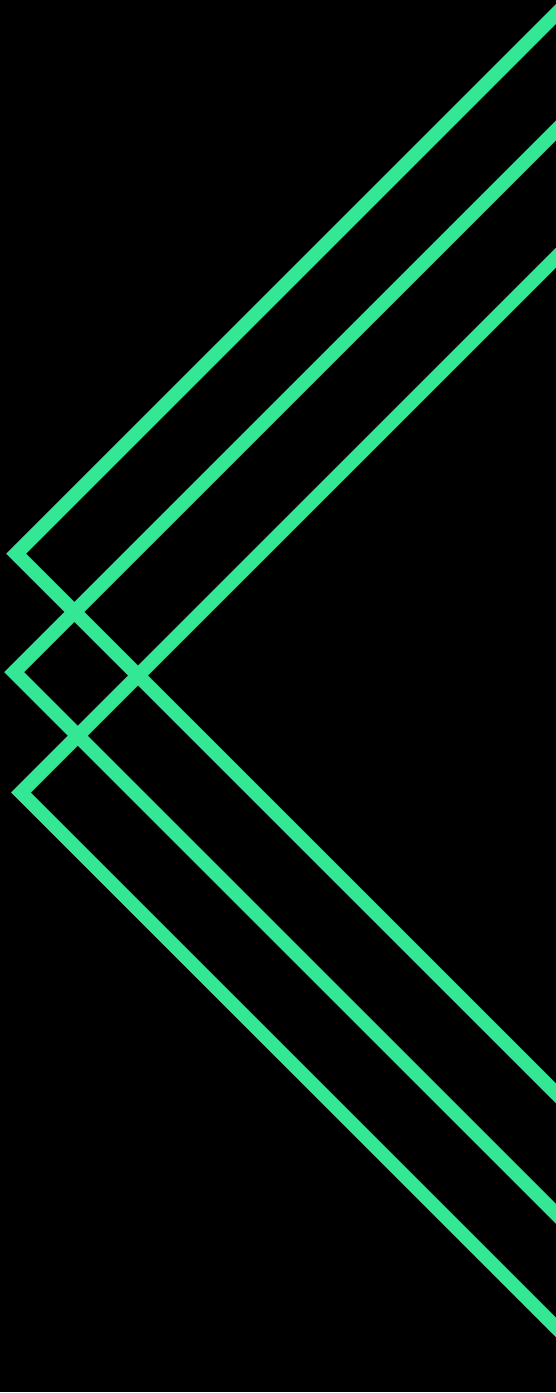



October 2022

# CASTLES AND WAR

PROJECT REPORT



Marie Elyse bassil 503962  
Kiana Naidu 502218

# General information

We have 2 different codes:

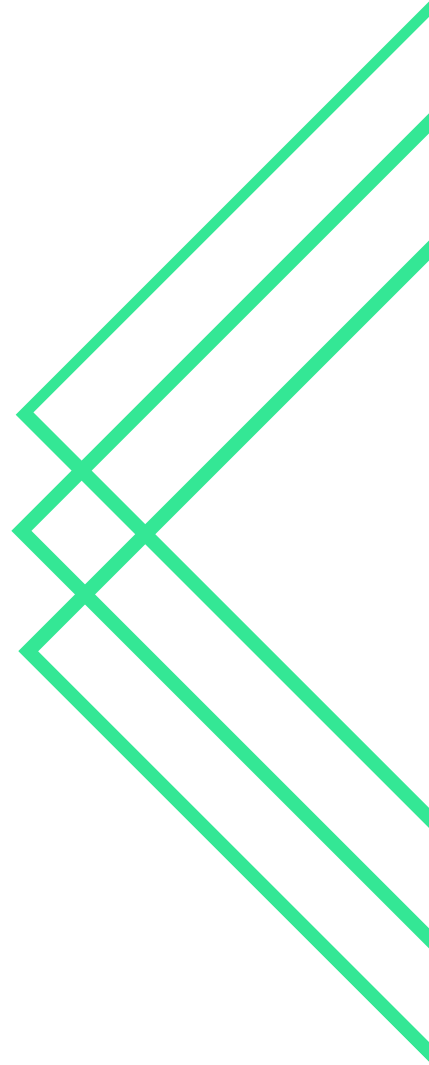
The first one is `object.py` which contains the set of classes to represent the objects in the game.

While the second one, `game.py` is what manages the whole game.

