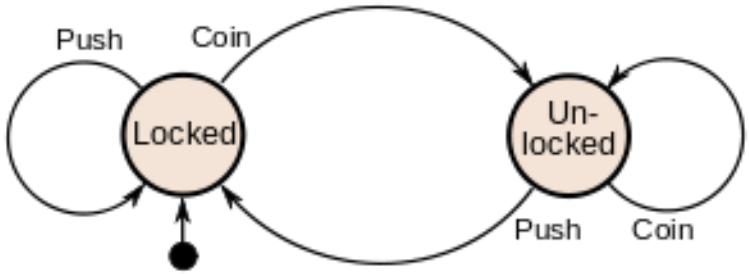


The View class

Finite States Machines (I)

Finite-State Machine (FSM)

A graph whose nodes represent states, usually associated with behaviours



(Source)

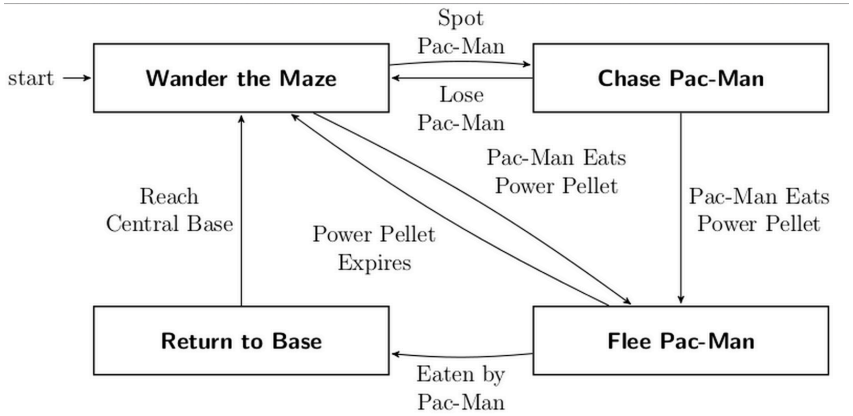
The View class

Finite States Machines (II)

FSMs have many applications

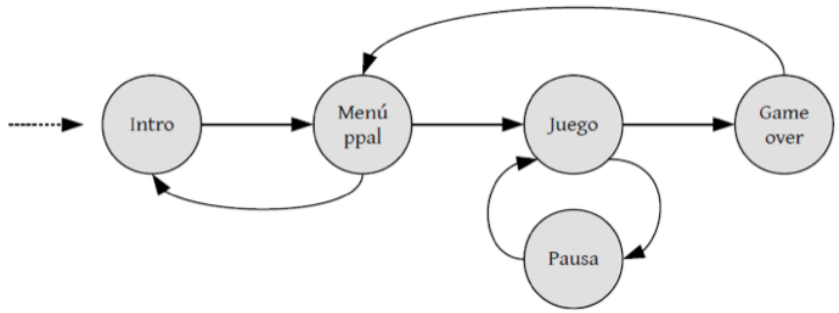
- Central role in Theory of Computation
- Good to model behaviours ... such as a NPC or an entire videogame

NPC behaviour



(Source)

