OArcade Reference Manual

OVERVIEW

A basic starting template looks like this:

```
import op_arcade as oa
import op_arcade.constants_classes as cc

class Level_1(oa.View):

    def on_create(self):
         self.background_color = cc.Colors.AZURE
         self.player = Player(100, 100)

    def on_step(self):
        self.clear()
        self.scene.draw()
        self.draw_grid()



class Player(oa.Sprite):

    def on_create(self):
        self.visual = cc.Anims.Players.PIXEL_GUY
        self.layer = cc.Layers.MOVING_GAME_OBJECTS


    def on_step(self):
         # Code to execute on every frame refresh
         pass


oa.run(Level_1)
```

There are four sections of code to implement:

- **`class Level_1`** → **`def on_create`** – code that runs when the level is created
- **`class Level_1`** → **`def on_step`** – code that runs on every frame refresh
- **`class Player`** → **`def on_create`** – code that runs when the player is created
- **`class Player`** → **`def on_step`** – code that runs on every frame refresh

Inside the **`Level_1`** class, **`self`** refers to the level object, which is an extension of **`oa.View`**.

Inside the **`Player`** class, **`self`** refers to the player object, which is an extension of **`oa.Sprite`**.

THE `constants_classes` MODULE

This module provides shortcuts to built-in images, animations, colours, and more. Typing `cc.` in your editor will bring up a box of categories. Pick a category, type in another dot, and you'll get a list of subcategories, and so on.

The hierarchy is as follows:

- cc
  - Images
    - Walls → *…many*
    - Items → *…many*
    - Players → *…many*
    - Enemies → *…many*
  - Anims
    - Players
      - TopDown → *…many*
      - SideScrolling → *…many*
    - Enemies → *…many*
  - Layers
    - BACKGROUND
    - DECOR
    - WALLS
    - STATIONARY_GAME_OBJECTS
    - MOVING_GAME_OBJECTS
  - Colors → *…many*
  - Keys → *…many*
  - Mouse → *…many*
    - LEFT_BUTTON
    - MIDDLE_BUTTON
    - RIGHT_BUTTON

THE `oa.View` CLASS

This is an extension of the `View` class in the arcade module. The most useful attributes and methods inherited from this class are documented here. You will extend this class for every level or screen in your game, but you won't explicitly create objects from this class – the game engine does this for you.

Attributes

Most of these attributes are read only, with the exception being `background_color`. Within an `ov.View` class (such as a level), you can access these by preceding them by `self.`, and from elsewhere you can access them by preceding them by `oa.refs.cur_view.`.

Name	Description
`center_x`	The centre of the window along the x-axis
`center_y`	The centre of the window along the y-axis
`height`	The height of the window
`width`	The width of the window
`mouse_x`	The current x-coordinate of the mouse pointer.
`mouse_y`	The current y-coordinate of the mouse pointer.
`background_color`	The background colour of the level

Methods you will call

You will usually use these within an `ov.View` class (such as a level). You can access them from within an `oa.View` class by preceding them by `self.`, and from elsewhere you can access them by preceding them by `oa.refs.cur_view.`.

`add_default_layers` – Adds the five default layers to the view: ***background***, ***decor***, ***walls***, ***stationary_game_objects***, and ***moving_game_objects***. The system will call this automatically for you unless you've overridden the `setup_layers` method. Does not take any arguments.

`add_layer` – Adds a new layer to the view

ARGUMENTS	
`name`	The name of the layer
`colliding=False`	Will items in this layer act as walls?
`use_spatial_hash=False`	For performance reasons, this should be set to `True` for non-moving colliding sprites (such as walls).

add_sf_map_layers – Adds layers from a map file created with ***SpriteFusion***.

ARGUMENTS	
path	The path to the folder exported from SpriteFusion containing **map.json** and **spritesheet.png**
scale=1	The scale factor for all map tiles

clear - Clears the window with the configured background color. Does not take any arguments.

create_hwall – Creates a horizontal row of sprites on the wall layer (meaning sprites with physics enabled will not be able to pass through).

ARGUMENTS	
x	The x coordinate where the first sprite in the row will be centered
y	The y coordinate where the first sprite in the row will be centered
length_in_blocks	How many blocks will be in the row
visual_filename	The filename of the image or animation this sprite will use
scale=1	The scale factor for the tiles

create_vwall – Creates a vertical column of sprites on the wall layer (meaning sprites with physics enabled will not be able to pass through).

ARGUMENTS	
x	The x coordinate where the first sprite in the column will be centered
y	The y coordinate where the first sprite in the column will be centered
length_in_blocks	How many blocks will be in the column
visual_filename	The filename of the image or animation this sprite will use
scale=1	The scale factor for the tiles

create_sprites – Create several copies of identical sprites.

