

# OOP in Arcade

Videogames Technology  
Asignatura transversal

Departamento de Automática

## 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. The Texture class

## ■ Introduction

- Hitboxes
- Data formats

## 5. The Sprite class

- Introduction
- Sprites in Arcade
- Methods and attributes
- Animation example
- Collision detection
- Class hierarchy

## 6. Sprite lists

- Motivation
- Drawing with sprite lists

## 7. Resources management

- Spritesheets
- Tile maps
- Built-in resources

# 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

```
1 import arcade
2
3
4 class MyGame(arcade.Window):
5     def __init__(self, width, height, title):
6         # Call the parent class's init function
7         super().__init__(width, height, title)
8
9         # Set the background color
10        #arcade.set_background_color(arcade.color.ASH_GREY)
11        self.background_color = arcade.color.ASH_GREY
12
13    def on_draw(self):
14        """ Called whenever we need to draw the window. """
15        self.clear()
16
17        arcade.draw_circle_filled(50, 50, 15, arcade.color.
18                                  AUBURN)
19
20    def main():
21        window = MyGame(640, 480, "Drawing Example")
22
23        arcade.run()
24
25    main()
```

## Animated ball

```
1 import arcade
2
3 class MyGame(arcade.Window):
4
5     def __init__(self, width, height, title):
6         super().__init__(width, height, title)
7
8         arcade.set_background_color(arcade.color.ASH_GREY)
9
10        self.ball_x = 50
11        self.ball_y = 50
12
13    def on_draw(self):
14        """ Called whenever we need to draw the window. """
15        self.clear()
16
17        arcade.draw_circle_filled(self.ball_x, self.ball_y, 15,
18                                  arcade.color.AUBURN)
19
20    def on_update(self, delta_time):
21        """ Called to update our objects. """
22        self.ball_x += 1
23        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] = 0.01666666666666666,  
    update_rate: Optional[float] = 0.01666666666666666)
```

Remember to use reference documentation

- (`arcade.Window` reference)

$$\frac{1}{0,0166666666666666} = 60\text{Hz}$$

# 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

```
1 def on_mouse_press(self, x, y, button, modifiers):
2     """ Called when the user presses a mouse button. """
3
4     if button == arcade.MOUSE_BUTTON_LEFT:
5         print("Left mouse button pressed at", x, y)
6     elif button == arcade.MOUSE_BUTTON_RIGHT:
7         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

```
1 def update(self, delta_time):  
2     # Update the position according to the game controller  
3     if self.joystick:  
4         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

Arcade provides the `View` class

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



(Source)

