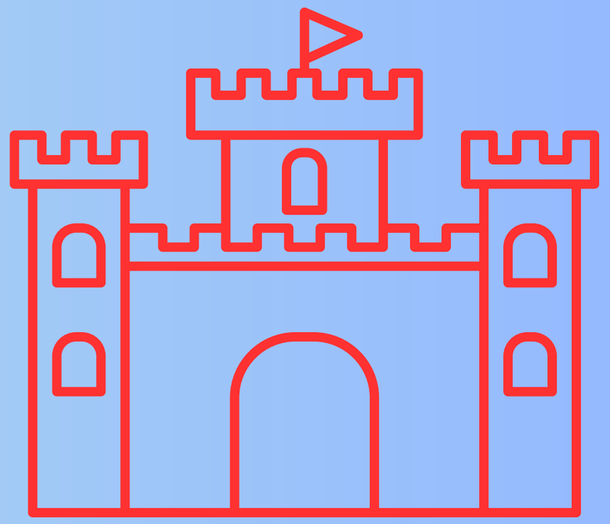
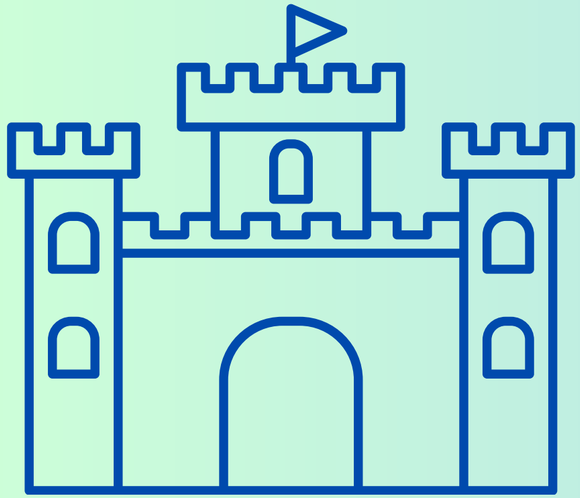


CASTLES WAR

Pagni Angelica - 505887



General Information

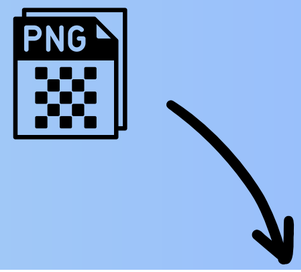
One single ‘main’ file for the code

- Code cohesion
- Faster to find error
- Easier to develop

Watching videos on Youtube

Learning how to use PyGame library

Initial setup

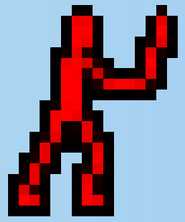


One of the first thing I did was to build the display of the game. I added the background by loading it as a .png image and then I added the title of the game and the logo icon.
After this I wrote all the constants that are essential for defining behavior and properties of various game elements.

```
background_image = pygame.image.load("pxfuel.jpg")
```

```
# Icon and window name
pygame.display.set_caption("CASTLES WAR")
programIcon = pygame.image.load('logo.png')
pygame.display.set_icon(programIcon)
```

Sprites



- Use a list to append the sprites.
- Frame base animation with an index to track the correct png. Use a loop with a counter to display the motion.
- The animation loop can be interrupted when the game is stopped or when the soldier need to change its status.

```
# Soldier 1 (red)
ready_1_soldier = pygame.image.load(os.path.join('sprites', 'player1', 'sword', 'ready.png'))

run_1_soldier = []
run_1_soldier.append(pygame.image.load('sprites/player1/sword/run-1.png'))
run_1_soldier.append(pygame.image.load('sprites/player1/sword/run-2.png'))
run_1_soldier.append(pygame.image.load('sprites/player1/sword/run-3.png'))
run_1_soldier.append(pygame.image.load('sprites/player1/sword/run-4.png'))
run_1_soldier.append(pygame.image.load('sprites/player1/sword/run-5.png'))
run_1_soldier.append(pygame.image.load('sprites/player1/sword/run-6.png'))
run_1_soldier.append(pygame.image.load('sprites/player1/sword/run-7.png'))
run_1_soldier.append(pygame.image.load('sprites/player1/sword/run-8.png'))
run_1_soldier.append(pygame.image.load('sprites/player1/sword/run-9.png'))
run_1_soldier.append(pygame.image.load('sprites/player1/sword/run-10.png'))
```

```
# Connect with the sprites
self.ready = sprites_ready
self.run = sprites_run
self.attack = sprites_attack
self.dead_sprites = sprites_dead
self.image = self.ready
self.tic = 0
self.index = 0
self.time = 0
self.health = SOLDIER_HEALTH
self.rect = self.image.get_rect()
```

