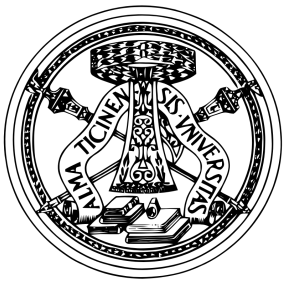
Report of Python cannon game project

Bachelor of Science in Artificial Intelligence

Computer Programming, Algorithms and Data Structures

Submitted by:

Sara Ruggeri (matr. 535570)



I. Goal of the project

My primary goal when developing this project was to build an easy-to-play and interactive game using Python, while also offering an immersive retro experience with the use of pixel art.

I wanted to showcase my personal understanding of programming concepts, especially object-oriented programming, with the use of Kivy in a practical way.

Sugar Wars lets players of all levels experiment with projectile motion physics by adjusting parameters like angle and velocity to hit targets, while impersonating an adventurer trying to save a bunny's jam supply.

The levels get harder and harder as the game progresses, but are still kept to a manageable grade, as to allow players to reach a pretty high score.

As I had never completed any Python projects before, this was a big challenge for me. Slowly, I started to understand more and more of how Kivy worked, and I began to truly enjoy working with it. My original idea grew as I developed, and changed during the course of the project multiple times, making me learn from my mistakes.

The main assumptions are that the player has a basic understanding of how to play simple games on a computer, and that the devices on which the game will be played on have an adequate screen resolution and processing speed.

II. Logical Structure and Design Choices

The overall code can be divided into 6 parts:

I. Global Graphics Configuration and Utility Functions

The first lines fix the frame rate at a maximum of 60 frames per second, and set the window at a specific size, making it impossible to resize it. This makes it easier to build the game, as we don't have to define any functions to adjust objects neither from shrinking nor from enlarging. In the beginning, I actually had implemented such functions, which then got taken out as I didn't need them anymore.

The functions following right after are the true basis of the game's main interactions: **‘collides’** works to detect when two objects overlap and returns a Boolean, while **distance** calculates the space between two points.

II. DataHandler

It’s designed to manage leaderboard data in the game, it reads and writes data onto a .csv file. It aims to **construct a list** of tuples, each containing a name and a score from each line of the file, and it includes error handling both in the case where the file isn’t found or when there is some malformed data: in that case, it just skips it.

It also is capable of **adding new entries** to the leaderboard file.

III. GUI Elements

Bullet, Laser and Cupcake classes are all of the weapons the player will have to use in order to interact in the game itself.

The **bullet** is initialized as a rectangle with a fixed size and an image source. It can be shot by the player and used to hit obstacles and enemies.

The **cupcake**’s initialisation is identical to the one of the bullet, but the size and image source change. It is shot by the player and affects surrounding obstacles and/or enemies up to a certain radius.

For unit weapons, meaning bullets and cupcakes (the equivalent of bomb), besides the usual initialization using self.canvas, there is also a function called ‘**set_pos**’. This function is crucial in order to move the bullet to a new location in response to user input, so it’s easy to imagine how this will become useful in the development of the game.

The **laser** is, instead, initialized as a rectangle with a fixed size and an initial color. It is used by the player and can hit enemies and some obstacles as well, such as the mirror. When the laser hits the mirror, it gets reflected.

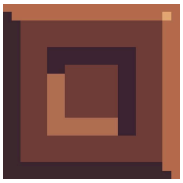
The Laser class includes setting the **color** of the laser to be **randomized**, as well as methods to update the translation of the laser and to set its position and its

rotation. If the current color is within a threshold (0.01) of the target color, a new random target color is generated. This helps to transition gradually between colors.

The **Mirror** and **Vertical Mirror** classes are initialized as white rectangles of a fixed size. The horizontal mirror has a fixed position, and the vertical mirror also has one, but only in the case where no other is being specified.

```
    # start the cooldown, that is used to avoid multiple collisions
    def start_cooldown(self):
        self.cooldown = True
        # after 0.5 seconds, the cooldown is reset
        threading.Timer(0.5, self.reset_cooldown).start()
    def reset_cooldown(self):
        self.cooldown = False
```

For both Mirrors, I implemented a **cooldown**, a method through which multiple collisions are avoided by setting a period of time (0.5 seconds) in which you just cannot hit the same mirror again.



The **Rock** class uses a rectangle with a size-specified image source and non-fixed position as a canvas. The Perpetio class does the exact same thing but with a different image source. While the rock, if hit by the bullet or by the cupcake, gets deleted, the perpetio never does.

