**CASTLE WAR PROJECT REPORT**

**Goals of the project**

The goal of the project is to use the pygame library to create a 2d strategy game called ‘Castle War’ . The game involves two players, with the colours re­d and blue represe­nting their respective­ armies and in order to win a player has to take down the enemy castle.

At the beginning of the game, there are 250 resources given to each player which are used to train units.

There are three types of units that are divided into civil and attacking units.

The civil units are made of workers costing 25 resources, they take one turn to train and have the role to either repair the castle or to mine for resources. The attacking units are archers and knights. Archers cost 35 resources, they are long range attackers, they take three turns to train and they are the fastest unit when it comes to their speed while running. Knights cost 50 resources, they are close range attackers, they take three turns to train, just like the archers and they are the attack unit that inflicts the most damage.

As the enemy attacking unit reaches each player's tower’s attacking range, there are arrows that are shot toward the enemy causing damage upon hitting their targets.

Each unit has a respective key for training and unleashing. The keys used to train red units are: “q” for workers, “w” for knights and “e” for archers. Whereas the keys for training blue units are: “p” for workers, “o” for knights and “i” for archers. The keys to unleash red units are: “a” to send workers to mine, “s” to send workers to the wall (to repair the wall), “d” to unleash knights and “f” to unleash archers. The keys to unleash blue units are: “l” to send workers to mine, “k” to send workers to the wall (to repair the wall), “j” to unleash knights and “h” to unleash archers and lastly the key “g” can be pressed by both players to unleash all the units of both teams.

The game is turn based and for each turn the player can choose to either train a unit, unleash a unit or to do nothing. The turn is automatically passes from one player to the other if the player gives their unit a command or if the timer reaches 0 (players have 5 seconds to choose their move).

**Game logic**

My project is divided in six files:

* “castle\_war.py” has the main code,
* “Background.py” contains elements of the background,
* ”SpritesCode.py” contains the sprite classes,
* “Attack.py” handles the attacking logic of the sprites,
* “variables.py” handles variable changes and has some additional functions
* “Save\_game.py” that saves the game state.

The game begins with a home screen controlled by the boolean variable ‘show\_home’. Whenever this variable is true, the home page will be drawn on the screen through the function ‘home\_screen’. The home page includes a background with three rectangles, for aesthetic purposes, the game title and two buttons “New game” and “Load game”.

Whenever the home page is shown, the function ‘init\_variables’ in “Variables.py” initialises the variables that can change during the game, so that if a player goes back to the home page after finishing the game these variables will be automatically initialised to their initial state.

The “load game” button starts the game with the state that it was saved at. This is done by storing the relevant variables in the dictionary ‘data’ through the function ‘store\_data’ (defined in “variables.py”)’. Within this function there is another function called ‘store\_data\_spriteCode’ which specifically handles storing the relevant variables of the file “SpritesCode.py”. The dictionary is then saved in a text file through JSON with ‘save\_data’ (in “Save\_game.py”). When the 'load\_game' variable is set to true, the game data is loaded first by calling 'load\_data()' in "Save\_game.py", which retrieves the previously saved dictionary from the text file. Then the 'getData\_values' function extracts the variable values from the loaded dictionary. Any user can save a game state simply by pressing the key “v”.

The reason why I chose to implement saving the game state in this way, instead of letting the players load data whenever they press a key, is because the player could use that to benefit them. For instance a player could save the game when they are at advantage and then load the data just before losing, thereby returning to the most advantageous position.

After the game officialy starts, several objects are drawn onto the screen through ‘draw\_background’ function in the file “Background.py''. Firstly the background is filled with white. Then the red and blue buildings, a green rectangle at the bottom of the screen (representing the grass) and a light red rectangle are shown on screen. The red rectangle shows on the side of the screen of the red player whenever it is its turn so when it is the blue player's turn the red rectangle disappears and in its stead a blue rectangle shows on the blue player’s side of the screen. Other elements shown on the screen are the resources of each player(’showScores’), the red and blue tower’s health points’(Red\_TowerHealth’, ‘Blue\_TowerHealth’) and two messages. The first one shows that in order to pause the game a user has to press the backspace key and the other one shows that in order to save the game a user has to press the “v” key.