# 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
```

```
1  class GameOverView(arcade.View):
2      """ View to show when game is over """
3
4      def __init__(self):
5          """ This is run once when we switch to this view """
6          super().__init__()
7          self.texture = arcade.load_texture("game_over.png")
8
9      def on_draw(self):
10         """ Draw this view """
11         self.clear()
12         self.texture.draw_sized(WIDTH / 2, HEIGHT / 2, WIDTH,
13                               HEIGHT)
14
15     def on_mouse_press(self, _x, _y, _button, _modifiers):
16         """ If the user presses the mouse button, re-start the
17             game. """
18         game_view = GameView()
19         game_view.setup()
20         self.window.show_view(game_view)
```

# The View class

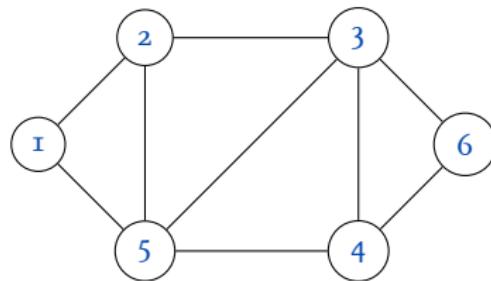
## Graphs

**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

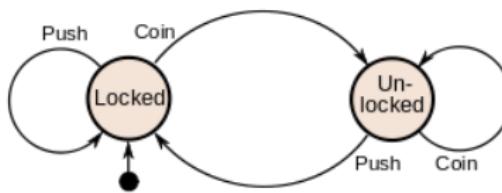


# The View class

## Finite States Machines (I)

### Finite-State Machine (FSM)

A graph whose nodes represent states,  
usually associated with behaviours



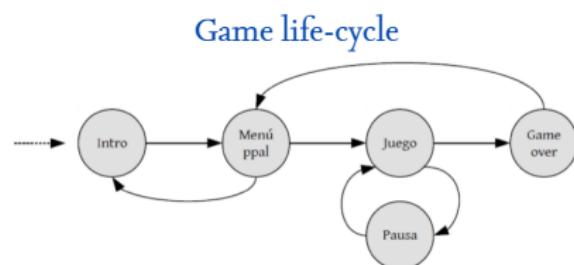
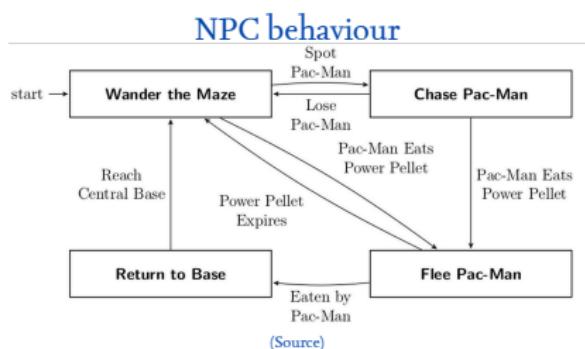
(Source)

# 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



# The Texture class

## Introduction



In Arcade, images are named **textures**

- Implemented by the `arcade.Texture` class

Two texture loading methods

- `arcade.load_texture()`
- `arcade.Texture()`

### Functional

```
mario = arcade.load_texture ("mario.png")  
mario = arcade.load_texture (Path("mario.png"))
```

### Object-oriented

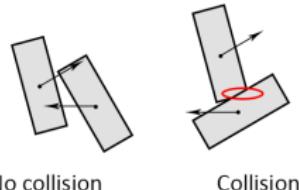
```
mario = arcade.Texture ("mario.png")  
mario = arcade.Texture (Path ("mario.png"))
```

# The Texture class

## Hitboxes

The Texture class provides

- Hitbox geometry, used in **collision detection**
- **Caching**



(Source)



# The Texture class

## Data formats (I)

In general, 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, ...

# The Texture class

## 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)

# The Sprite class

## Introduction

### Sprite

A sprite is a 2D image used in videogames



# The Sprite class

## Sprites in Arcade

In Arcade, a sprite contains ...

- One or more textures
- Texture size
- Image data
- Collision detection

Acts as a base class for (our)  
derived classes

Implemented in the `arcade.Sprite` class

```
mario = arcade.Sprite('mario.png')
```

```
mario = arcade.Sprite(Path('mario.png'))
```

Inherits from `arcade.BasicSprite`

# The Sprite class

## Constructor

### Constructor

```
class arcade.Sprite(  
    path_or_texture: str | Path | bytes | Texture | None = None,  
    scale: float | tuple[float | int, float | int] | Vec2 = 1.0,  
    center_x: float = 0.0,  
    center_y: float = 0.0,  
    angle: float = 0.0,  
    **kwargs: Any)
```

(Reference documentation)

# The Sprite class

## Methods and attributes (I)

### arcade.Sprite

#### Methods

- `update(delta_time: float = 0.016)`
- `append_texture(texture: Texture)`
- `set_texture(texture_no: int)`
- `update(delta_time: float = 0.016)`
- `stop()`
- `kill()`
- `draw_hit_box()`

#### Attributes

- `center_x: float and center_y: float.`  
Center of the sprite
- `change_x: float and change_y: float.`  
Velocity in X and Y
- `texture: Texture`
- `textures: list[Texture]`
- `alpha: int [0-255]`
- `angle: float`
- `scale: tuple`
- `bottom: float and top: float`
- `left: float and right: float`
- `visible: bool`

# The Sprite class

## Methods and attributes (II)

### Placing a sprite

