BSC AI 2023

CASTLE WAR

Project Exam

UNIVERSITÀ DI PAVIA

UNIVERSITÀ DEGLI STUDI DI MILANO BICOCCA

UNIVERSITÀ DEGLI STUDI DI MILANO

Presented To

PROFESSOR STEFANO FERRARI

Presented By

MATTIA MORO & GIACOMO COLOSIO

INDEX

WHO ARE WE?

Mattia Moro
From Lecco

Giacomo Colosio
From Bergamo

CASTLE WAR

Our startegy game

Castle War

Castel War , created by Giacomo Colosio and Mattia Moro is a video game generated with the python computer language. Specifically based on the use of the Pygame library that allows the creation of games.

Why a strategy game?

The idea of the game that we wanted to follow is clear, aware that other languages allow better video game creations we wanted to make it as entertaining as possible basing the game not only on the graphical aspect but also and above all on the strategic aspect.

BSC ARTIFICIAL INTELLIGENCE 2022/23
CASTLE WAR
10 marzo 2023 Ore 14:30
MATTIA MORO & GIACOMO COLOSIO

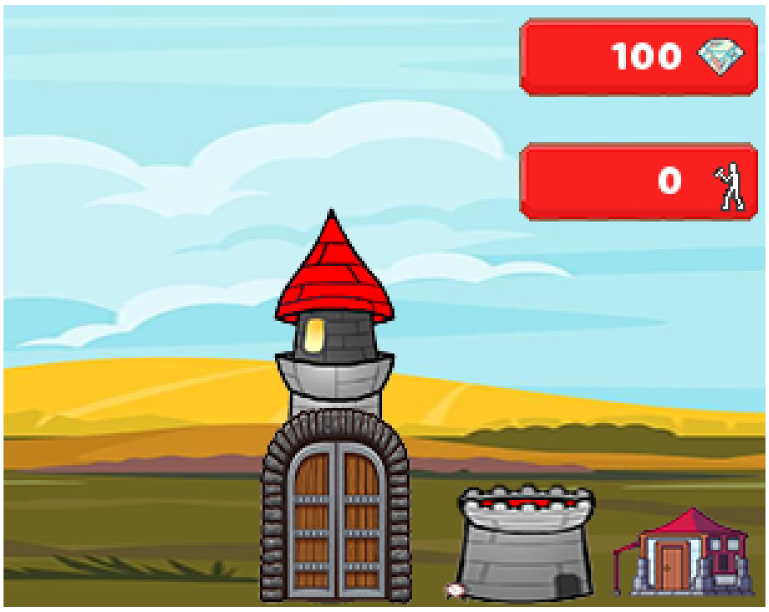
Context

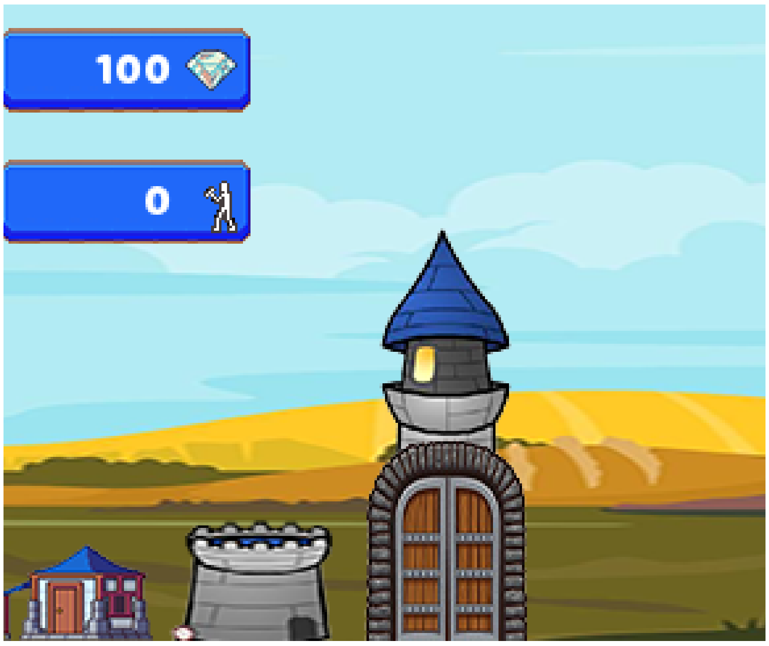
The game is about an invented medieval battle between the kingdom of Bergamo and the kingdom of Lecco, two of the poorest kingdoms of the time so much so that they consisted of only a tower, a castle and a mine. Who will prevail ?