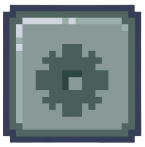
For both Rocks and Perpetios, there is also an '**on_pos**' method, an automatic Kivy built-in mechanism to ensure that the graphical representation of a rock stays in sync with its logical position. In essence, if the position of a rock's instance is set to (x, y), two random coordinates, this method will automatically update the graphical representation's position to (x, y) as well.

The **Powerbar** class is a rectangle of a specified color, size and position, although this will change as a result of key presses, and will result in a variation in the bullet and cupcake's velocities.

The **Wormhole** class' peculiarity is the fact that it is initialized with several parameters: these are used to configure the visual aspects of the wormhole's front and rear parts, and these appear as two Image

widgets within such class. The wormhole allows the player to shoot a bullet or cupcake inside one of its sides and get it teleported to the other side.

Let's now move on to the more 'fixed' elements, that not only appear inside the levels, but also outside of them. Moreover, those handling settings and scoring.



Initialized as an image button and added as a widget in a specified position of the screen, the **options button**, when clicked, opens a popup with 3 other buttons organized with a Box Layout, where each performs a different function:

- **Quit**: quits the game
- **Resume**: closes options
- **Help**: opens a brief tutorial, that explains how to play

The choice of putting the **Music Button** outside of the options popup came from the need to control music, without having to actually open a popup, which can be annoying. Music Button handles music playback, and allows the user to control music's state by just clicking a button.

Score Display and **Score Banner** do the same thing in different ways. The main difference is that the Display is the one being shown in the levels, and needs to be constantly updated, while the Banner appears at the end of levels displaying the final score in a different size and position with respect to the Display. I could've probably found a way to do the same with just one class, but this approach was intuitive and didn't cause any conflict, so I kept it this way.

IV. Game Widgets (one per level)

Each share a common underlying structure, but some have elements that others don't, and vice versa. They are all classes inheriting from RelativeLayout, that is a layout class in Kivy that allows you to position and size widgets relative to the layout's properties or to other widgets within the same layout. This means you can place a widget relative to the layout's right edge or make a widget a certain percentage of the layout's width, for example.

For simplicity, I will describe each element, even though some levels don't have them and others do.

In the initialisation of each of the levels, we set a **global variable final_score**_1 (or 2, 3 with respect to the specific level we are considering) that will store the final score, which will be the starting score for level 2. Initially, but this is just for the first level, we also set the initial score to 10000 points.

We schedule, after 20 seconds, the calling of a function (**start_deducing_points**) that not only will deduce 10 points per second, but will also display a warning informing the player that, from now on, they will lose points as time passes.

Then, we handle keyboard input by first requesting access to keyboard input and assigning it to self._keyboard, then we bind methods to specific keyboard events:

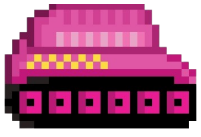
- **_on_key_down**: handles keys such that, when a key is pressed, this will result in the activation of specific weapons in the game.
- **_on_key_up**: when a key, of those that were defined before, is released, it is removed from the set of keys that are being pressed

Then we specify that no collisions are occurring at start, and right after we load the sound that will be used as hit sound when a rock is stricken.

We then initialize an empty list to keep track of all the rocks, this will be needed both for the creation of rock walls and for efficient collision detection on these rocks. We call the functions to **create horizontal and vertical walls**. What they do is they start with a number of rocks in the wall, and this can be changed as much as we want. For vertical walls, x position can be varied, while y position is based on the rock's heights. Each time we add a new rock, we move up by a rock's height, and we add it to the list of rocks.

The creation of the horizontal wall happens similarly, although for the subsequent levels a different function will be used, that allows to set **starting_x and finish_x** as arguments of the function, in a way that allows for more flexibility as to where exactly the wall will be set.

After the functions for the horizontal and vertical rock wall are called, the background image is defined as part of self.canvas.before, as it's intended to appear before anything else. The enemy is then created, as well as the player (that allows for cannon's translation and rotation).



The player is made up of two elements: a cannon and a tank.

Finally all of the widgets that make up the level itself are added directly using the add_widget method.

Finally, we initialize the ‘active’ variables for bullet, laser and cupcake as False.

There are, in fact, functions describing bullet, laser and cupcake activation. They are called each time the player presses the key corresponding to them (space, l and k respectively).

For the **activate_bullet** method it can be said that, if the bullet isn’t already, it must be activated. After that, the score is immediately updated with 100 points less. Bullet mass, that is fixed, and velocity, that instead depends directly on the powerbar’s size, as well as the time it has been shot at, are set. The cannon’s tip position is needed as well, in order to calculate its parabolic trajectory, that is defined using the set_pos function that was discussed earlier: the trigonometric equations ensure that the bullet is fired exactly from the cannon’s tip.