Game life-cycle

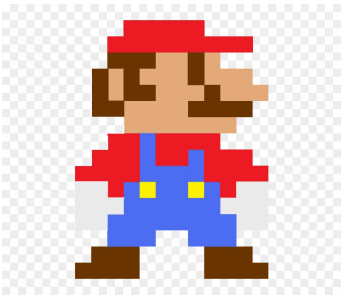


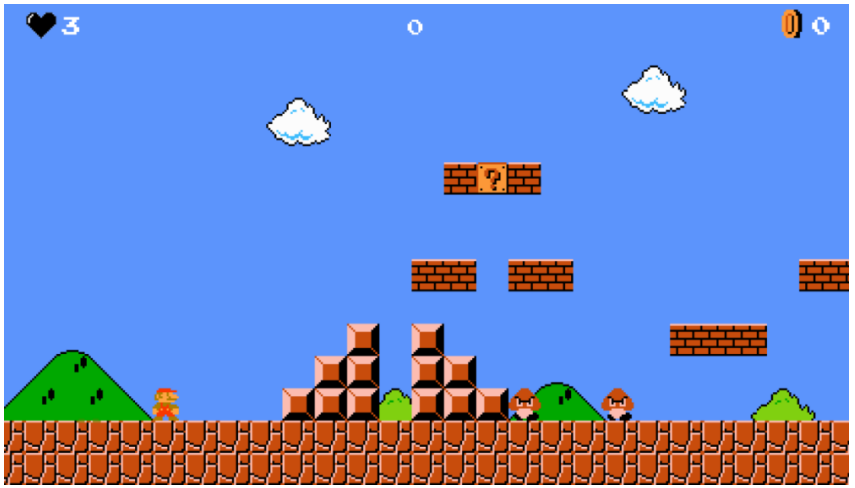
Sprites

Introduction

Sprite

A sprite is a 2D image used in videogames



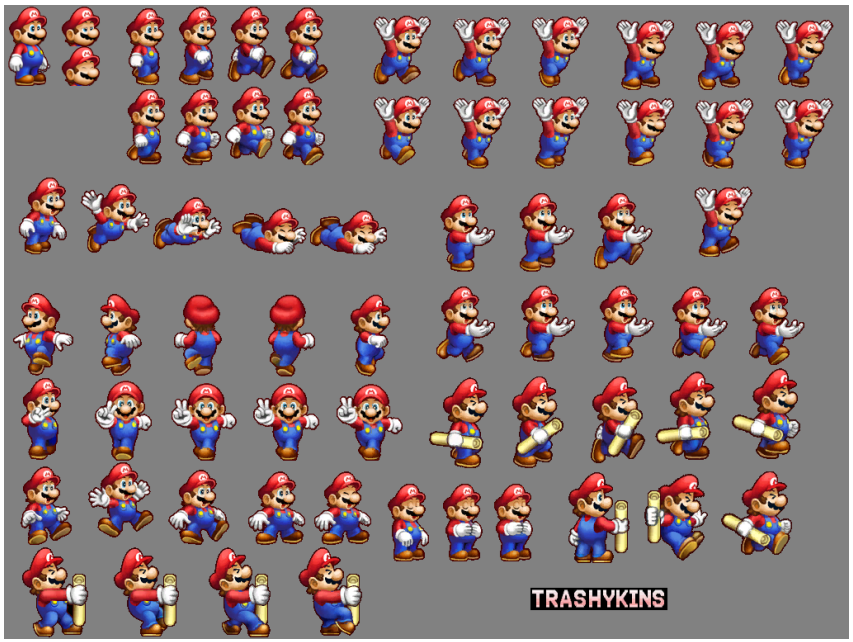


Sprites

Spritesheets

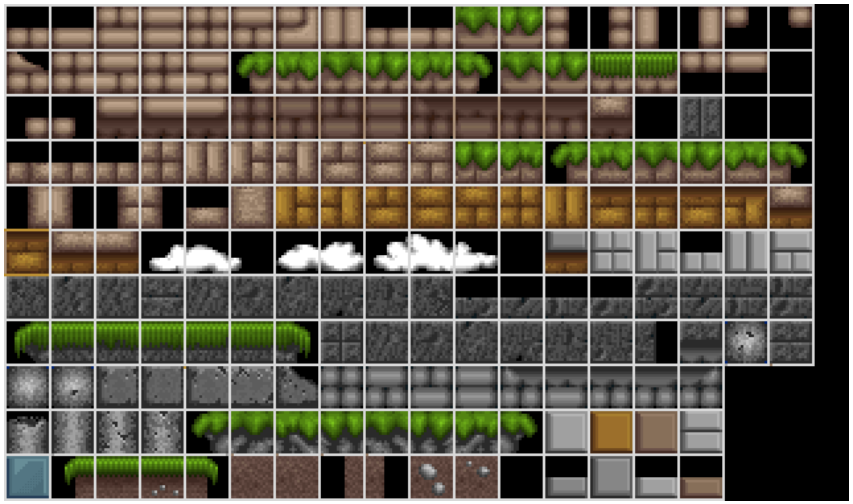
A videogame contains many sprites

- Difficult maintenance
- Solution: Spritesheets



Advantages

- One file contains many sprites
- Less I/O operations $\Rightarrow$ Better performance
- Less memory consumption



Sprites

Data formats (I)

In genenal, any data can be stored in three forms

- Not compressed
- Compressed with loss
- Compressed without loss

	Image format	Sound format	Binary data
Not compressed	BMP	WAV	
Compressed with loss	JPG	MP3	
Compressed without loss	PNG, GIF	-	ZIP, bzip, rar, ...

Sprites

Data formats (II)

Attending to what information is stored in image format, there are two types of image formats:

- Bitmap: stores each pixel
  - Scales bad
  - Formats: JPG, PNG, BMP, GIF
- Vectorial: stores coordinates
  - Scales well
  - Not supported by Arcade
  - Formats: SVG, EPS

Many open assets for your games!

