Università degli Studi di Pavia

Bachelor degree in Artificial Intelligence

Castles war — Final exam project — A.Y. 2021/22

STEFANO FERRARI

Computer programming, Algorithms and Data str., Mod. 1

Contents

date: January 25, 2022
version: 1.0

The final exam project consists in the design and implementation of a strategy game named "Castles war". The game is intended to be a 2D side-view game, implemented using the library `pygame` [1] (see Sect. 9.1). In order to allow some flexibility in the design, the specifications below described will refer to the game parameter through suitable constants, summarized in Sect. 7. The game is intended to be played by two players interacting through the same keyboard and the same video display. Optionally, the project can consider to substitute a player (or both) with an automatic (programmed) player.

1. Rules

The game is played by two players, each of which manages a castle. The castle has a wall that protects the inside buildings and units. The goal of the game is to destroy the opponent wall (and hence conquer the castle).
Each castle has the following buildings:

- a tower, which can hit the attackers;
- a barracks, which can train units;
- a mine, from which workers can extract resources.

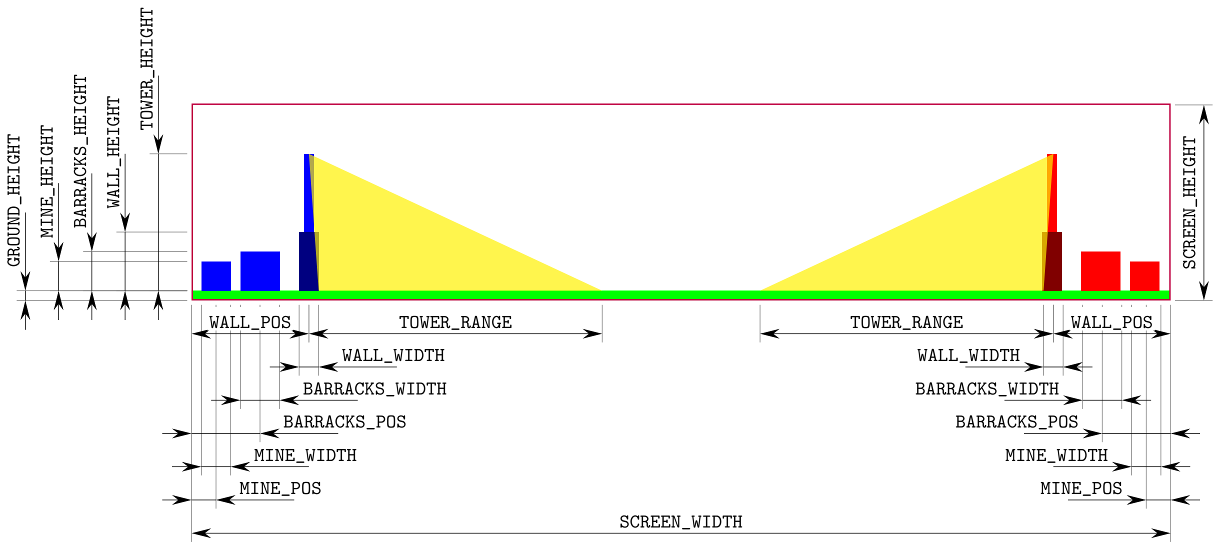

Figure 1: Reference schema for the game playground. The dimension are referred as the geometrical constants (see Sect. 7).

The layout of the structural elements of the game is depicted in Fig. 2. The castles are positioned to the opposite extremes of the screen. The military units can start to combat after passing the wall (`WALL_POS`). All the structural elements rest on the ground, which is an horizontal plain layer at the bottom of the screen, up to `GROUND_HEIGHT` from the lower side of the screen. This is also the vertical level at which the units move.

Each player starts the game with a population of one unit of each type and an initial amount of resources (`INIT_RESOURCES`).

The game is turn-based (with a simultaneous resolution of the turns), although the turn can be so short in time that the rules can be adapted to a real-time game.

The game world is characterized by the following measurement units:

Time The time is measured in *turns*. Depending on the game dynamics (i.e., real-time v.s. turn-based) more than one frame per turn can be considered. For a real-time game dynamics, the equivalence between the duration time of one frame and one turn can be considered. However, for turn duration longer than 1 s, several frames can be visualized.

Distance The distance is measured in *steps*. For graphical purposes, the equivalence between steps and pixels can be considered, for the sake of simplicity.