THE PROJECT

Introduction and small description of the project

- The game is played by **two players**, each managing a castle.
- The goal of the game is to destroy the opponent's wall and conquer it's castle.
- Each castle has a wall that protects the inside buildings and units.
- The castle has three main buildings: **a tower, a barracks, and a mine.**

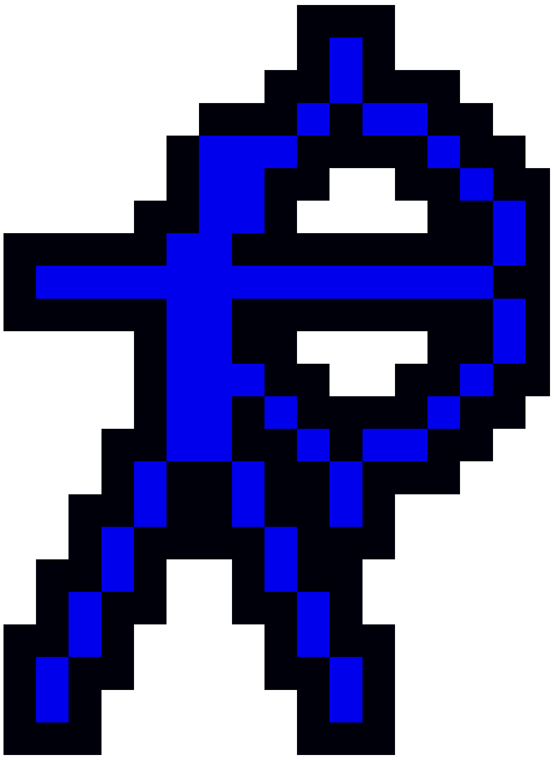




- The **tower can hit attackers, the barracks can train units, and the mine can extract resources**.
- The layout of the game includes a ground plane, where all the elements of the game rest.
- Military units can start to combat after passing the wall.
- Each player starts with a population of zero unit of each type and an initial amount of resources, 100 in our case .

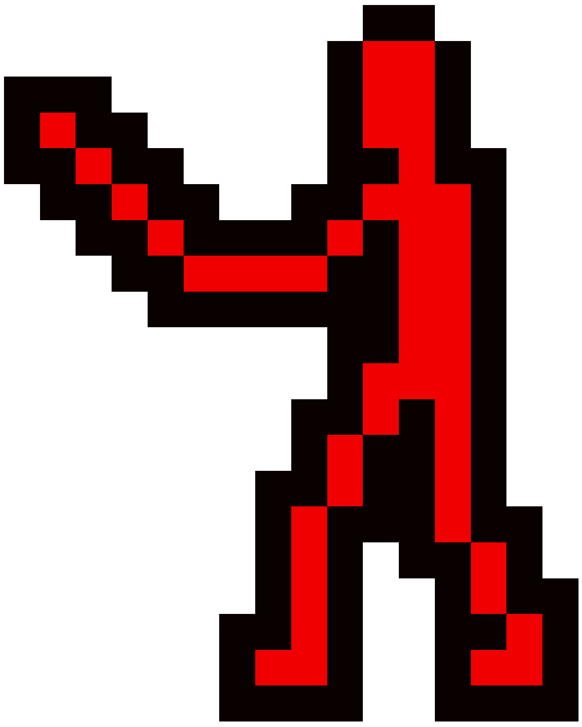
THE PROJECT

Introduction and small description of the project



- In one turn the player has two possible options. The first is to create a character that can be deployed in the next turn. The second is to field a character if it has already been trained previously. The first turn then will be mandatory training of a character.
- The game consists of combat in which different persionages can be used to defeat the enemy tower