Included in the file, we also have a text file names "commands" which only contains the controls we added in the game and the required keys to press in order to get the game started (create workers, sending them to the mines/walls, releasing them,...).

However, we did not add a code file for the constants provided by the teacher, since we put them in the same code file as the game.

We will now thoroughly discuss both code files and explain them.



# OBJECT.PY

What is the purpose of this code?

It is a class, which is a code template for creating objects and designs, basically to insert everything on the screen (animations, towers, buildings, units,...), which is how we built the design we want for our game.

The following code has 7 classes and each class has its functions:

- 1- Class Animation
- 2- Class Object
- 3- Class Entity
- 4- Class Worker
- 5- Class Swordsman
- 6- Class Archer
- 7- Class Tower

# 1- Class Animation

## Function Init:

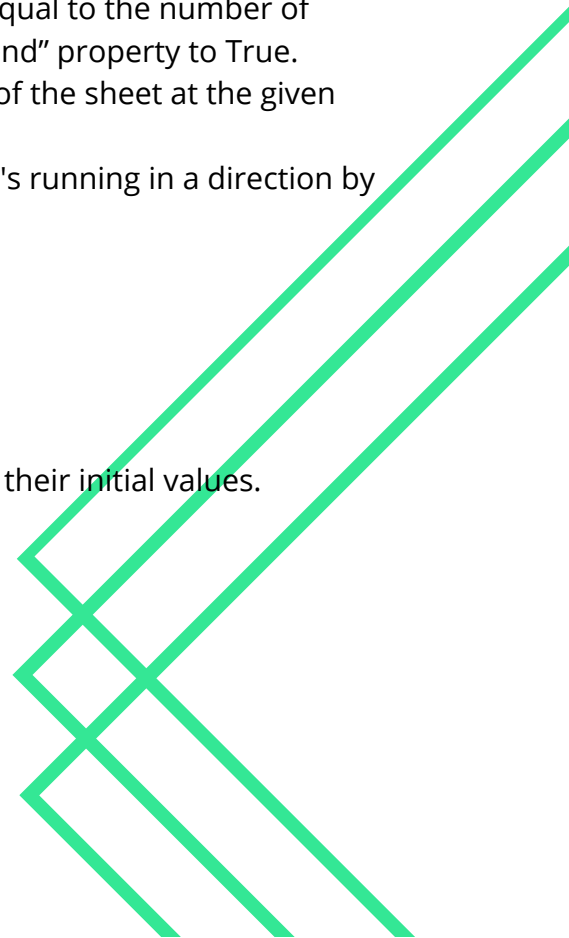
- Started with the "init" constructor to initialize the attributes of the class, precisely to open the animation file and read in the data, then the data was cleaned up and we set the appending to True. Then, we added a path to the sprite sheet from the data that was input, split the data into lines and used the first line as the path to the sprite sheet, got the divisions in the sprite sheet, then the framerate and the frames. then we appended the frames to a list.
- We added the images (corresponding images it must show) through using image.load which allows us to load the image corresponding to this object in the corresponding image files.
- We calculated the rate at which to play the animation, then the number of frames in the animation, divided the animation into the desired number of divisions.
- Finally we set the current frame then the animation end flag, and drew the image using calculated divisions.

## Function Update:

- We set the "anim\_end" property to false, we let the code increment the "current\_frame" property by the game\_rate divided by the number of frames in the animation.
- If the "current\_frame" property is greater than or equal to the number of frames in the animation, the code sets the "anim\_end" property to True.
- Then we set the image variable to the sub surface of the sheet at the given coordinates.
- The function "update" updates the specific unit if it's running in a direction by changing its x and y positions.

## Function Reset:

- To reset the image and current frame variables to their initial values.



## 2- Class Object:

- The code defines this class which inherits from the Sprite class.
- The init function is defined and takes three arguments which are position, anims and scale.
- The position is set to the location where the sprite will be drawn on the screen.
- The anims argument is a list of Animation objects.
- The scale argument sets the size of the sprite.
- The flip\_x and flip\_y arguments determine whether the sprite will be flipped horizontally or vertically.
- Then, we set up the class variables, loaded the image from the file, flipped and scaled the image, then set its position and its rectangle.
- Then, we updated the animation at the current state using the game's framerate, set the surface of the animation to the updated image, set the image of the sprite to the updated surface.
- Finally, we calculated the x position of the sprite based on the game's unit of measurement and the width of the screen and we set the y value for the player object to the screen height minus the player object's height, or the screen height divided by the player object's height multiplied by the player object's y position if the player object's y position is None.