Display information

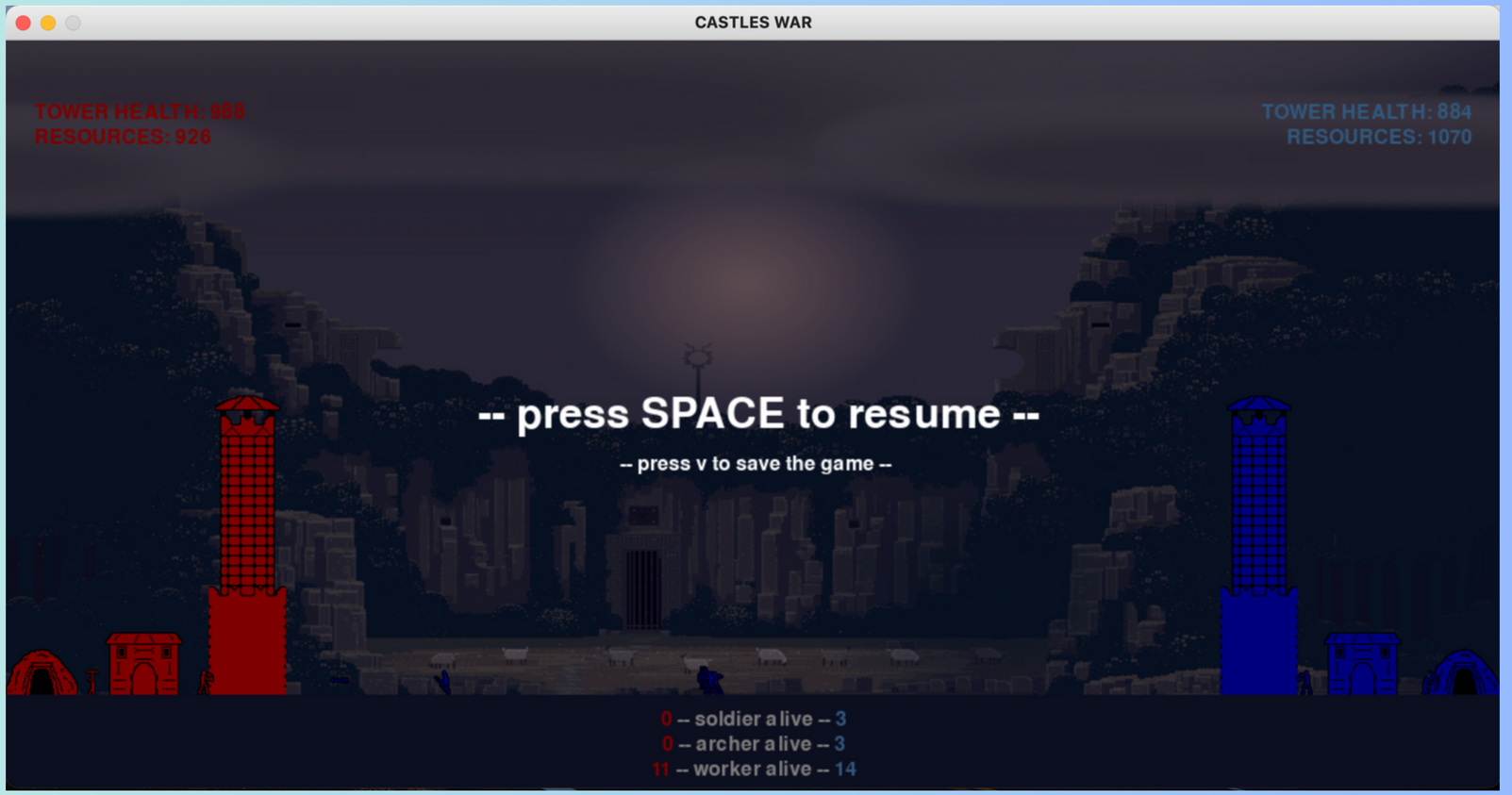
- Amount of resources
- Tower health
- Units alive

Resume command

For the resume command, I wanted to be minimal, so I just put a text ‘press SPACE to resume’ and I made the background semitransparent. The **pygame.SRCALPHA** flag and the RGBA color in the fill method enable the transparency effect.