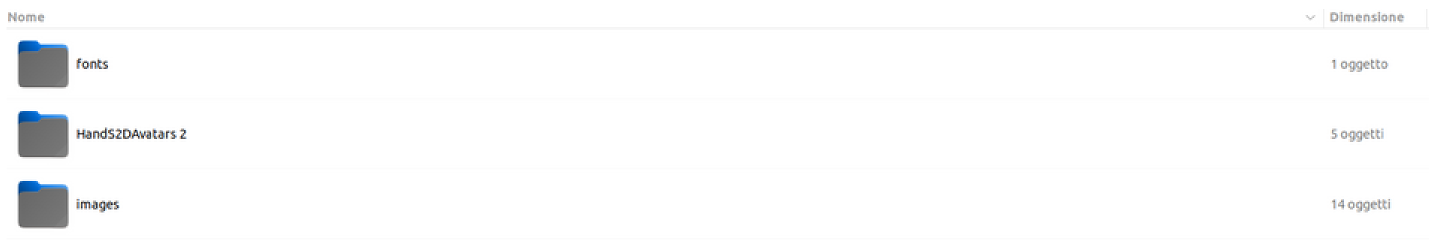
- The game is turn-based, but the rules can be adapted to a real-time game.
- Time is measured in turns, and distance is measured in steps.
- Health is measured in health-points (HP), and an element can have an active role if its health is larger than 0.
- New unities can be added if the player can afford their cost, expressed in a given amount of resources.



ORGANIZATION OF THE CODE

How the code is organize

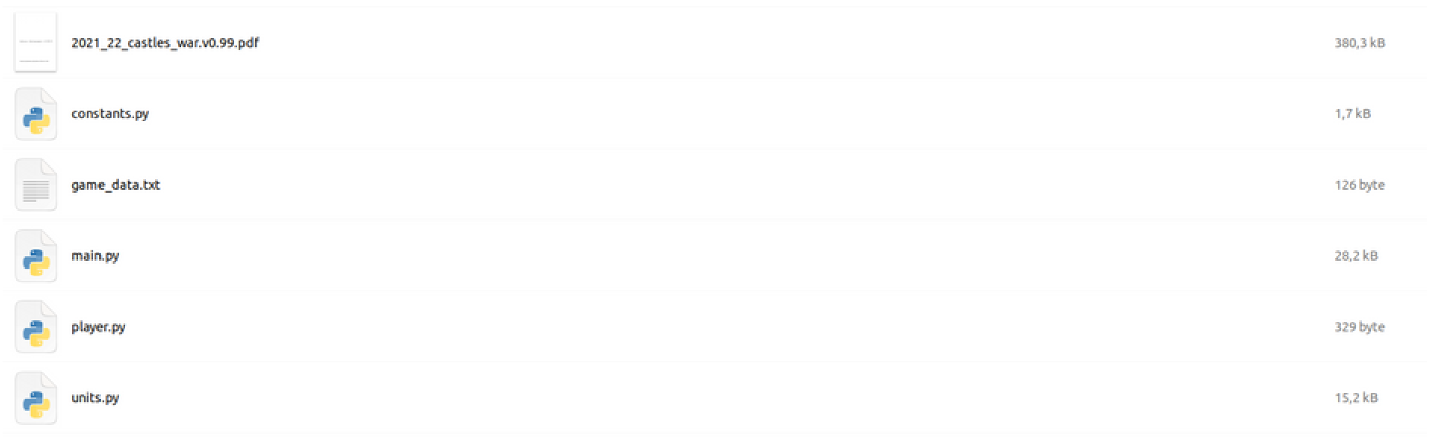
The code is organized in a folder in which the following folder are contained:



Nome	Dimensione
fonts	1 oggetto
HandS2DAvatars 2	5 oggetti
images	14 oggetti

The font folder contains the fold used for the text.
The Image Folder contains all the static image used for the game such as the tower and the background.
The folder HandS2DAvatars2 contains some possible frames necessary for the improvements.

In addition to these folders, these codes are contained:



2021_22_castles_war.v0.99.pdf	380,3 kB
constants.py	1,7 kB
game_data.txt	126 byte
main.py	28,2 kB
player.py	329 byte
units.py	15,2 kB

Constant contains the constants of the game of which we discuss next, in fact one our decisions of the constants are justifiable by thinking about our desire to make this game as much about strategy as possible.
The main is the center of the game; it contains everything necessary for operation and relies on everything in the folder.
Player and units, on the other hand, contain the classes used in the main.
Now we can continue by looking in details this three codes:

ORGANIZATION OF THE CODE

player.py and units.py

player.py

player.py defines a class named "Player". The class has three attributes: "units", "resources", and "wall". The "units" attribute is a list of three integers that represent the number of units the player has.
The first element in the list represents the number of workers, the second represents the number of swordsmen, and the third represents the number of archers.
The "resources" attribute is set to the constant value "INIT_RESOURCES", which is imported from the "constants" module.
This attribute represents the amount of resources the player has. The "wall" attribute is set to the constant value "WALL_HEALTH", which is also imported from the "constants" module. This attribute represents the health of the player's wall.
The "init" method is called when an object of the "Player" class is created. It initializes the attributes of the object to their default values. The "self" parameter refers to the instance of the class that is being created.