ARGUMENTS	
points	A list of tuples containing the points where the sprites should be created.
sprite_subclass	A reference to the class that will create the sprites. Should be either **oa.Sprite** or a subclass of this.
Further arguments and keyword arguments provided will be passed to the **sprite_subclass** constructor.	

EXAMPLE
`self.create_sprites([(100, 500), (150, 500), (200, 500), (250, 500)], oa.Sprite, visual_filename=cc.Anims.Items.GOLD_COIN, scale=0.5)`

`create_wall_grid` – Creates a grid of sprites on the wall layer (meaning sprites with physics enabled will not be able to pass through).

ARGUMENTS	
`x`	The x coordinate where the bottom-left sprite will be centered
`y`	The y coordinate where the bottom-left sprite will be centered
`blocks_per_row`	How many blocks will be in each row
`blocks_per_column`	How many blocks will be in each column
`visual_filename`	The filename of the image or animation this sprite will use
`scale=1`	The scale factor for the tiles

`draw_grid` – Creates a grid on top of the level. Mainly used for development purposes to help with sprite placement.

ARGUMENTS	
`color=cc.Colors.GRAY`	The colour of the grid.

`is_key_down` – Returns **`True`** if the specified keyboard key is currently down, otherwise **`False`**.

ARGUMENTS	
`key`	The key to check (use **`cc.Keys.___`**)

Methods to you can override in subclasses (eg – in your levels)

These methods are called automatically by the system under certain circumstances. They generally do nothing, and are there for you to override in your subclasses. Use the method signatures as shown below to handle these events. For example, if you want to handle key presses in a certain level, you would add the method:

```
def on_key_press(self, symbol, modifiers):
    # add code here to handle key presses
```

Most of the arguments are self explanatory. Here is some further explanation on some of them:

- **`self`** – Ignore this one.
- **`symbol`** – Found in methods handling keyboard events, it indicates what key was pressed and will match a constant under **`cc.Keys.`**.
- **`modifiers`** - Tells you if keys such as SHIFT and CTRL were being held down when the event occurred. For more information on modifiers see the arcade documentation.
- **`buttons`** – Found in methods handling mouse events, it indicates which mouse button was pressed. Possible values are **`cc.Mouse.LEFT_BUTTON`**, **`cc.Mouse.MIDDLE_BUTTON`**, and **`cc.Mouse.RIGHT_BUTTON`**.

- **dx** and **dy** – Found in methods involving mouse movement, it indicates the change in the x and the y coordinates since the last time the method was called.

Method Signature	Description
`on_create(self)`	Called when the view is created. Add code in here to initialize the level/view.
`on_step(self)`	Called on every frame refresh. Add code here that should be executing repeatedly during game play.
`on_key_press(self, symbol, modifiers)`	Called when a key is pressed.
`on_key_release(self, symbol, modifiers)`	Called when a key is released.
`on_mouse_drag(self, x, y, dx, dy, buttons, modifiers)`	Called when the mouse is dragged inside the window.
`on_mouse_enter(self, x, y)`	Called when the mouse enters the view.
`on_mouse_enter(self, x, y)`	Called when the mouse leaves the view. Note the **x** and **y** will be outside the window view.
`on_mouse_motion(self, x, y, dx, dy)`	Called repeatedly while the mouse is moving in the window area.
`on_mouse_press(self, x, y, buttons, modifiers)`	Called when a mouse button is pressed.
`on_mouse_release(self, x, y, buttons, modifiers)`	Called when a mouse button is released.
`on_mouse_scroll(self, x, y, scroll_x, scroll_y)`	Called when the mouse wheel is scrolled. The scroll arguments may be positive or negative to indicate direction, but the units are unstandardized.
`on_resize(self, width, height)`	Called when the window is resized.
`layer_setup`	Called prior to **on_create**. You can override this to set up your layers manually rather than just having the default layers added. If you override this, the default layers will not be added unless you explicitly call the **add_default_layers** method.

THE `oa.Sprite` CLASS

This is an extension of the **`Sprite`** class in the arcade module. The most useful attributes and methods inherited from this class are documented here.

You can create objects from this class directly to create simple sprites such as items, but to make complex sprites such as players and enemies, you'll usually need to extend this class.