Finally, the bullet’s velocity components are described as well, using bullet velocity and cannon angle. Activate cupcake is defined in the same way, but this time using a bigger mass and score decrease.

The laser does the same thing in a similar way, but there are a few differences that must be considered: first, there is no way to control its velocity, which is fixed, and second, the laser uses two separate methods (set_pos_laser and set_trans_laser) to position and rotate the laser.

The function **hide_enemy** is called when the enemy has been hit by a bullet, laser or cupcake. Once this happens, the music automatically stops, the enemy position is moved off screen, the final_score_1 variable is updated, and we schedule the transition to an intermediate screen.

Handling mirror collision is a delicate operation that needs to be partially dealt with outside of the update function. The method **reflect_laser** calculates laser reflection based on the mirror’s orientation (vertical or horizontal) and even ensures that the laser isn’t immediately colliding with the same mirror over and over again by introducing an offset distance.

To work with wormholes, there is the need for a function, **teleport_bullet**, that is able to determine which part the bullet will exit from based on the one it came in

touch with first, the point in which it entered the wormhole, the part it will exit from and how.

The **update method** is designed to handle parts of the game that must be checked and executed periodically.

This is where collisions are detected and handled accordingly. There are many types of collision that can verify during the game:

I. **Mirror-laser** collision: if the laser is colliding with the mirror and the mirror isn't on cooldown, the laser is reflected (but it must be specified whether the mirror is vertical or not) and and the collision process is restarted
II. **Bullet-rock** collision: as there is a list of rocks that can get hit, if the bullet hits a specific rock, it's only that rock that will have to be removed. When there is a collision, a hit sound plays.
III. **Bullet/cupcake-wormhole** collision: if the collision is being verified, the function teleport bullet is called and executed, based on the part of the wormhole with which they are colliding.
IV. **Bullet/laser/cupcake-perpetio** collision: each time this happens, the corresponding weapon gets deactivated and moved off-screen.
V. **Cupcake-rock(s)** collision: as the cupcake has a radius up to which all of the objects inside of it will be affected, we can remove a number of rocks within that radius even if the cupcake widget is actually hitting just one of them. There is a for loop to find rocks inside of that radius, and then we add them to a list of 'affected_rocks', that are then removed.
VI. **Laser/bullet/cupcake-enemy** collision: the function hide_enemy described earlier is called
VII. **Tank-enemy** collision, **tank-wall** collision: checking for these types of collisions allows to limit player movement up to a maximum and minimum x value, based on the player's size and position.

In the update function, the player's and cannon's movements, as well as the powerbar's, are defined by key presses that allow them to change within a certain range.

Finally, the bullet, laser and cupcake trajectories are calculated. They employ the use of dynamics and real physics. The y component of the bullet (and cupcake) is defined by $y = x_0 + v_0 t - \frac{1}{2} g t^2$, x as $x = x_0 + v_0 t$.

If the bullet surpasses a certain x or y axis, it gets deactivated and moved where it can't be seen. The same for the laser and cupcake.

The laser's x and y components, as it follows a linear trajectory, follow a different dynamics than the bullet and cupcake, and their equations reflect the ones of a uniform rectilinear motion.

V. Screens

Each page is managed by a specific screen, in a way that makes it easier to organize them in the actual App class.

The **Homepage** is pretty basic, with a background displaying the game's name and a start button. Pressing it will lead to the story screen.

The **Story Screen** displays two images, one after the other, and this succession can be controlled by pressing a 'next' button.

Once the second story image is displayed, pressing the next button will lead to the levels page.

In the **Levels Page** screen, we are presented with 3 distinct buttons each indicating a different level. Pressing '1' will lead to level 1, while pressing buttons 2 and 3 without having concluded the previous levels will display a popup that informs the player that the level they requested is currently locked.

Each level is represented by a different screen (**Level1, Level2 and Level3**) with a game_widget widget and an options button.

When Level 1 is completed, the game moves onto Intermediate Screen 1, and the same happens for Level 2 and Intermediate Screen 2. These pages' purpose is to display the level's final score, a 'next' button to automatically move on to the following level and a 'back to levels' button, that makes the user move back to Levels Page.

When completing Level 3, instead of an intermediate screen, the player is presented with the leaderboard screen.

The **Leaderboard** first displays a popup with a text input, where the player can insert their name. Once this has been cleared, the player confirms their name and can actually view the leaderboard. The latter is composed of the leaderboard that was defined inside the Data Handler. The player's name and final score are thus inserted in the leaderboard, which is updated and displayed through a Scroll View and a Box Layout. There is also a 'home' button that, if clicked, brings the player back to the home screen.