Health Each element in the game has a property called *health* measured in *health-points* (HP). An element can have an active role if its health is larger than 0.

Cost New unities can be added to the game if the player can afford their cost, expressed in a given amount of *resources*. This is a general name for goods that can be implemented in the game. For instance, they can be gems, coins, food (in the last case, the mine building can become a farm).

Each turn consists in the following phases:

command in which the player can issue an order of the following:

- train unit;
- dispatch unit;
- unleash all the military units;

combat in which the elements of the game are updated using the rules specified in the following;

work in which the buildings properties are updated using the rules specified in the following.

A more detailed description is in Sect. 6.

2. Units

There are three types of units:

Worker Civil unit that can dispatched to the mine to extract resources or to the wall to restore it.

Swordsman Military unit that can attack adversary units and wall with a short range attack.

Archer Military unit that can attack adversary units and wall with a long range attack.

Each unit has a behavior and ability ruled by its parameters, further explained in the following.

2.1. Worker

A new worker can be generated by the barracks using `WORKER_COST` resources. The generation of a worker requires `WORKER_TRAIN` turns to be completed. Once created, the worker waits in the barracks until it is dispatched to the mine or to the wall. The worker moves at a speed of `WORKER_SPEED`. When it is working in the mine, it generates `WORKER_PROD` resources per turn. The produced resources sum to the accumulated resources of the player and can be used in the following turns. The accumulated resources do not expires and are unlimited. However, limits to the resources can be optionally added in order to increase the complexity of the game. When the worker is at the wall, it can restore `WORKER_REPAIR` HP of the wall per turn.

2.2. Swordsman

A new swordsman can be generated by the barracks using `SWORD_COST` resources. The generation of a swordsman requires `SWORD_TRAIN` turns to be completed. Once created, the swordsman waits in the barracks until it is released to attack the adversary castle. The swordsman moves at a speed of `SWORD_SPEED` until it reaches an adversary unit (or the wall) in its range, i.e., when the distance from the adversary unit is no larger than `SWORD_RANGE`. During the combat, a swordsman delivers a damage of `SWORD_HIT` to the target. After having hit a target, the swordsman is inactive for `SWORD_REST` turns. During the combat, the swordsman can also receive a damage that will decrease its health, which is initially `SWORD_HEALTH` HP.

2.3. Archer

A new archer can be generated by the barracks using `ARCHER_COST` resources. The generation of an archer requires `ARCHER_TRAIN` turns to be completed. Once created, the archer waits in the barracks until it is released to attack the adversary castle. The archer moves at a speed of `ARCHER_SPEED` until it reaches an adversary unit (or the wall) in its range, i.e., when the distance from the adversary unit is no larger than `ARCHER_RANGE`. During the combat, an archer shoots an arrow which delivers a damage of `ARCHER_HIT` to the target. The arrow travels at a speed of `ARROW_SPEED` until it reaches the adversary (or go out the game world). If the target unit dies before the arrow can reach it, the arrow will continue to move along its trajectory, until it its another adversary target or reaches the limit of the playground. After having shoot an arrow, the archer is inactive for `ARCHER_REST` turns. During the combat, the archer can also receive a damage that will decrease its health, which is initially `ARCHER_HEALTH` HP.

3. Buildings

The buildings in the castle are the following:

Wall It is the protection of the castle. Initially, it has a maximum health of `WALL_HEALTH`, that can decrease when hit by an adversary unit. The wall health can be restored by the workers up to `WALL_HEALTH`. No restoring cost is specifically required, but it could be an option for adding complexity to the game. The restored HP per turn is not limited and the wall can host an unlimited number of worker for the repair. Optionally these factors can be limited, to add complexity to the game. When the wall health reaches 0, it is destroyed and the game is over. The winner is the adversary (unless in the rare case in which also the adversary wall is destroyed in the same turn).

Barracks It can train new units and hold them until a dispatching command is given. No maximum number of units is prescribed, but it could be an option for adding complexity to the game.

Mine It allows to obtain resources through the activity of the workers. The mine has an unlimited deposit from which the resources can extracted at an unlimited rate, and can host an unlimited number of workers. Also for these parameters, a maximum can be optionally defined, to add complexity to the game.

Tower The default defense of the castle. It can shoot arrows to the adversary units closer than `TOWER_RANGE`. During the combat, the tower shoots an arrow which delivers a damage of `TOWER_HIT` to the target. The arrow travels at a speed of `ARROW_SPEED` until it reaches the adversary (or the ground). Differently from an arrow shoot by an archer, a tower arrow has a direction, which depends on the rule defined to aim to a target and that cannot be changed after the arrow has been shoot. After having shoot an arrow, the tower is inactive for `TOWER_REST` turns.