Next in the main loop, there is a section of code that handles the display of the player’s expenditure. This code executes when the variable ‘minus\_points\_displayed’ is set to true and it includes a timer to show the cost for 1 second.

After that, we have the code that shows which unit is training, this is handled by ‘red\_training\_message’ and ‘blue\_training\_message’ in “Building.py”.

If the unit has finished the training, a sprite is added to its respective group. This is handled by the functions ‘add\_red\_unit’ and ‘add\_blue\_unit’ that check whether the units have finished training. If that is the case it adds a sprite to its group.

In the game you can’t train a unit that is already being trained. So if ‘red\_train\_one\_at\_time’ or ‘blue\_train\_one\_at\_time’ is true, a message is shown on the screen to tell the player to choose another unit. A similar thing happens if ‘red\_cost\_message’ or ‘blue\_cost\_message’ is true. The players cannot train units if they don’t have enough resources so those bool variables show that they need more resources.

The fact that you can’t train more than one unit at the same time is the reason why I chose to make both archers and knights train for 3 turns. Indeed, by initially letting the archer train for 2 turns and knights train for 3 turns I noticed that the armies were too similar. For instance, knowing that the worker trains for one turn, each player can choose to train the knight first, the archers second and the worker at last so that all the sprites would appear at the same time at the fourth turn.

As mentioned above, if there are workers that are sent to the mine, additional resources will be added for each turn and there is a countdown implemented to the game. The management of the added resources is performed by ‘show\_added\_resources’ in “Variables.py'' which displays the added resources for 1.5 seconds and adds them to ‘init\_resources’.

The countdown is handled, again, in “Variables.py”, it changes the number shown on the screen for every second that passes and turns ‘red\_turn’ to true or false (depending on whose turn it was) if the counter goes below zero.

Other functions have the following roles: ‘Check\_unit\_ready’ (in “Variables.py”), checks if a certain unit is ready, while ‘draw\_update\_unit’ (in “Background.py”), draws and updates sprites of a certain sprite group.

Next we have the attack mode functions. ‘RedKnight\_attackMode’ and ‘BlueKnight\_attackMode’ in “Attack.py” verify if the knight is resting. If that is not the case, they calculate the distance to enemy units. If the enemy units are in its range it attacks the closest enemy, by setting the ‘attack’ attribute to true. If it’s not in its range it sets the ‘unleash’ attribute to true (indicating running animation). If a knight goes through all the images in the list of the attacking animation it enters the resting state and it waits 600 milliseconds before resuming the attack.

The shooting logic for archers is handled by ‘RedArcher\_shootMode’ and ‘BlueArcher\_shootMode’. These functions’ structure is similar to the one of the knights, except that there are arrows that are added to their sprite group if the archers are shooting and that knights inflict damage when their attacking attribute is true, whereas archers inflict damage when their arrows collide with their targets. The archer resting logic is the same except for the resting time, archers rest for 400 milliseconds.

Lastly, ‘RedTower\_shootMode’ and ‘BlueTower\_shootMode’ control the attacks of the towers. The code structure is similar to that of the archers. The difference between the two is that towers enter the resting state if more than 10 arrows accurately hit their target. Again, the resting code is the same except for the duration (13000 milliseconds)

The arrows from both the archers and the towers are “killed” if they collide with the enemy or if they touch the ground, causing them to disappear from the screen.

To pause the game, as previously mentioned, one of the players can press the backspace key. This puts the boolean variable ‘paused’ to true, halts the game’s logic and displays a message saying that the game is paused.

As stated above, each unit is associated with a respective key that triggers the training process. This is managed by ‘train\_red\_units’ and ‘train\_blue\_units’. The dispatching of units mainly involves setting their ‘unleash’ attribute to true except for the worker which makes distinction of ‘move\_toMine’ and ‘move\_toWall’ to either mine or repair.