- (Kenney)

Sprites

The `Sprite` class (I)

You will need to provide a path to the file

- *Absolute path*: Starts from the root directory
  - Example (Windows): `c:\\Users\atreides\Desktop\mygame\assets\sprites\mario.png`
  - Example (Linux): `/home/atreides/mygame/assets/sprites/mario.png`
- *Relative path*: Relative to the project's directory
  - Example (Windows): `assets\sprites\mario.png`
  - Example (linux): `assets/sprites/mario.png`

Always use relative paths in your projects!!!

Sprites

The Sprite class (II)

Sprites are a fundamental concept in Arcade

Creating a sprite

```
character = arcade.Sprite('images/character.png')
```

Placing a sprite

```
character.center_x = 300
character.center_y = 200
```

Sprite

The Sprite class (III)

(Reference documentation)

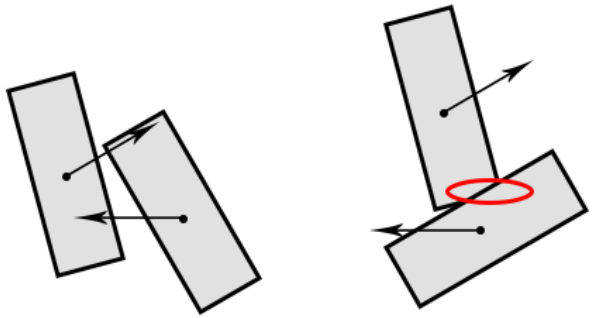
Constructor

```
class arcade.Sprite(
        filename: Optional[str] = None,
        scale: float = 1,
        image_x: float = 0, # offset within sprite sheet
        image_y: float = 0, # offset within sprite sheet
        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',
        angle: float = 0)
```

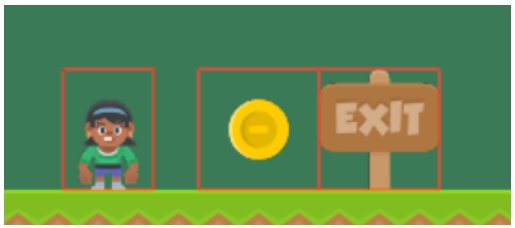
Sprite

The Sprite class (IV)

Sprites in Arcade implement collision detection and handling



Three values for `hit_box_algoritm`: 'None', 'Simple' and 'Detailed'



None'

'Simple'

'Detailed'

Sprite

The `Sprite` class (V)

`arcade.Sprite`

Methods

- `on_update(delta_time: float = 0.016)`.
- `draw()`.
- `append_texture(Texture: arcade.texture.Texture`.
  Appends a new texture (image)
- `set_texture(texture_no: int)`.
- `update_animation(delta_time: float = 0.016))`.
- `set_position(center_x: float, center_y: float)`.
- `turn_left(theta: float = 90.0)`
- `turn_right(theta: float = 90.0)`
- `stop()`

Attributes

- alpha: int.
- angle: float.
- bottom: float and bottom: float.
- center_x: float and center_y: float.
  Center of the sprite
- change_x: float and change_y: float.
  Velocity in X and Y
- height: float.
- visible: bool.

(Reference documentation)

Sprites

The Sprite class: Examples

```
# Make the sprite invisible
sprite.visible = False

# Change back to visible
sprite.visible = True

# Toggle visible
sprite.visible = not sprite.visible
```