```
mario. center_x = 300  
mario. center_y = 200
```

### Moving a sprite

```
mario. change_x = 10  
mario. change_y = 1
```

### Chaging visibility

```
1 # Make the sprite invisible  
2 mario. visible = False  
  
3  
4 # Change back to visible  
5 mario. visible = True  
  
6  
7 # Toggle visible  
8 mario. visible = not sprite. visible
```

```
1 class Explosion(arcade.Sprite):
2     """ This class creates an explosion animation """
3
4     def __init__(self, texture_list):
5         super().__init__(texture_list[0])
6         # How long the explosion has been around.
7         self.time_elapsed = 0
8
9         # Start at the first frame
10        self.current_texture = 0
11        self.textures = texture_list
12
13    def update(self, delta_time=1 / 60):
14        self.time_elapsed += delta_time
15        # Update to the next frame of the animation.
16        # If we are at the end of our frames, then delete this
17        # sprite.
18        self.current_texture = int(self.time_elapsed * 60)
19        if self.current_texture < len(self.textures):
20            self.set_texture(self.current_texture)
21
22    else:
23        self.remove_from_sprite_lists()
```

(Source)

# The Sprite class

## Collision detection

Methods in `arcade.Sprite` to detect collisions

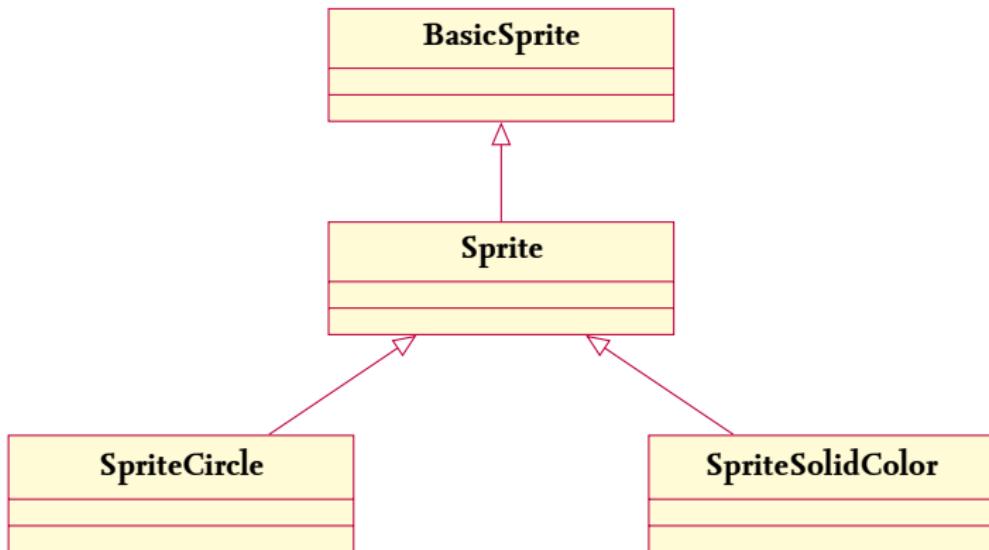
- `collides_with_list(sprite_list: SpriteList) → list(SpriteType)`
- `collides_with_point(point: Union[Tuple[float, float], List[float]]) → bool`
- `collides_with_sprite(point: BasicSprite) → bool`
- `draw_hit_box()`

Functions in Arcade to detect collisions

- `arcade.check_for_collision(sprite1: BasicSprite, sprite2: BasicSprite) → bool`
- `arcade.check_for_collision_with_list(sprite: BasicSprite, sprite_list: SpriteList) → bool`

# The Sprite class

## Class hierarchy



- ([SpriteCircle documentation](#))
- ([SpriteSolidColor documentation](#))

# Sprite lists

## Motivation

Arcade heavily uses **sprite lists**

- Batch drawing (GPU optimization)
- Collision detection
- Debug drawing for hit boxes

Drawing with sprite lists

1. Create a **SpriteList**
2. Create and append **Sprite** instances
3. Call **draw()**