Constructor

The two mandatory arguments are the x and y coordinates of where the sprite will be placed (the coordinates represent the *centre* of the sprite). There are several optional keyword arguments:

- **`visual_filename=None`** – The filename of the image or animation this sprite will use. Animations are YAML files which are documented later in this manual.
- **`layer=None`** – The layer the sprite should be added to. Unless you've manually created your own layers, you should set this to the appropriate option below to optimize performance:
  - `cc.Layers.BACKGROUND`
  - `cc.Layers.DECOR`
  - `cc.Layers.WALLS`
  - `cc.Layers.STATIONARY_GAME_OBJECTS`
  - `cc.Layers.MOVING_GAME_OBJECTS`
- **`active_anim=None`** – If this sprite contains multiple animations, the name of the first animation to be used.
- **`group=None`** – You can optionally add the sprite to a group, which can be used later on to facilitate collision detection.
- **`scale=1`** – The scale factor of the sprite.
- **`angle=0`** – The angle of rotation of the sprite.

Example

```
oa.Sprite(300, 300, cc.Anims.Items.GOLD_COIN,
    cc.Layers.STATIONARY_GAME_OBJECTS, scale=0.5, group="coin")
```

Attributes

Name	Description
`alpha`	A value between 0 and 255 indicating the transparency of the sprite, where 0 is completely invisible and 255 is completely opaque.
`angle`	The angle of rotation of the sprite.
`bottom`	The bottom y position of the sprite.
`center_x`	The centre x position of the sprite.
`center_y`	The centre y position of the sprite.
`change_x`	The sprite's speed along the x axis.
`change_y`	The sprite's speed along the y axis.
`color`	The color tint of the sprite. Any colour format compatible with the arcade game engine is acceptable, but it is easiest to pick a color from **`cc.Colors`**.

`height`	The height of the sprite
`layer`	The name of the layer the sprite is part of. See the **Constructor** section above for more information about layers.
`left`	The leftmost x position of the sprite.
`right`	The rightmost x position of the sprite.
`scale`	The scale factor of the entire sprite.
`scale_x`	The scale factor along the x axis.
`scale_y`	The scale factor along the y axis.
`top`	The highest y position of the sprite.
`visible`	Boolean value – **`True`** if sprite is visible, **`False`** if sprite is invisible.
`visual`	The filename of the image or animation of the sprite.
`view`	A reference to the current **`View`** object.
`width`	The height of the sprite
`x`	The centre x position of the sprite (alias of **`center_x`**)
`y`	The centre y position of the sprite (alias of **`center_y`**)

Methods you will call

`activate_anim` – Activate an animation on a sprite containing multiple animations.

ARGUMENTS	
`name`	The name of the animation to activate.
`hflip=False`	Whether the animation should be flipped horizontally.
`vflip=False`	Whether the animation should be flipped vertically.
`start_frozen=False`	Whether the animation should start frozen on a single frame.

`bind_platformer_directional_keys` – Designed to be called on the Player sprite(s) only for platformer games. Sets the movement of this sprite to be controlled by three keyboard keys for the directions **left** and **right** and for jumping.

ARGUMENTS	
`speed=5`	How fast the sprite should move.
`left=cc.Keys.LEFT`	The key that should be used for moving left.
`right=cc.Keys.RIGHT`	The key that should be used for moving right.
`jump=cc.Keys.UP`	The key that should be used for jumping.
`jump_strength=10`	The strength of the player's jump (or more specifically, the initial value of the player's **`change_y`** when the jump begins).

`bind_top_down_directional_keys` – Designed to be called on the Player sprite(s) only for top-down games. Sets the movement of this sprite to be controlled by four keyboard keys for the directions **up**, **down**, **left**, and **right**.

ARGUMENTS	
`speed=5`	How fast the sprite should move.
`up=cc.Keys.UP`	The key that should be used for moving up.
`down=cc.Keys.DOWN`	The key that should be used for moving down.
`left=cc.Keys.LEFT`	The key that should be used for moving left.
`right=cc.Keys.RIGHT`	The key that should be used for moving right.

`enable_physics` – Usually called on the Player sprite(s) only. Enables simple physics on this sprite. Prevents collisions with walls (*ie*, any sprites inside a colliding layer), and optionally enables gravity.

ARGUMENTS	
`gravity=0`	The strength of the gravity. For a platformer game, a value of 0.5 would be reasonable to start with, then you can adjust from there.

`find_colliding` – If we're colliding with at least one instance of the specified collider, returns a reference to the first instance found. Otherwise returns **`None`**.

ARGUMENTS	
`collider`	What we're detecting a collision with. Can be either a reference to a single sprite, a string matching the name of a sprite group, or a reference to a class that extends **`oa.Sprite`**.

`freeze_anim` – Freeze the animation on the current frame (default), or on a specified frame.

ARGUMENTS	
`frame_index=None`	The index of the frame to freeze on. If not specified, the current frame will be used.

`get_hit_list_with` – Returns a list of all instances of the specified collider that we're currently colliding with.