## 3- Class Entity

- The code defines this class which inherits from the Sprite class.
- The entity class is initialized, taking in: position, health, hit, speed and rest variables as well as anims and scale arrays.
- Then the init function is called.

## 4- Class Worker

- First, we defined a class called worker, which inherits from the entity class and we initialised it setting the player parameter to the player object.
- Then, we defined the attributes and methods of the worker class

## 5- Class Swordsman

- Then, we defined a class called swordsman and initialised it setting the player parameter to the player object.
- And defined the attributes and methods of the swordsman class

## 6- Class Archer

- We defined an Archer class, that takes a player argument, which is used to initialize the player's position on the game field.
- The archer class defines several variables, including the player's health, the amount of damage the player has dealt, the player's speed, and how much time the player has until they are exhausted.
- It also defines an animation that will be played when the player is idle, and another animation that will be played when the player is running.

## 7-Class Tower

- We defined a tower class and its constructor takes a player argument, it defines several variables, including the player's position on the game field, the amount of damage the player has dealt to the tower, and how long the tower will take to regenerate.
- It also defines an animation that will be played when the player shoots the tower.

# game.py

## What is the purpose of this code?

This is the code that manages the whole game.

The first thing we did in this code is `import` the `pygame`, `sys`, and `math` modules into the program.

Then we added `"from pygame.locals import *"` which allows us to import all of the constants and variables from the `pygame.locals` module. This allows us to use the `WALL_POS`, `WALL_HEALTH`, `MINE_POS`, and `BARRACKS_POS` constants and variables later in the program.

We initialized the `pygame` module as well as the `font` module.

Finally, we added the constants required, with a comment next to each one explaining what it is.



## Function Main:

Firstly, we imported the "object" module (This is a library of pre-made objects that we can use in our game.)

The "global" keyword means that these variables (FONT, turn, game\_time) are accessible from any part of the game.

The next few lines set up some global variables that we will use throughout the game:

"pygame.display.set\_caption" sets the text that will appear at the top of the screen, "screen" sets up the dimensions of the game window, and "clock" tracks the amount of time that has passed since the game started.

The "adjusted\_unit" variable is a list of two numbers. The first number is the original unit of measurement (inches, pixels, etc.) divided by the width of the game window. This number is then converted to a percentage so that it can be used in the game. The second number in the list is the original unit of measurement divided by the height of the game window.

The "gameover" variable is set to False at the beginning of the game a

Then, we created a group of sprites called "sprites", and then created three objects within that group. The first object is a tower, and the second and third objects are mines. The mine objects are positioned at specific coordinates, and they are also scaled to be twice the size of the other sprites. The barracks objects are also positioned at specific coordinates, and they are also scaled to be twice the size of the other sprites. Finally, 6 portraits are created and added to the group of sprites. These portraits are of workers, swords, and archers.

Finally, we created a list of sprites representing a player's base. For each player, the code sets the player's resources, queue, barracks, and wall. It also sets the number of workers in the player's mine and at the player's wall.



## Function Animation:

Firstly, we created a dictionary called "\_entities" that stores information about each entity in the game. Then, for each sprite in the sprites list, the code checks to see what type of sprite it is. If the sprite is an Archer, Swordsman, or Tower, the code increments its "rest" variable by 1. The code then stores information about the sprite in the "\_entities" dictionary using the round() function to create a unique key for each sprite.

The attack:

The code starts by attacking all units in the game, for each unit, it checks if the unit has been flipped on the x-axis. If the unit has not been flipped on the x-axis, then it checks if the unit is spotted, If the unit is not spotted, then it checks if the unit is an Archer, If the unit is an Archer, then it sets the animation state to 2, It then creates an arrow sprite and adds it to the sprites list.

Then, the code checks whether the arrow's x coordinate is less than the unit's x coordinate plus 1. If it is, the arrow's x coordinate is incremented by 15 divided by the framerate.

The update() function is called to update the screen.

The clock.tick() function is called, which increments the game clock.

The sprites.remove() function is called, which removes the arrow from the sprite list.

The unit.animations[2].reset() function is called, which resets the unit's animation 2. If the unit is a Swordsman, the unit.anim\_state is set to 2.