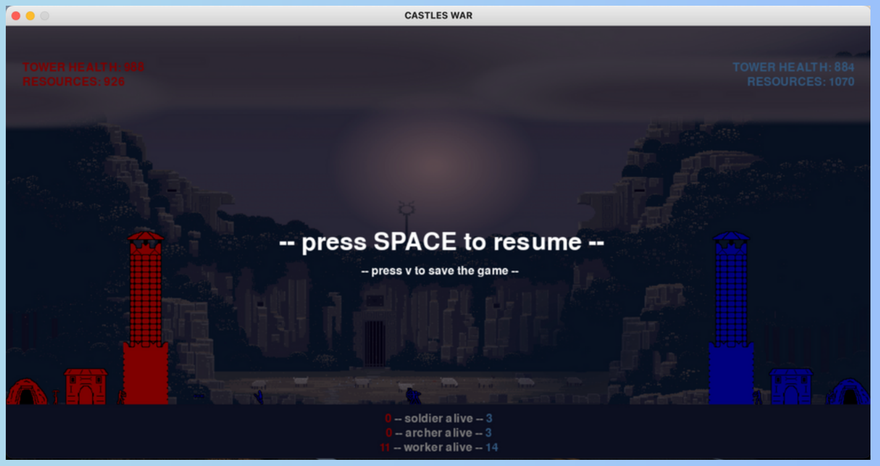
```
overlay = pygame.Surface((WIDTH, HEIGHT), pygame.SRCALPHA)
overlay.fill((0, 0, 0, 128))
```

(0, 0, 0, 128) transparency to make the screen partially visible.

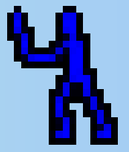


Save and load

The game data, including resources and tower health, is saved in 'save_stats.txt' via **json.dump()**. To load a saved game, players press 'b' in the start game screen. The 'save_stats.txt' file is opened, data is loaded using **json.load()**, and the game state is updated. Players see a 'loading successful' message, allowing them to continue where they left off.