```
for sprite in sprites :  
    sprite.draw() # Not in 3.X  
    sprite_list.draw()
```

# Sprite lists

## Drawing with sprite lists

First, we create a sprite list

```
1 wall = arcade.Sprite('boxCrate.png')
2 wall.center_x = 300
3 wall.center_y = 300
4 wall_list = arcade.SpriteList()
5 wall_list.append(wall)
```

and then we draw it

```
wall_list.draw()
```

# Sprite lists

## Collision detection

Sprites can be removed from a list

```
wall . remove_from_sprite_lists ()
```

Lists in Arcade also implement collision detection

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

Complete example in (python -m arcade.examples.sprite\_minimal)

# Resources management

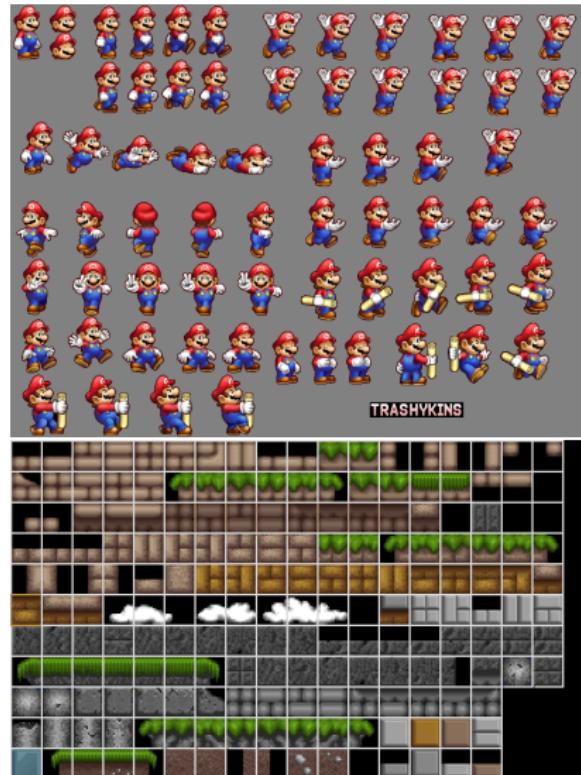
## Spritesheets (I)

A videogame contains many sprites

- Difficult maintenance
- Solution: Spritesheets

Advantages

- One file contains many sprites
- Less I/O operations ⇒ Better performance
- Less memory consumption



# Resources management

## Spritesheets (II)

Arcade provides the `arcade.SpriteSheet` class

- Gets sprites from a spritesheet



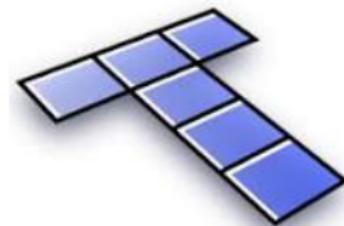
# Resources management

## Tile maps

Locating sprites in the game is a tough work

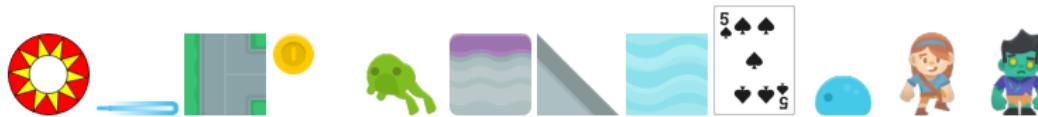
- Closely related to **level design**
- There are tools that ease this task
- Stores maps as JSON files
- Supported by `arcade.TileMap`

(Tiled Map Editor)



# Resources management

## Resources handlers



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")
```

# Resources management

## Built-in resources

### Type of built-in resources

- Images (characters, backgrounds, etc)
- Sounds and music
- Fonts
- GUI basic assets (such as buttons and checkboxes)

(List of resources)