4. Arrows

The arrows are intended to travel using a linear trajectory. Hence, the arrows shoot by archers will travel horizontally, while the arrows shoot by the towers will travel diagonally (from the tip of the tower to the ground). Once shoot, an arrow moves along its trajectory until it reaches an enemy target (a unit or the wall) or the playground limits (the ground or the limits of the screen, although this second case should never happen). While for arrows shot by archers the hit with the adversary target can be defined as when the arrow (horizontal) position match or exceeds the (horizontal) target position, for a tower arrow, the hit rule should more carefully defined. In fact, after being shot, the arrow cannot change its trajectory and the target can move, resulting in three possible situations:

1. the target is still for all the duration of the arrow travel: the horizontal positions matching rule can be used;
2. the target moves and the arrow will never cross it: the arrow will miss the target continue its trajectory up to hitting the ground or another target;
3. the target moves and the arrow will cross it: the arrow hit the target.

The definition of a suitable rule for crossing the target is a design choice. A simple acceptable solution could be evaluating the collision between the sprites.

5. Commands

Each player can control the game through the following actions:

Train This command is issued to the barracks in order to start the generation of a new unit of a given type. If the player has enough resources to the aim, the resources are decreased by the amount needed and the train can start. Otherwise, the command is ignored (and eventually reported). If the barracks is already training a unit when a new training command is issued, the new unit training will start at the first turn available. The barracks, hence, should maintain an unlimited queue of orders and fulfill them in sequence. Optionally, the design can consider the rejection of new training orders when the barracks is busy (in this case, a new unit could be trained only when both the resources and the barracks are available). Once trained, the unit stay in the barracks until a dispatching command is issued.

The commands are issued by pressing the following keys:

	Player 1	Player 2
Worker	`'q'`	`'p'`
Swordsman	`'w'`	`'o'`
Archer	`'e'`	`'i'`

Dispatch This command is issued to change the action of a unit. When a worker is dispatched to a building (mine or wall), the command is issued first to the workers in the barracks. If no workers in barracks are available, a worker availability in the other building is checked. If positive, the worker is moved in the target building, otherwise the command is ignored (and eventually reported). Similarly for the military units, their availability is checked and in case the order is fulfilled, otherwise the command is ignored. The "unleash all" command is applied to all the military units in the barracks, which are sent to combat.

The commands are issued by pressing the following keys:

	Player 1	Player 2
Worker to mine	`'a'`	`'l'`
Worker to wall	`'s'`	`'k'`
Swordsman attack	`'d'`	`'j'`
Archer attack	`'f'`	`'h'`
Unleash all	`'z'`	`'m'`

Pause The Pause command temporary stops the execution of the game. The command is issued by pressing the space bar. The game can continue when the space bar is pressed again. Both the command can be issued by any of the players.

Save The Save command stores on a file the game status, using the format given in Sect. 8. Optionally, the program can support multiple saves, either by proposing the choice among a redefined number of slots, or asking a name for the game save. Besides, the program may consider the option of auto-save on a redefined slot or file. This command can be issued by any player, by pressing the `'V'` key (`Shift-v`).

Load The Load command reads the initial game status, using the format given in Sect. 8. The current game status will be reset and reinitialized to the one loaded. Optionally, the program can propose a list of saved games from which to choose the game to be loaded. The game will start in a Pause status and the operation will start when the space bar is pressed. This command can be issued by any player, by pressing the `'B'` key (`Shift-b`).

6. Game dynamics

Although some game dynamics details can be subject to the design choices, some common rules should be considered.

Each turn, as shortly specified in Sect. 1, is composed of several phases: *command*, *combat*, and *work*. Each phase should consider all the units, not differentiating between the players. However, to (optionally) make the game dynamics to turn-based, the possibility to actively participate to the game can be alternatively assigned to only one of the players. The following rules are intended fora real-time dynamics game.

6.1. Command

s In the Command phase, the players input is evaluated and the status of the units and buildings is changed accordingly. Depending on the game dynamics, a player could be able to input more than one command per turn. The command can be fulfilled all before passing to the next phase, or only one per turn (or a maximum number per turn). This is a design choice.

