Report of Python cannon game project

RedHood's Berry Blast

Bachelor of Science in Artificial Intelligence
Computer Programming, Algorithms and Data Structures, Mod. 1

Submitted by:
Eunice Abike Omolayo Afuye (matr. 536113)
Alexandra Serena (matr. 535282)

I. GOAL OF THE PROJECT

The goal of our project was to design and build a simple yet interactive cannon game using Python and the Kivy framework, presenting it with a typical pixel-art style that is both elementary and enjoyable.

The story involves Little Redhood on a quest to save her grandma from the Wolf, which is allied to the Evil Crow and the Rattlesnake. At each level, Little Redhood can use the cannon to shoot Berries (Bullets), Cupcakes (Bombshells) or a Chocolate Laser and destroy all the obstacles, in order to finally shoot the adversary and pass to the next one. The last enemy is the Wolf itself, guarding the house in which Grandma is being kept hostage. Defeating the Wolf means winning the entire game.

A central role is played by physics' principles that involve parabolic motion of the projectiles (determined by gravity), the bouncing motion of the bullets and the reflection of the laser.

We wanted to allow the user to interact as much as possible with the environment of the game, therefore one has the option to adjust the angle of the cannon, the velocity and the mass of the projectiles.
The levels reveal different challenges and game elements, aiming to increase the difficulty at each level while still maintaining great chances of destroying the obstacles and winning the game.

The game has been a great challenge to us as we didn't have any prior experience with Python and programming before starting this project, and the idea of developing an actual game definitely felt complicated. Once we started organizing the work, got familiar with Python's basics and discovered Kivy's properties and tools, everything started falling into place and the game began taking the shape we imagined. Through many issues and doubts, we were able to design a game of which we are proud of and, most importantly, has been an important learning process.

II. LOGICAL STRUCTURE AND DESIGN CHOICES

II.I GENERAL ORGANIZATION OF THE CODE

The project is organized into a clear folder structure to separate concerns and make the codebase maintainable and scalable. Each folder and file has a specific purpose, reflecting the main components of the game:

- ***main.py***

This is the entry point of the application. It initializes the Kivy configuration, sets up the main App class, and manages the transition between screens.

- ***constants/***
  Contains files like `screen_constants.py` and `physics_constants.py` that define global constants (e.g., screen size, FPS, physics parameters). Centralizing constants makes it easy to adjust game settings without searching through the codebase.

- ***functions/***
  Houses utility modules such as `save_load.py` for saving/loading scores and `timer_widget.py` for time management. This separation keeps utility logic distinct from game logic and UI code.

- ***screens/***
  contains all the Kivy screens (e.g., `start_screen.py`, `story_screen.py`, `game_screen.py`) and the screen manager. Each screen is responsible for a specific part of the user experience, such as the main menu, story introduction, or gameplay.

- ***levels/***
  Includes files related to level-specific logic and widgets, such as `cannon.py` and its corresponding `.kv` file. This keeps level mechanics and visual representation modular and easy to update.

- ***projectiles/***
  contains all projectile-related code, such as `laser.py` and other projectile types. Each projectile has its own file to encapsulate its behavior, movement, and collision logic.

- ***obstacles/***
  Includes classes for obstacles like `mirror.py` and `perpetio.py`. This separation allows for easy addition or modification of obstacle types without affecting other parts of the code.

- ***resources/***
  Stores images keeping the codebase clean and organized.

-***ui/***
The ui folder contains user interface components that are designed to be reusable and modular across the game.

This modular structure makes the project easier to navigate, debug, and extend. By grouping related functionality into folders, each part of the game (UI, logic, resources) can be developed and tested independently, supporting future scalability and collaboration.

II.II. **GLOBAL GRAPHICS CONFIGURATION AND UTILITY FUNCTIONS**

The game window is configured at startup using Kivy's Config object, setting a fixed resolution (SCREEN_WIDTH, SCREEN_HEIGHT) and frame rate (FPS).
Utility functions are organized in the functions folder, including, score saving/loading and timer management.
Constants for screen and physics setting are centralized in the constants folder for easy tuning and maintenance.

FUNCTIONS/

The files **save_load.py**, **timer_widget.py**, e **hall_of_fame.py** are used to manage the player's progress, measure time during levels, and save and display high scores in the Kivy game.

- **save_load.py** → Allows saving and retrieving the state of played levels and scores. It's used to:
    - Remember the time taken for each level.
    - Delete or reset saved data.
    - Manage the **Hall of Fame** (save scores in JSON format).

  The function saved_score (player_name, level_name, target_type, time_taken) builds a record with the player's name, level, target type, time, and timestamp.
    - Through the reset_scores() function, the Hall of Fame is reset.

The purpose of this module is to allow the player to **continue from where they left off** or **view their results** in future sessions.

- **timer_widget.py** → Manages everything related to game time.
  The properties used are:
    1. current_time: tempo rimanente (min 0, max iniziale 60).
    2. running: boolean indicating if the timer is active.
    3. level_name, target_type, on_timer_end_callback: store the level, target type, and a callback function to invoke at the end.
    4. time_elapsed: Returns the time elapsed since the timer started (60 - current_time).