units.py

units.py defines a different 5 classes.
The class named "Worker" creates an object representing a worker in the game.
When a worker object is created, it initializes its health to 100, velocity to a constant "WORKER_SPEED," and assigns the provided status and team attributes to the object.
The class also sets the object's image, position, and sprite frames depending on the worker's team.
Two other similar classes define the archer and swordsman All these classes are characterized by methods such as: init and update. Instead, they are uniquely characterized as, for example, the shoot method contained in the archer class
The last two classes, on the other hand, are used to define the archers' bow and the tower projectile. Using the same principle as the previous classes, we created the init and update methods

ORGANIZATION OF THE CODE

main.py

main.py

main.py is the real game and it's divided in different part:

As first thing we imports the following **libraries** and classes: pygame, constants, player, and units. It then initializes the pygame module and creates a Pygame display window. **As a second thing**, we **defined the possible variables** and the font used by taking from the folder we mentioned earlier. Before to strart the real game code then we start our own and real code **we define 3 functions , display, update and commands.**

The function **'display'** starts by displaying the ground with a green color using the fill() method of the win object. It then blits an image representing the game background onto the screen. Next, the function displays two towers on the screen, one on the left side and one on the right side. The towers are represented by images named btower_img and rtower_img, respectively. The function also displays two barracks on the screen, one on the left side and one on the right side. The barracks are represented by images named bbarracks_img and rbarracks_img, respectively. Two mines are also displayed on the screen, one on the left side and one on the right side. The mines are represented by images named bmine_img and rmine_img, respectively. Additionally, the function displays walls on the left and right side of the screen, represented by images named blue_wall_img and red_wall_img, respectively.
The function also displays the number of resources (diamonds) available to each player, as well as the number of workers each player has. These numbers are represented by text rendered using the render() method of the font object, and are displayed on the top left and top right corners of the screen, respectively. Finally, the function displays various game elements, including workers, swords, and archers. These elements are represented by objects in lists named workers_disp, swords_disp, and archers_disp, respectively. The function iterates through these lists and calls the display() method of each object, which displays the object on the screen. If an object's health reaches 0, it is removed from the list.

The function **'update'** updates various aspects of the game state. It starts by declaring global variables related to the current turn, the number of workers for each player, and the amount of time that has passed since the start of the turn. The function then increments the turn counter and adds 50 resources to each player's resource pool.
The function then updates the number of workers for each player by iterating through the workers_disp list, which contains objects representing workers in the game.

ORGANIZATION OF THE CODE

main.py

It then updates the remaining time for each worker, swordsman, and archer unit in their respective lists. If the remaining time for a unit reaches 0, it is removed from the list and its corresponding player's unit count is incremented.
The function then checks the status of each worker unit in workers_disp and updates the resource production or wall repair for the corresponding player based on the unit's status.
Finally, the function updates the remaining time for the player's tower that was previously attacked in the previous turn, switches the turn to the other player, and updates the turn time

The function called **"commands"** handles keyboard inputs for the game.

After the following functions are defined, the actual code of the game is used. Briefl code presents a main loop, which is executed as long as the variable "run" remains True.