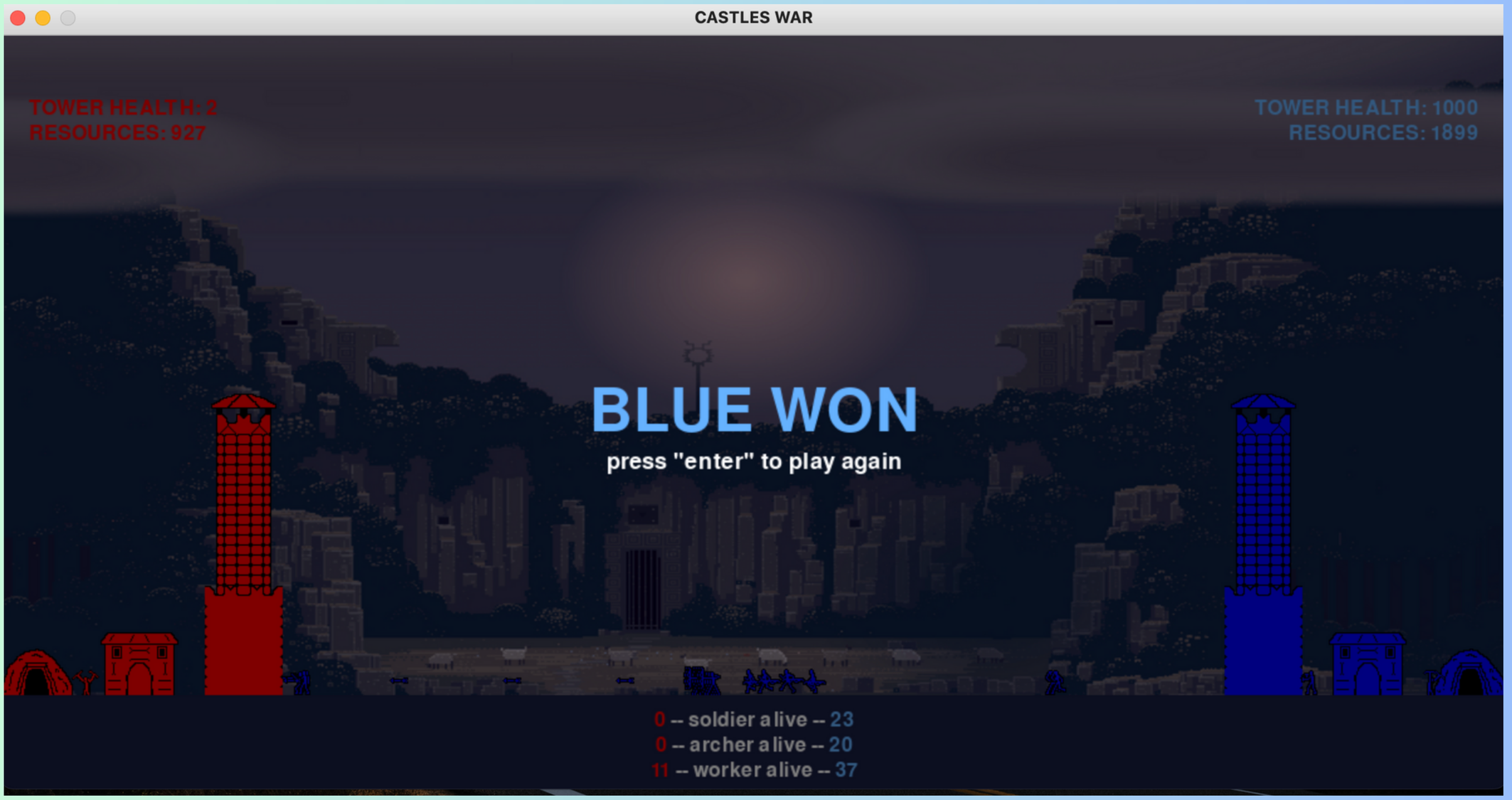


Game over

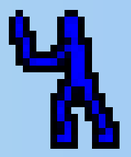


The game ends when one of the tower’s health drop to zero or below. Check the tower health at every enemy hit.

```
def finish_game_red():

    finish_game_text = winning_font.render('RED WON', True, RED)
    finish_game_rect = finish_game_text.get_rect()
    finish_game_rect.center = (WIDTH // 2, HEIGHT // 2)

    play_again_text = font_number1.render('press "enter" to play again', True, "WHITE")
    play_again_rect = play_again_text.get_rect()
    play_again_rect.center = (WIDTH // 2, HEIGHT //2 + 40)

    overlay = pygame.Surface((WIDTH, HEIGHT), pygame.SRCALPHA)
    overlay.fill((0, 0, 0, 128))
```


CASTLES WAR
TOWER HEALTH: 2
RESOURCES: 927
TOWER HEALTH: 1000
RESOURCES: 1899
BLUE WON
press "enter" to play again
0 -- soldier alive -- 23
0 -- archer alive -- 20
11 -- worker alive -- 37

Soldiers class



- **Versatile Units**: soldiers serve dual roles, offense, and defense.
- **Training and Movement:** generated in barracks, soldiers advance toward the enemy castle, engaging in combat upon collision.
- **Player-Dependent Start**: player 1 starts left near their barracks, Player 2 on the right.
- **Initial "Ready" State**: soldiers begin ready and their actions are managed by the update method.
- **Combat Abilities**: soldiers can attack units or tower, changing appearance during combat.
- **Health Monitoring**: health is monitored; reaching zero triggers a defeat animation and removal.
- **Unified Class**: a single class handles soldiers for both Player 1 and Player 2 for code simplicity.

```python
# CLASS 1: SOLDIER
class Soldier(pygame.sprite.Sprite):
    total_soldiers_created = 0
    def __init__(self, player, sprites_ready, sprites_run, sprites_attack, sprites_dead):
        super().__init__()

        self.player = player  # To identify player 1 and player 2
```