ARGUMENTS	
`collider`	What we're detecting a collision with. Can be either a reference to a single sprite, a string matching the name of a sprite group, or a reference to a class that extends **`oa.Sprite`**.

`has_collision_with` – Returns **`True`** if the sprite is colliding with the specified collider, otherwise **`False`**.

ARGUMENTS	
`collider`	What we're detecting a collision with. Can be either a reference to a single sprite, a string matching the name of a sprite group, or a reference to a class that extends **`oa.Sprite`**.

`move_towards_xy` – Sets the player's **`change_x`** and **`change_y`** so that it moves towards the specified (x, y) coordinates.

ARGUMENTS	
`x`	The x coordinate to travel towards.
`y`	The y coordinate to travel towards.
`speed`	The speed to move towards the coordinates.

`set_visual` – Set the visual, and if applicable, the active animation of the sprite.

ARGUMENTS	
`visual_filename`	The filename of the image or animation this sprite will use. Animations are YAML files which are documented later in this manual.
`active_anim=1`	If this sprite contains multiple animations, the name of the first animation to be used.

Methods to you can override in subclasses (eg – in your levels)

These methods are called automatically by the system under certain circumstances. They generally do nothing, and are there for you to override in your subclasses. Use the method signatures as shown below to handle these events.

Method Signature	Description
`on_create(self)`	Called when the sprite is created. Add code in here to initialize the sprite.
`on_step(self)`	Called on every frame refresh. Add code here that should be executing repeatedly during game play.

THE oa.Camera CLASS

This is an extension of the `Camera2D` class in the arcade module. The features documented here are the minimum you need when setting up both *word* and *dashboard* cameras.

Example

```
import arcade
import op_arcade as oa
import op_arcade.constants_classes as cc

class Level_1(oa.View):

    def on_create(self):
        self.dashboard_camera = oa.Camera() # Dashboard stays stationary
        self.world_camera = oa.Camera()  # World scrolls to follow player
        self.points = 0
        self.points_text_obj = arcade.Text(f"Points: {self.points}", 10, self.height-20)
        self.player = Player(100, 100)

    def on_step(self):
        self.clear()

       # Scroll the world camera so it centers on the player
        self.world_camera.center_on_sprite(self.player, left_bound=0,
             lower_bound=0,right_bound=2000, upper_bound=2000)

       # Use the world camera to draw the scene, and optionally the grid
        self.world_camera.use()
        self.scene.draw()
        self.draw_grid()

       # Use the dashboard camera to draw stationary dashboard objects
        self.dashboard_camera.use()
        self.points_text_obj.draw()


class Player(oa.Sprite):

    def on_create(self):
        self.visual = cc.Anims.Players.PIXEL_GUY
        self.layer = cc.Layers.MOVING_GAME_OBJECTS
        self.bind_top_down_directional_keys(speed=10)

oa.run(Level_1)
```

Constructor

The constructor accepts no arguments.

Methods

`center_on_sprite` – Centres the camera on a specified sprite. You can also specify bounds that the camera will not exceed.

ARGUMENTS	
`sprite`	A reference to the sprite to centre on.
`lower_bound=None`	The minimum y value the camera will show.
`upper_bound=None`	The maximum y value the camera will show.
`left_bound=None`	The minimum x value the camera will show.
`right_bound=None`	The maximum x value the camera will show.

`point_in_view` – Checks to see if a given coordinate is in view of the camera.

ARGUMENTS	
`x`	The x-coordinate of the point to check.
`y`	The y-coordinate of the point to check.

`use` – Use this camera. Does not take any arguments.

THE `oa.Wall` CLASS

A convenience class for creating walls. This is identical to the **`oa.Sprite`** class, except that all walls are automatically added to the **`cc.Layers.Walls`** layer.

Example

```
oa.Wall(100, 300, cc.Images.Walls.BOX_CRATE_EMPTY, scale=0.25)
```

OTHER FUNCTIONS IN THE `oa` MODULE

These functions are not tied to any classes, and are called using **`oa.function_name(...)`**.

`run` – Begins the program.

ARGUMENTS	
`view_class`	The class that represents the first view to show in the program.
`width=1280`	The width of the window.
`height=720`	The height of the window.
`title="Arcade Window"`	The string in the title bar of the window.

`set_timer` – Starts a timer that will call a specified function when it finishes.

ARGUMENTS	
`seconds`	The number of seconds to set the timer for.
`function_to_call`	The name of the function to call when time runs out. Do **not** put any parentheses after the function name here.
`args=[]`	An optional list of arguments to pass to the function when it is called.

`show_view` – Changes the current view to the specified view.

ARGUMENTS	
`view_class`	The class that represents the view to switch to.