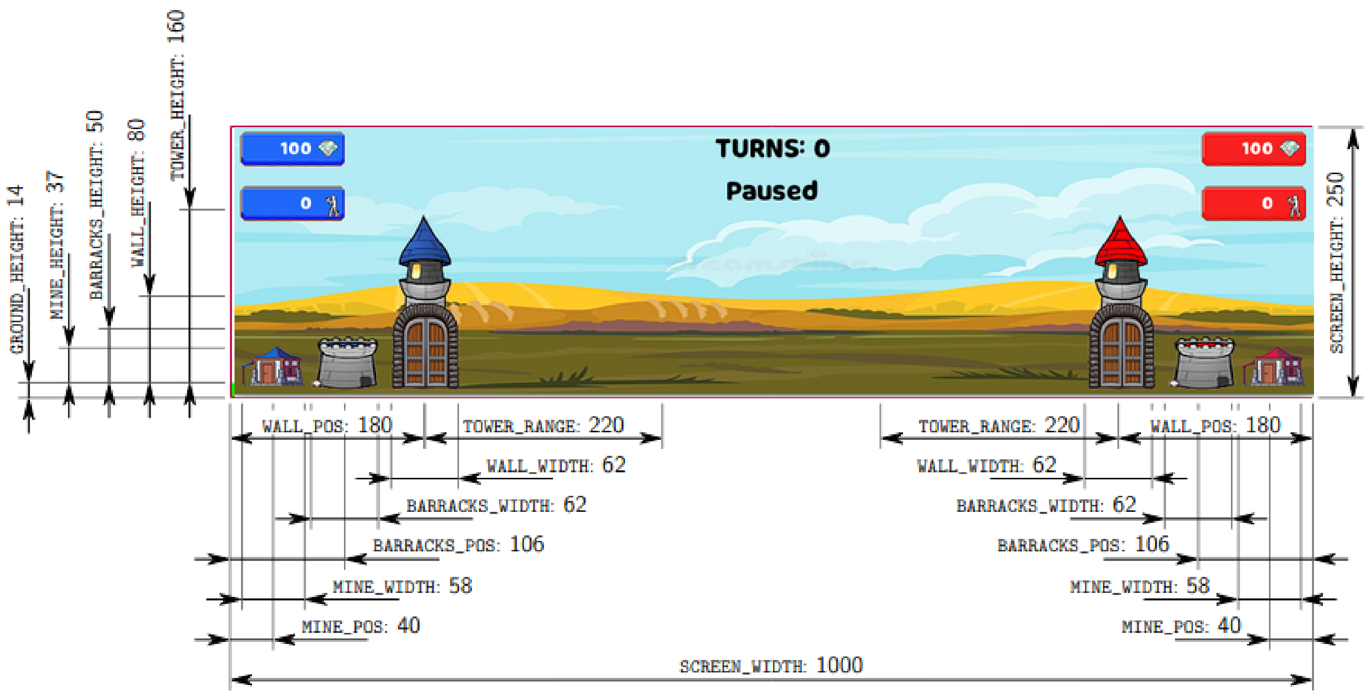
Within the main loop, the following operations are performed:
- The keys pressed by the user via the "pygame.key.get_pressed()" function are recorded.
- A "for" loop is created that cycles over all events generated by Pygame via the "pygame.event.get()" function.
- If the event generated is "QUIT," the value of "run" is set to False.
- A file "game_data.txt" could be created with 'v' command, doing this information about the game that has just finished will be written. The following data is then saved: the number of turns, the status of the wall and resources of the two players, the status of the game objects (worker, swordsman, archer) for both players.
- It is checked whether the "SPACE" key has been pressed. If yes, the execution of the game is stopped.
- It is checked whether the wall of the two players is still standing. If yes, the following operations are performed: the screen is wiped, the current state of the game (position of objects, scores, etc.) is drawn, the current number of turns and which team's turn is displayed, any commands entered by the user are checked, archer and tower attacks are handled.
- If the main cycle is interrupted, the game is closed via the "pygame.quit()" function.
- It is also possible to load the data saved in the text document used for saving .

ASSUMPTIONS, METHODS, PROCEDURES

Motivations of the design choices

The gameplay interface

The game screen is a box of size : **(250X1000)** structured as follows:



For the construction of the game, we empathized with a possible clash between **Lecco**, the blue troops taking their color from the flag of Lecco and the troops of Bergamo , the red ones also taking their color from the flag of **Bergamo.**

The **value of the constants used are those given in the project instructions**. Instead, we made some changes in the playability following our preferences for a possible game. The background, on the other hand, is taken from the web and is in theme with the hilly nature type of Bergamo, the attached city. At the top of the screen we have two rectangles that respectively indicate the **number of resources ,** denoted by the number of diamonds, **and the number of troops**, which corresponds to the sum of all the troops alerted during the game.

As a first thing, **we have chosen not to decrease the number of troops after deploying them** because one difficulty of the game will have to be precisely to remember the number of one's own troops and of the opponent. Also as a second choice **we decided to have a box counting all types of characters and not one per category** so as not to allow players to copy each other's moves.