6.2. Combat

In the Combat phase, each unit can move, deliver or receive an attack. The units that has a positive value of health (positive amount of HP) can actively operate in this phase. The order of evaluation of the actions can be random, but the principle is that no change in the status of the units will become effective until the end of this phase. Hence, for instance, a unit having positive value of health at the beginning of the turn can operate its action although it receives injuries that deplete all of its HP during the current combat phase (from units evaluated before it). For this reason, a unit should have a temporary counter to collect all the injuries, that will be applied simultaneously at the end of the combat phase. Hence, the Combat phase can be structured as:

1. *move* the units, in which the units are moved (workers toward the target building, military units toward the adversary units);
2. *fight*, in which the military units choose a target and deliver injuries, according the the combat rules for the unit;
3. *amend*, in which the military units status are updated.

In this phase, the towers and the walls can be considered as units.

When dispatched to attack, military units leaves the barracks and move toward the adversary castle. To engage a fight, they have to pass their wall (`WALL_POS`). After that point, they can engage if there is an adversary unit in their attack range, and cannot move forward until the adversary units are alive. The principle is that the most advanced unit of each player cannot pass the most advanced adversary unit. Units of the same player, instead, do not block each other, and a unit (say, a swordsman) can pass another (say, an archer) that is engaged in combat, until an adversary unit enter in its attack range. Hence, since the attack range is the same for all the units of a given type, all of them will reach the same position, in presence of an adversary unit (or the wall). This poses no problem in the combat dynamics, but, due to the 2D side-view, make the visualization more complex, since a player cannot understand how many units the adversary has on the field. For this reason, a visual hint is strongly suggested. This can be in form of text over the units, showing number and health, possibly cumulatively for groups of units sharing the same position.

A unit should choose a target applying hierarchically the following rules:

1. those in the attack range;
2. the closer;
3. the weaker (i.e., the one that can be killed in less turns).

Optionally, other rules can be stated.

The evaluation of the units actions should be random (to avoid biases). In the choice of the target, a unit can consider the parameters of the adversary units: position, health and injuries.

6.3. Work

In Work phase, the workers operate and the status of the amount of resources and health of the wall are updated. The behavior of the worker when the wall is completely restored can be customized. One choice is leave the workers in the building, waiting for a dispatching command. Otherwise, the worker could return automatically to the mine, possibly after waiting for a given number of turns, as a cautionary measure (for instance, the double of `SWORD_REST`).

At the end of the turn, the status of the walls is checked and in case a winner is declared. Otherwise, the next turn takes place.

Using the above rules, a unit could in the same turn move (if not close enough to an adversary unit) and attack (if after the move phase the units become close enough). This is not a problem, but to avoid it, optionally, a temporary variable to store the change in the position can be implemented, and, similarly to the injuries, the changes can take place in the amend (sub)phase.

7. Constants

In this section, some parameters that characterize the game are reported as constants. Their values should be chosen to customize the dynamics or the character of the designed game. Referring to them in the code allows to modify easily the overall dynamics of the game. A proposal for an implementation of these parameters is further described in Sect. 9.3.

7.1. Game constants

The following constants rule the layout of the structural elements of the game (see Fig. 2).

- `SCREEN_WIDTH`: width of the game window
- `SCREEN_HEIGHT`: height of the game window
- `WALL_POS`: horizontal coordinate of the center of the wall
- `WALL_WIDTH`: width of the wall
- `MINE_POS`: horizontal coordinate of the center of the mine
- `MINE_WIDTH`: width of the mine
- `BARRACKS_POS`: horizontal coordinate of the center of the barracks
- `BARRACKS_WIDTH`: width of the barracks
- `TOWER_HEIGHT`: height of the tower
- `GROUND_HEIGHT`: height of the ground
- `MINE_HEIGHT`: height of the mine
- `BARRACKS_HEIGHT`: height of the barracks

- `WALL_HEIGHT`: height of the wall

Besides, some constants can be defined to automatically check the coherence of the values chosen for the above constants:

- `BORDER`: minimum distance of the mine building from the border. The condition
  `BORDER <= MINE_POS-MINE_WIDTH/2`
  assures that the mine is in the displayed area.

- `BUILDING_SEP`: minimum distance between the buildings. The condition
  `BUILDING_SEP <= BARRACKS_POS-BARRACKS_WIDTH/2 - MINE_POS-MINE_WIDTH/2 and`
  `BUILDING_SEP <= WALL_POS-WALL_WIDTH/2 - BARRACKS_POS-BARRACKS_WIDTH/2`
  assures that the buildings are not overlapping.