The “SpritesCode.py” file mainly contains classes. ‘Worker’ is a superclass containing the animation logic of the worker. Including movement towards the mine, movement towards the wall, digging and lastly repairing. The subclasses that inherit the ‘Worker’ class are ‘RedWorker’ and ‘BlueWorker’, differing only in the images of the sprites and position. ‘AttackUnit’ is another superclass having the logic of running (unleash) and falling animations, as well as the health bar of the sprites. Its subclasses are ‘RedArcher’, ‘BlueArcher’, ‘RedKnight’ and ‘BlueArcher’ each with their own unique images, attacking animation logic, speed and health points.

The penultimate superclass is ‘ArcherArrows’ as the name suggests it controls the archers’ arrows animation. Its subclasses are ‘RedArcher\_Arrows’ and ‘BlueArcher\_Arrows’

The last superclass is ‘TowerArrows’ which contains the logic of the towers’ arrows animation and its subclasses are ‘RedTower\_Arrows’ and ‘BlueTower\_Arrows’.

The reason why the arrows classes are divided into ‘TowerArrows’ and ‘ArcheArrows’ is because they have different moving logic. The archer archers’ arrows only move in the x direction until it reaches the target. However, the tower's arrows have to compute the position of the nearest enemy attacking unit that is in its range and move in both the x and y direction to reach the enemy.

Additionally, the “SpriteCode.py” file also contains the code to manage the health of the red and blue towers. More specifically, ‘Red\_TowerHealth’ and ‘Blue\_TowerHealth’ show the health bar of their respective towers, ‘Tower\_getDamage’ handles damage inflicted to the towers and ‘Tower\_getHealth’ manages the health added to the tower due to workers repairing it.

Lastly, the functions ‘get\_red\_tower\_health’ and ‘ get\_blue\_tower\_health’ retrieve the value of each tower's current health. These values will be used in the main code to check whether one of the players has won the game. In such a case, the boolean variable ‘game\_over’ will be set to True and a text announcing the winner will be displayed on screen alongside a button that allows the users to go back to the home screen.

**Possible improvements**

A possible improvement to this game would be to include a tutorial since the rules are not very straightforward. The tutorial could start automatically after pressing the start button and it could show visually what key each player has to press for a certain command and the normal game should start either right after the tutorial is done or after one of the users presses the button ‘skip’.

**Resources**

<https://www.geeksforgeeks.org/python-display-text-to-pygame-window/>

<https://www.youtube.com/watch?v=POelpwwqxek&list=PLI8CnIMh7oBFN3cx8k_GsBYIRovhKi2gz>

<https://www.programiz.com/python-programming/global-keyword>

<https://www.pygame.org/docs/ref/draw.html?highlight=pygame%20rect#pygame.draw.rect>

<https://www.pygame.org/docs/ref/time.html?highlight=time#module-pygame.time>

<https://www.youtube.com/watch?v=4TfZjhw0J-8>

<http://www.codingwithruss.com/pygame/sprite-class-and-sprite-groups-explained/>

<https://www.reddit.com/r/pygame/comments/13fsyy7/get_sprites_from_group/>

<https://youtu.be/MYaxPa_eZS0>

<https://www.pygame.org/docs/ref/sprite.html>

<https://youtu.be/hDu8mcAlY4E>

<https://www.pygame.org/docs/ref/event.html>

<https://coderslegacy.com/python/pygame-sprite-collision-detection/>

<https://youtu.be/JmpA7TU_0Ms>

<https://api.arcade.academy/en/2.6.0/examples/sprite_bullets_enemy_aims.html>

<https://www.codepile.net/pile/XydlGQy1>

<https://www.pygame.org/docs/ref/sprite.html#pygame.sprite.spritecollide>

<https://www.youtube.com/watch?v=RSl87lqOXDE>

<https://www.youtube.com/watch?v=QuM-jEQ7fAA>

<https://github.com/pickry/programmingknowledge/blob/master/button.py>

<https://www.youtube.com/watch?v=__mZO-53PPM&t=9s>

<https://www.pygame.org/docs/ref/image.html#pygame.image.tostring>

<https://stackoverflow.com/questions/68585252/how-to-check-if-a-global-variable-exists-and-if-not-then-define-it-as-global>

<https://www.geeksforgeeks.org/pygame-flip-the-image/>