ASSUMPTIONS, METHODS, PROCEDURES

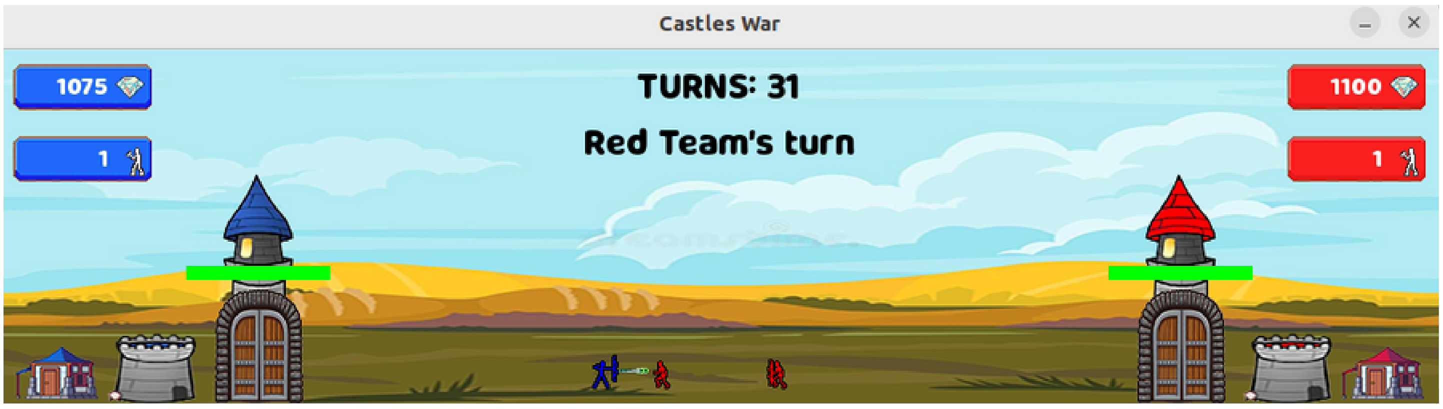
Motivations of the features choices

The Rounds

The game as mentioned above was created by us with the intention of making it fun under a logical and strategic point of view. Therefore, **our game is divided into 5-second rounds in which different decisions can be made**. To complicate the game you have that as the first thing in a turn you can only train a troop or send it to fight. As the second thing instead it is also time that complicates things given the need to respond with immediacy the move of the a

Damage and life points

Unlike a game of chess, **the game is programmed not to have particularly long games, and because of this the characters move fairly quickly and the life of the buildings is not very high**. In fact following what we said in the previous paragraph the game asks for responsiveness and promptness in responses. Because of this, characters do a lot of damage, so it is essential to be able to defend your tower. As an aid the protection of the tower, that is, the spiked ball we have inserted is very solid.The only thing to last slightly longer is the clash between the different types of characters , this in fact allows the player to have some time to pause and reason, this can be more useful especially to the less experienced. So here is what the game dynamics look like during a fight:

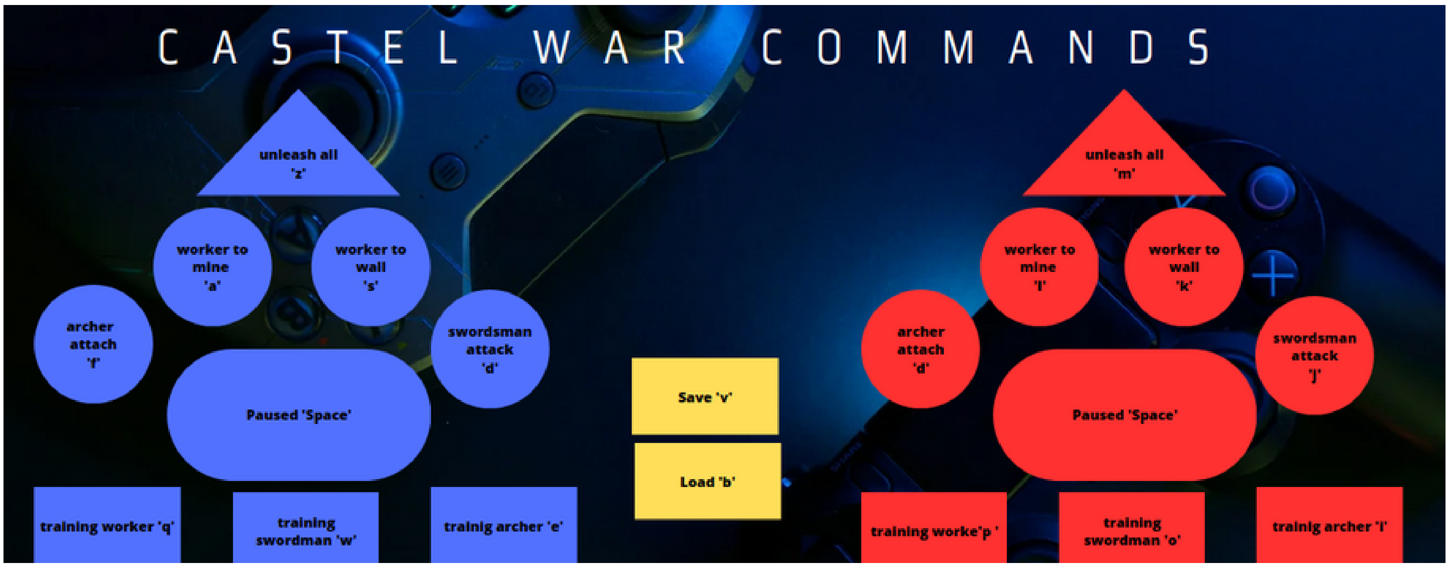


ASSUMPTIONS, METHODS, PROCEDURES

Motivations of the features choices

Command

In the **Command phase,** the players input is evaluated and the status of the units and buildings is changed accordingly. In our game dynamics, a player is able to inpu one command per turn. The command are fulfilled all before passing to the next phase. The following commands are used.



Combat

The Combat phase in a game or simulation involves each unit having the ability to move and either deliver or receive an attack. Units with positive health values can participate, and their actions are evaluated in a random order. Changes in unit status do not take effect until the end of the phase, so a unit can still operate even if it loses all its HP during combat. A temporary counter should is used to collect injuries for each unit, which will be applied simultaneously at the end of the phase. The Combat phase consists of moving units, fighting, and amending unit status. Towers and walls are considered units as well. Units must choose a target based on rules such as attack range, proximity, and weakness. The evaluation of unit actions should be random to avoid bias

TOOLS AND RESOURCES

Python and The pygame lybrary

Python

IPython is a dynamic, interpreted (bytecode-compiled) language. There are no type declarations of variables, parameters, functions, or methods in source code. This makes the code short and flexible, and you lose the compile-time type checking of the source code.

A Python library is a collection of related modules. It contains bundles of code that can be used repeatedly in different programs. It makes Python Programming simpler and convenient for the programmer. In this project the main library that we had used is pygame.

Pygame

Pygame is a popular Python library widely used for developing games, multimedia applications, and graphical user interfaces. It was first published by Pete Shinners in 2000 as an alternative to the PySDL library. Since then, pygame has become a developer favorite for its ease of use and versatility. One of Pygame's main strengths is its ability to handle both 2D and 3D graphics.

In our case pygame is the base for our 2D castle war and in particular we have used to show our ability to treates problem to obtain an optimal solution using python language merged with the object oriented programming

POSSIBLE IMPROVEMENTS

Castel war 2: Bergamo's revenge

The game we proposed has merits, in fact it is very fun and functional, the graphics as far as the pygame library allows is nice and yet some changes can be made for a possible second version, ironically called : **Castel war 2: Bergamo's revenge.**

Game menu

A first possibility would be to **add a menu**, as in an old classic game in which inspired by many games seen over the years might contain:

As a first step, one could have a section that allows one to choose the color (and thus the city) of player 1 and player 2. to do this one would need to have several images with the different colors and have the chosen color used in the code via a variable. It could be also possible change background changing photo.

Another possible improvement would be to add the ability to have an automatic player called a CPU (computer player), this part could be do in two ways, using an reinforncmet learning or using preset moves.

One 'last thing we could add is the possibility of naming different saves of different games and then reusing them in the future when needed. That way you would not have one save but several.

Other possible improvements

Enhancements: Adding new power-ups such as weapons, items, special abilities, and character enhancements can increase player involvement and make the player feel more powerful.

Bonus challenges: Adding bonus challenges such as mini-games or timed levels can entertain the player and increase the game's longevity.

Additional characters: Adding playable characters with unique abilities can give the game a variety of playability and involvement.

Improved graphics: Adding details to images and animations can make the game feel more modern and attractive.

New music and sound effects: Adding new music and sound effects can make the game experience more immersive and engaging.

UNIVERSITÀ DI PAVIA

BICOCCA

UNIVERSITÀ DEGLI STUDI DI MILANO