Workers

- **Workers' Vital Role**: workers handle resource gathering and tower repair, crucial for gameplay.
- **Player-Specific Start**: player 1's workers originate on the left, while Player 2's start on the right.
- **Initial "Ready" State**: workers begin ready and transition between mining and repairing, each represented by unique animations.
- **Subclasses for Clarity**: to enhance code organization, two subclasses exist: "worker_1" for Player 1 and "worker_2" for Player 2, streamlining code management.

```
class Worker_1(Worker):
    def __init__(self, sprites_ready, sprites_run_mine, sprites_run_tower, sprites_mine, sprites_repair):
        super().__init__(1, sprites_ready, sprites_run_mine, sprites_run_tower, sprites_mine, sprites_repair)

class Worker_2(Worker):
    def __init__(self, sprites_ready, sprites_run_mine, sprites_run_tower, sprites_mine, sprites_repair):
        super().__init__(2, sprites_ready, sprites_run_mine, sprites_run_tower, sprites_mine, sprites_repair)
```

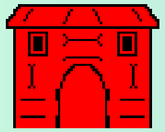
Archers

- **Two Separate Classes**: archers are divided into two separate classes for Player 1 and Player 2.
- **Reason for Separation**: this separation is due to the existence of three distinct archer scenarios, resulting in a more organized and readable code structure.

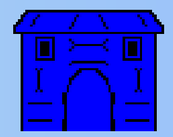
Archer Behavior Scenarios:

- **Unleashed Archer**: when “unleash” is set to True, the archers appears to be running. The movement direction depends on the player (right for Player 1, left for Player 2).
- **Shooting Archer**: if “isshoot” is set to True the archer animates shooting arrows. Animation frames simulate arrow shots with timing controlled by "tic" and "index" variables.
- **Archer's Demise**: the archer is removed from the game if health drops to zero or below.

```python
# possibility 1: unleashed archer
if self.time > ARCHER_TRAIN and self.unleash == True and self.isshoot == False:
    self.tic += 1
    if self.tic == 6:
        self.index += 1
        self.tic = 0
    if self.index >= len(self.run):
        self.index = 0

    self.image = self.run[self.index]

    self.rect.x += ARCHER_SPEED
```




Other classes

Building Classes:

- **Tower_1_red**: represents Player 1's tower.
- **Tower_2_blue**: represents Player 2's tower.
- **Mine_1_red**: represents Player 1's mine for resource generation.
- **Mine_2_blue**: represents Player 2's mine for resource generation.

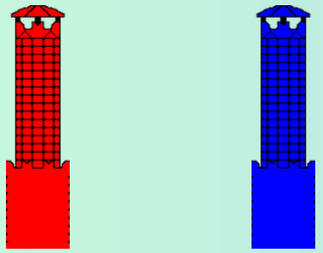
Roles:

- Towers: core structures for protection or destruction, pivotal for victory.
- Mines: generate resources for players.
- Arrows: used for attacks, a fundamental part of combat in the game.

Arrow Classes:

- **Arrow_1**: represents arrows fired by Player 1.
- **Arrow_2**: represents arrows fired by Player 2.
- **Tower_1_arrow_pos_1**: represents arrows fired by Player 1's tower with one trajectory.
- **Tower_1_arrow_pos_2**: represents arrows fired by Player 1's tower with another trajectory.
- **Tower_2_arrow_pos_1**: represents arrows fired by Player 2's tower with one trajectory.
- **Tower_2_arrow_pos_2**: represents arrows fired by Player 2's tower with another trajectory.

To avoid working with both the tower and the wall I decided to create a single image by merging them together.



```
class Tower_1_arrows_pos_1(pygame.sprite.Sprite):
    def __init__(self, tower_1_arrows_pos_1):
        super().__init__()

        self.image = tower_1_arrows_pos_1
        self.rect = self.image.get_rect(left=TOWER_POS, top=HEIGHT - TOWER_HEIGHT)
```

Possible improvements

One thing that I could have done was to add the save and load of the position of the units, but after multiple trials I was only able to do the save and load of the resources.

Conclusion

In conclusion, this project was very challenging for me and understand why the code didn’t work properly was the hardest thing, but through some Youtube videos or tips and advices from my classmates I was able to go through it and adjust it.

I understood that sometimes is better to have a small pieces of code in order to understand it better and in case of an error find it and correct it faster.