While the unit's animation 2's anim\_end is False, the update\_screen() function is called and the clock.tick() function is called.

If the unit is a Tower, the unit.anim\_state is set to 1.

Then, the code checks to see if the arrow's x coordinate is less than the unit's x coordinate plus 1, and/or the arrow's y coordinate is less than the unit's y coordinate plus 1. If so, increments the arrow's x coordinate by 15 divided by the framerate, and increments the arrow's y coordinate by 4 divided by the number of frames per second divided by 15.

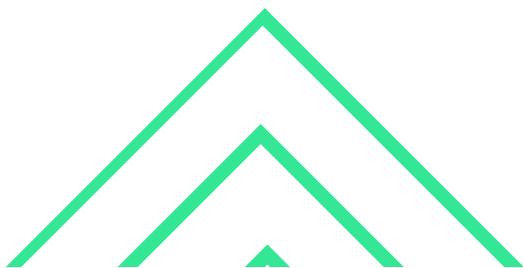
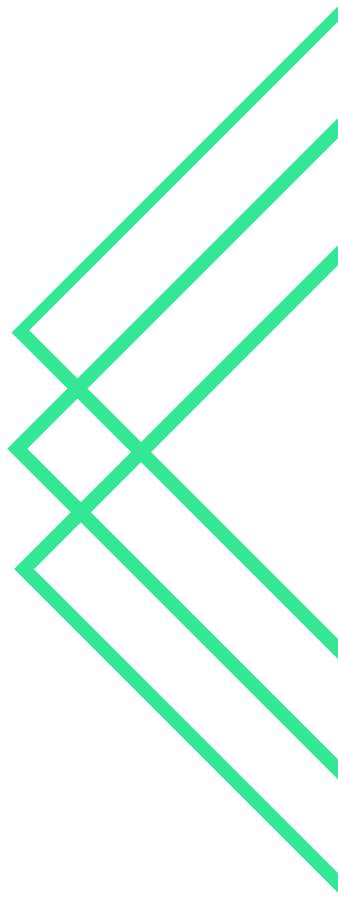
Then it updates the screen, Tickles the clock, Removes the arrow from the sprites list, Resets the unit's animations and Sets the unit's anim\_state to 0

We started with player one.

we made the code iterate through all of the sprites in the sprites list. For every sprite that is an Archer or Swordsman and has its flip\_x property set to False (meaning it is not flipped horizontally), the code stores the sprite's x position in the previous dictionary and sets the unit\_move dictionary to 0 for that sprite. If the sprite has reached its rest position (based on the ARCHER\_REST or SWORD\_REST constants), then the code increments the unit\_move value for that sprite by 1 for each frame of the animation, up to the speed of the sprite. If the loop has gone through all of the sprites and there are still values in the previous dictionary, then the code sets the sprite's anim\_state property to 1 and removes the sprite from the previous dictionary. Then, the code increments the x position of each sprite by the unit\_move value divided by the framerate.

Then, for each sprite that is reversed, the code sets the animation state of the reversed sprites to 0 and pops the previous sprite off the list and sets it to the current sprite then it removes the finished sprite from the list and ticks the clock at a framerate of FRAMERATE. Finally, it updates the screen.

Then, we created a new empty list called "\_entities", that loops through all the sprites in the sprites list. If the sprite is an Archer, Swordsman, or tower, it adds it to the list and updated the screen.



We use a for loop to iterate through all of the entities in the game, and checking each entity to see if it has been flipped on the x-axis. If it has, the code sets a flag to "spotted" and then iterates through a list of all the entities within a certain radius of the current entity, checking each one to see if it has been flipped on the x-axis. If it has, the code sets the unit's anim\_state to 2 (which makes it an "arrow" object) and adds the arrow object to the game's sprite list. then we animate the arrow moving across the screen, while checking to make sure it does not go off the edge of the screen. If it does, the arrow is reset to the beginning of the screen.

Then, we check every frame if the player's arrow is within the bounds of the enemy tower, and if it is, then it subtracts the enemy's health by the arrow's damage. If the enemy's health reaches 0, then the enemy tower is destroyed and the player's unit's health is set to -1.



Then, for player two:

We store the position of every sprite in the game in a dictionary called "previous" and the position of every sprite in the game in a dictionary called "unit\_move".