VI. App

I didn't divide the code because I had gotten to a point where it was better to just keep it as a unique file, instead of not doing so. Of course this was at the cost of a bigger effort in maintenance, debugging and troubleshooting. Overall, if I could go back, I would start dividing it sooner, as that would've been a wiser decision: more stress in the code's organization would've saved me some time in the end.

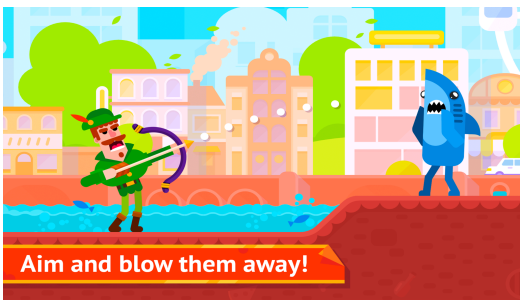
Another important choice was also to not use a .kv file. This came as a necessity, as I found it much easier to handle a single file rather than two different ones.

III. Tools, Resources and References

- **Kivy (v2.3.0)**
- **Python (v3.12.4)**
- **VS Code (v1.91)**: quick and intuitive IDE
- **Chat GPT 3.5**: I used AI especially when struggling with understanding Kivy use and its concepts, while I also sometimes asked if some methods could be improved efficiency-wise.
- **Resprite (v1.35)**: drawing tool made specifically for pixel art. I drew every element in the game (backgrounds, icons, characters and so on) solely relying on this App.
- **Suno AI (v3):** used to create sound effects and the game's music

Inspiration

- Roman Stanek Cannon Game



- Samuele Pittore: we initially started the project together and some parts of the code from our initial collaboration were kept.

I was also inspired by other games with the same aim and gameplay, more specifically Bowmasters and Tank Stars.

The ‘friendliness’ of Sugar Wars’ tank and of its enemies, which were also designed to resemble characters from the acclaimed tv show for kids ‘Winnie The Pooh’, was inspired by Bowmasters’ character design.

The overall structure of levels, the main menu and the settings were instead inspired by old Super Mario and Pokemon games.

IV. Testing and results

I conducted testing on a Windows 10 laptop with an Intel i7 processor, 16GB RAM, using Python 3.12.4 and Kivy 2.3.0. Test cases included basic functionality checks, user interface assessments, performance benchmarks, and device compatibility tests on various screen resolutions and operating systems (MacOS for instance).

Manual testing was performed with predefined scenarios including firing the cannon, hitting enemies, mirror collision, and scoring. When a bug was encountered, before modifying the code, I inserted print statements before and after the action that was supposed to occur but wasn't. This approach was effective in verifying whether the code executed that particular section or if it was skipped, ignored, or incorrect.

Bug fixes

1. **Mirror Reflection Bug:**
    - **Issue**: The laser reflected by the mirror would continuously re-hit the same spot, causing it to get stuck before moving off.
    - **Fix**: Introduced an offset and a cooldown period during which the laser cannot hit the same mirror again.
2. **Rock Wall Collision Bug:**
    - **Issue**: When a bullet hit the rock wall, the lowest rock in the wall would be deleted instead of the specific rock that was hit.

- **Fix**: Implemented a method in the update class to iterate through the list of rocks using a for loop, ensuring only the rock that was hit was removed.

3. **Music Overlap Bug:**
    - **Issue**: Music would continue playing at the end of a level and overlap when moving to the next level.
    - **Fix**: Initialized the music for each level as false. When the player hits an enemy, the music automatically stops.

V. Possible Improvements

- A way to define levels more efficiently, without the need to copy and paste the similar part for three times
- New worlds (I thought of something Willy Wonka inspired)
- Ways to improve the tank: maybe a double cannon or an ‘automatic shooter’
- Power-ups: if you get a really high score, you can get power-ups (such as one that doesn’t make you lose points for a specific amount of time, or that makes you immune to enemies that hit you back)
- Boosts: specific improvements that appear randomly during levels, and you can get them by aiming at them and hitting them. If you do hit them, you gain back the points you just lost and you also gain some more, but if you don’t, you just lose the points and don’t get anything.
- Harder levels with multiple enemies hitting you back
- Companions: helpers that automatically shoot directed at the enemies and that can defend you from enemy attacks up to a certain limit (once their ‘score’ gets to zero, they die)
- A “toxic rain”, basically obstacles free-falling from above. If the player gets hit, he loses points.