Sprites

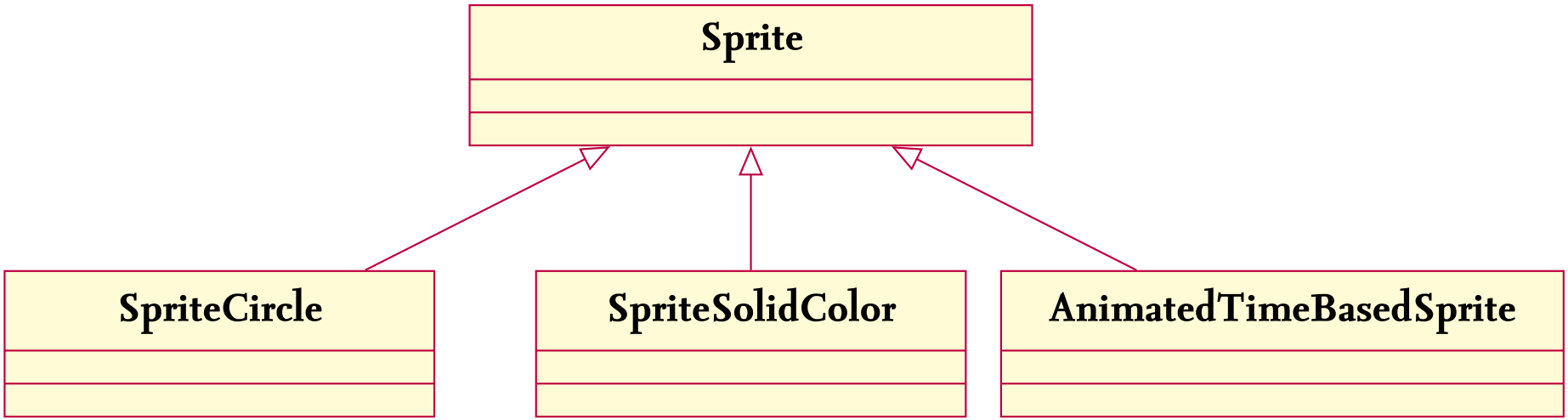
Collision detection

Collision detection methods

- `collides_with_point(point: Union[Tuple[float, float], List[float]])` $\rightarrow$ `bool`
- `draw_hit_box()`.
- `collides_with_list(sprite_list: SpriteList)` $\rightarrow$ `bool`

Sprite

Other classes



- (SpriteCircle documentation)
- (SpriteSolidColor documentation)
- (SpriteAnimatedTimeBasedSprite documentation)

Sprites

Sprite lists (I)

Arcade stores sprites in lists

```
wall = arcade.Sprite('images/boxCrate.png')
wall.center_x = 300
wall.center_y = 300
wall_list = arcade.SpriteList()
wall_list.append(wall)
```

Lists can be manipulated as a whole

```
wall_list.draw()
```

And sprites can be removed from a list

```
wall.remove_from_sprite_lists()
```

Sprites

Sprite lists (II)

Lists in Arcade also implement collision detection

```
 hit_list =
arcade.check_for_collision_with_list(player_sprite,
coin_list)
```

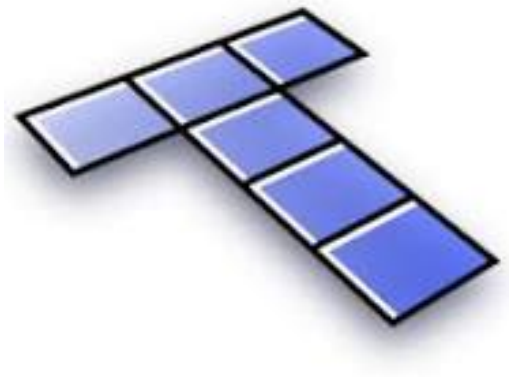
Sprites

Locating sprites

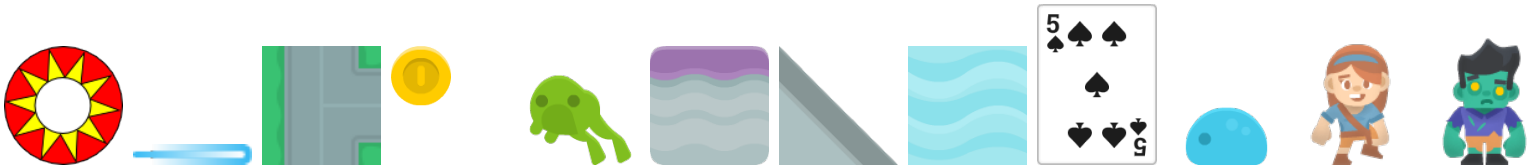
Locating sprites in the game is a tought work

- Closely related to level design
- There are tools that ease this task

(Tiled Map Editor)



Resources management (I)



Arcade comes along a collection of built-in resources

- Good for testing
- No need of files in disk
- Images, music, sounds, tiled maps and video

The path is something like ':resources:/path/to/resource'

```
arcade.Sprite(":resources:images/items/coinGold.png")
```

(List of resources)
(Arcade source code)

Resources management (II)

Adding new resources handlers

```
1 arcade.resources.add_resource_handle("characters", "resources/
      characters")
2 arcade.resources.add_resource_handle("maps", "resources/maps")
3 arcade.resources.add_resource_handle("data", "resources/data")
4 arcade.resources.add_resource_handle("sounds", "resources/sounds
      ")
5 arcade.resources.add_resource_handle("misc", "resources/misc")
```

Using resources handlers

```
1 self.player_sprite = PlayerSprite(":characters:Female/Female
      18-4.png")
```