Another condition that need to be verified is the following
`SCREEN_WIDTH-2*WALL_POS > 2*TOWER_RANGE`
to avoid that the wall is in adversary tower range.

7.2. Castle constants

- `INIT_RESOURCES`: initial amount of resources

7.3. Units constants

Worker:

- `WORKER_COST`: amount of resources to generate a new worker
- `WORKER_TRAIN`: number of turns required to generate a new worker
- `WORKER_PROD`: amount of resources that a worker generates each turn it spends in mine
- `WORKER_SPEED`: distance covered by a worker in one turn
- `WORKER_REPAIR`: amount of wall HP restored in one turn

Swordsman:

- `SWORD_COST`: amount of resources to generate a new swordsman
- `SWORD_TRAIN`: number of turns required to generate a new swordsman
- `SWORD_SPEED`: distance covered by a swordsman in one turn
- `SWORD_RANGE`: maximum distance at which the swordsman can hit an adversary unit
- `SWORD_HIT`: damage in HP delivered to the target
- `SWORD_REST`: turns of inactivity after delivering a damage
- `SWORD_HEALTH`: amount of HP initially given to the swordsman

Archer:

- ARCHER_COST: amount of resources to generate a new archer
- ARCHER_TRAIN: number of turns required to generate a new archer
- ARCHER_SPEED: distance covered by an archer in one turn
- ARCHER_RANGE: maximum distance at which the archer can target an adversary unit
- ARCHER_HIT: damage in HP delivered to the target by an arrow shoot by an archer
- ARCHER_REST: turns of inactivity after shooting an arrow
- ARCHER_HEALTH: amount of HP initially given to the archer

Arrow:

- ARROW_SPEED: distance covered by an arrow in one turn

7.4. Buildings constants

Wall:

- WALL_HEALTH: maximum of HP of the wall

Tower:

- TOWER_RANGE: maximum distance at which the tower can target an adversary unit
- TOWER_HIT: damage in HP delivered to the target by an arrow shoot by the tower
- TOWER_REST: turns of inactivity after shooting an arrow

8. Formats

8.1. Save/Load

The game status consists in the value of the properties of all the element of the game.
The following textual format is proposed. The file is structured in three macroblocks, corresponding to the properties of the game, the two players. The blocks are limited by START and END keywords, followed by Game, Player1, and Player2 keywords. Inside the blocks, there is the list of the properties (name and value) of the elements, one per line. For the game, the properties can be:

- TURN: #turns

For players:

- RESOURCES: #resources
- WALL: current wall health
- WORKER: worker position (BARRACKS, WALL, MINE)
- SWORDSMAN:

  - POS: swordsman's position (coordinate in the game world)
  - HEALTH: swordsman's health

- ARCHER:
  - POS: archer's position (coordinate in the game world)
  - HEALTH: archer's health

- ARCHER_ARROW: arrow position (coordinate in the game world)

- TOWER_ARROW: arrow position (coordinate in the game world)

Notice that the position of all the elements (units and archer arrows) are monodimensional (only its horizontal coordinate is important), while for tower arrows position two coordinates are needed.
Example:

```
START Game
TURN 4322
END Game

START Player1
WALL 1000
END Player1

START Player2
WALL 892
ARCHER POS 126 HEALTH 20
TOWER_ARROW 800 32
END Player2
```

9. Resources

9.1. The pygame library

pygame is a free and open-source cross-platform library for the development of multimedia applications like video games using Python [2].

Getting started with pygame is fairly easy, since several tutorials and introductory examples are available at the docs page [3]. There is also full reference documentation for the entire library.

The tutorials introduce the usage of the library gradually, covering all the fundamental aspects of the realization of a video game.

9.2. Sprites

The graphics is one of the main features of a video game. Beside the technical implementation of the animation of the game elements, the artistic aspect of the design and realization of the sprites can require considerable effort both for time and design. Since the artistic skills are not a evaluation factor for the project, simple solutions are acceptable. For instance, geometric shapes of proper appearance can be used to represent the sprites and the structural elements.

Another solution is to find the needed images on a repository (e.g., on opengameart.org [4]). On this website, the "Hack and Slash 2D Avatars" collection [5] can be of inspiration. A further processed version containing all the sprites needed for a minimal version of the game can be found on the course website (in the "Useful resources" folder [6]). This collection contains also the buildings, which have been used in Fig. 2 to illustrate a possible implementation of the parameters defined in Sect. 7.