If a sprite's flip\_x attribute is True, then stores the sprite's position in "previous" and sets "unit\_move" to 0, and If a sprite has rested for the amount of time specified by the ARCHER\_REST or SWORD\_REST variables, then sets "unit\_move" for that sprite to 0.5. For every sprite in the game, calculates how far it has moved by comparing its current position to the position stored in "previous"

If the sprite has moved more than 1 pixel, then increments "unit\_move" by 1 and breaks from the loop if the sprite has moved the amount of pixels specified by the ARCHER\_REST or SWORD\_REST variables.

Then, The code checks if the length of the "previous" list is greater than 0. If it is, the code sets the "finished" list to be an empty list.

Then, the code loops through the "previous" list, setting the "anim\_state" of each sprite to 1. It checks if the x position of the current sprite is less than the x position of the sprite in the "previous" list minus the "unit\_move" for that sprite. If it is, the code adds the sprite to the "finished" list and decreases the x position of the current sprite by the "unit\_move" divided by the framerate.

Then, the code loops through the "finished" list, setting the "anim\_state" of each sprite to 0, removes the sprite from the "finished" list, calls the "tick" function on the "clock" object, with the "framerate" as an argument. Finally, it updates the screen with the update function.

We use a for loop to iterate through all of the entities in the game, and checking each entity to see if it has been flipped on the x-axis. If it has, the code sets a flag to "spotted" and then iterates through a list of all the entities within a certain radius of the current entity, checking each one to see if it has been flipped on the x-axis. If it has, the code sets the unit's anim\_state to 2 (which makes it an "arrow" object) and adds the arrow object to the game's sprite list. then we animate the arrow moving across the screen, while checking to make sure it does not go off the edge of the screen. If it does, the arrow is reset to the beginning of the screen.

Then, we check every frame if the player's arrow is within the bounds of the enemy tower, and if it is, then it subtracts the enemy's health by the arrow's damage. If the enemy's health reaches 0, then the enemy tower is destroyed and the player's unit's health is set to -1.



Then, we set up some text variables that will be used to display information on the screen.

The code renders text to the screen that displays the player's turn number, the resources they have, and the health of the walls as well as the number of workers the player has.

We created text representing the number of units of each type present in the player's barracks, and renders it using a font.

Then, same thing to show the player's resources.

Then, we created a text display that shows the player's turn number, resources, and health. It also displays two worker count indicators.

The `blit` function is used to draw the text on the screen at the calculated position.

Then, we created a strategy game where the player can control two types of units: swordsmen and archers. There are two screens in the game, the first one containing the swordsmen and the second one containing the archers. The player can move the units around on the screen by clicking and dragging. It starts by adjusting the size of the images so that they fit on the screen, then blits the images to the screen in a specific order.

Then, we drew the texts "Mine", " 2 Mine", "Wall", "1 Wall" at certain positions, updated the sprites on the screen and drew them.

Finally, the code defines a variable called `_winner`, which stores the text "Player 1 Wins" or "Player 2 Wins" depending on the outcome of the game. It uses the `pygame.transform.scale()` function to scale the `_winner` variable so it is twice the size of the original text and blits the `_winner` variable to the center of the screen.

Then, we got all events from the event queue and checked if it's a quit event, we quit the game and display a quit message and let it exit the python interpreter.

We added a code to check if the user changed the window size and update everything according to the new window size (new font/screen/display).

Then, we got the keyboard input to assign a keyboard key to an action in the game.

For example, the game checks if the 'q' or 'p' keys are being pressed, If they are, it means the player wants to train a worker, so it does that. If the player has enough resources to train a worker, it takes the resources away from the player and trains a worker. Same thing is done with the other keys.

Then to deploy the worker to the mine, the code checks if a key combination of a and l are pressed together, if yes, the code sets a flag to indicate that a worker has been found. The code then goes through all the units in the player's barracks and checks if the unit type is a worker. If it is, the unit is removed from the barracks and the flag is set to True. If the worker isn't found, a new worker is created and added to the sprites list. The worker's animation state is set to 1 and its position is set to the mine. The code then updates the screen and ticks the clock. If the worker is removed from the sprites list, its position is updated and the player's mine count is incremented.

Finally, to skip turns, we added a code to see if the player has pressed the left or right shift key. If they have, the code will skip over the rest of the code.

And to end the game, the code checks if either of the walls have been destroyed. If they have, the game is over.

The two players compete to gather resources. If a player's resource meter runs out, they lose. The game keeps track of the time, and updates the screen accordingly.