The sprites sequence associated to every element of the game depends by it status, with a possible exception: the arrow shot by the tower. In fact, it is the only element which can move along a non-horizontal direction. The arrow direction will depend on the target position and can be influenced by the geometrical parameters of the implementation. In the above mentioned collection of sprites, three arrows sets are provided (horizontal, vertical, and diagonal), allowing to choose the most suitable accordingly to the game circumstances.

9.3. Constants

The value of the constants described in Sect. 7 should be chosen accordingly to several factors. For instance, the size of the sprites can suggest the value of the geometric parameters (the contrary is also valid: the sprites can be created to fit some geometrical constraints). The game dynamics modality (real-time vs. turn-based) can condition the time dependent parameters (e.g., training time, speed of the units).

Inspired by the size of the sprites in [6], a possible implementation is proposed in [7] and the corresponding geometry of the game is illustrated in Fig. 2.

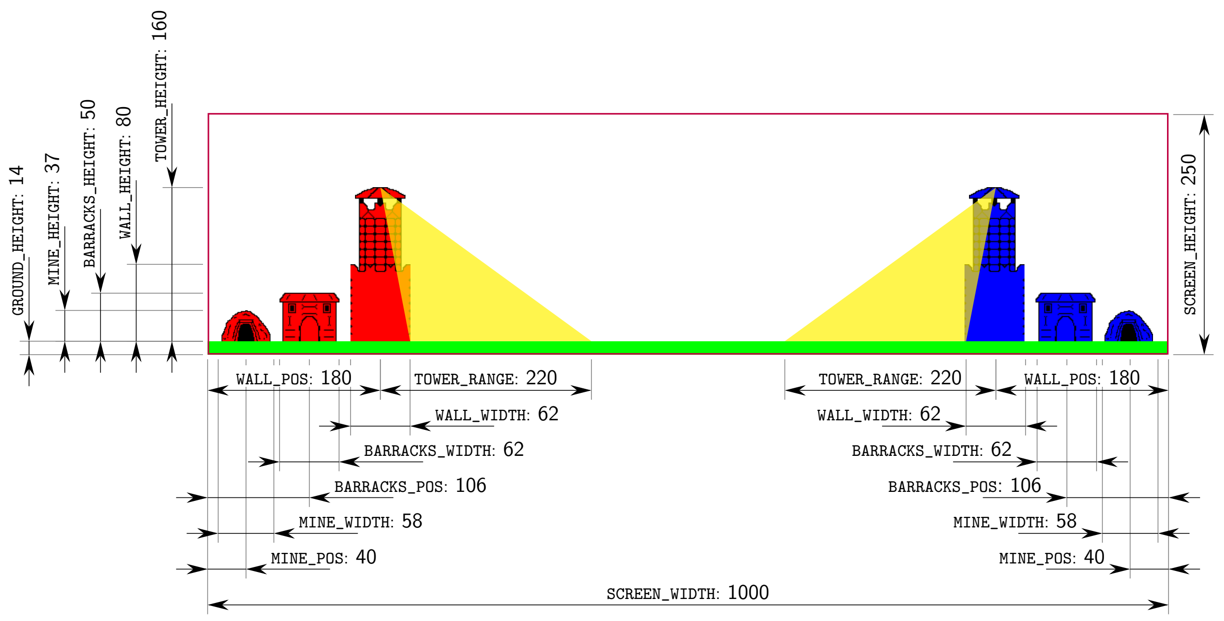

Figure 2: Reference schema for the game playground. The dimension are referred as the geometrical constants (see Sect. 7) and customized for the sprites in [6]. Values of the parameters are implemented in [7].

References

[1] "Pygame website." `https://www.pygame.org/`.

[2] "pygame · PyPi." `https://pypi.org/project/pygame/`.

[3] "Pygame documentation." `https://www.pygame.org/docs`.

[4] "Opengameart.org." `https://opengameart.org/`.

[5] "Hack and Slash 2D Avatars — OpenGameArt.org." `https://opengameart.org/content/hack-and-slash-2d-avatars`.

[6] ""Hack and Slash 2D Avatars" modified and extended." `https://elearning.unipv.it/pluginfile.php/144890/mod_folder/content/0/sprites.zip`, 2022.

[7] "Constants for castles war." `https://elearning.unipv.it/pluginfile.php/144890/mod_folder/content/0/castles_war_constants.py`, 2022.