    The structure is the following:
    - start() → Sets current_time = 60, running = True, and starts Clock.schedule_interval(self.update_time, 1) to update every second.
    - stop() → Sets running=False and removes the scheduled update_time method.
    - update_time(dt) → Invoked every second if running=True.
      Decreases current_time; if it reaches 0, it stops and calls on_timer_end_callback if defined.

level_completed()

1. Stops the timer.
2. Calculates time_taken = 60 - current_time.
3. Gets player_name from the running App (defaults to 'Unknown').
4. If level_name, target_type and player_name are valid, it calls saved_score(...) to store the score.
5. Calls SaveLoadManager().save_level_state(...) to save the level's state (time and target type).

With this timed functionality, the game becomes a **time-based challenge** and tracks player performance.

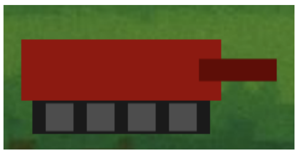
- **hall_of_fame.py** → Popup responsible for displaying and saving scores.
  - Gets the player_name from the App.
  - Converts time_taken to a string if necessary.
  - Builds a record (with name, level, target type, time, timestamp, and shots).
  - Checks for the data folder and creates it, if missing.
  - Loads existing scores from the JSON file (or creates an empty list).
  - Appends the new record and rewrites the JSON with indentation.

populate_scores()

- Retrieves scores_box from the interface and clears it.
- Loads JSON from data/scores.json or sets an empty list if the file doesn't exist.
- Filters records with the current player_name.
- If the list is empty, it displays "No scores yet!".
- Otherwise, sorts records by level and time, then adds a Label for each score with the format:
  "Player: ... — Level: ... — Target: ... — Time: ...s"

The purpose of the hall of fame is to motivate the player to improve and compare their results.

II.III. **GAME ELEMENTS** (Weapons: Redberry/Blueberry, Muffins, Laser
Obstacles: Rocks, Perpetio, Bulletproof Mirror, Elastonio
Options Button, Help Button, Start/Back Button)



Starting from the **tank**, we created it using Kivy's properties, it is in fact a widget designed as a main red rectangle supported by another thinner black and grey rectangle that is the base, and finally there is the darker red **cannon** emerging.

Then, to discuss **weapons**, we will list them here:

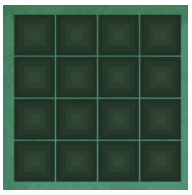
- **BERRIES as BULLETS**: we inserted the bullets in the shape of berries for which we exploited an AI generated image, we created a class where the bullet is a widget (subclassed as Image) and we assigned the velocity on both component x and y, where we applied downward acceleration for the gravity to the velocity on y.

- **CUPCAKES as BOMBSHELLS:** we inserted the bombshells in the shape of cupcakes for which we exploited an AI generated image, then identically to the bullets: we created a class where the bombshell is a widget (subclassed as Image) and defined its velocity both on x and y , where we applied downward acceleration for the gravity to the velocity on y. Differently from bullets, we gave the user the possibility to change the mass of the bombshell while playing, hence influencing the damage radius.

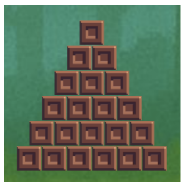
- **CHOCOLATE LASER as LASER:** we created the laser by drawing with Kivy a thin rectangle, we set the constants that determined the laser's constant velocity, the distance it travels into the obstacles and how long it can go. With canvas we were able to give the laser an external glow inside the function **draw_laser**, where we defined the coordinates of the laser. We also defined a function that allows the laser to perceive when it intersects with other lines and therefore with objects of the game (obstacles).

Moving forward, the obstacles are:

- **ROCKS:** we created a widget class for the single rocks -represented using an AI generated image- and then we were able to build entire rock groups with another class where we built a square made of rocks. In the rock groups class we used Relative Layout, a Kivy's tool that let us change the positions and move the blocks wherever we wanted.



- **PERPETIO:** we did the exact same thing as with the rocks. The only difference is that, instead of building squares made of separate perpetio widgets, we built them in pyramids, and then we moved them around again with Relative Layout.



- **ELASTONIO:** we created a widget class and we drew a rectangle with a specific thickness, we colored it yellow using Canva. When a projectile (bombshell) hit the elastonio, it would bounce back thanks to a function we called **bounce_back** -positioned in the bombshell.py file within the function that handles collisions- that inverted the velocity of the projectile the moment it hit the obstacle and could come back.

- **BULLETPROOF MIRROR:** we created a widget class and handled it in the same exact way as the elastonio, except we colored it blue with Canva and we managed the reflection of the laser differently from the bouncing on the elastonio.. As a matter of fact, we didn't simply invert the velocity, but we used the angle of incidence in order to compute the angle of reflection, applying therefore an important physics' property.

About the **enemies**, in each level's file that we used for the Kivy's graphical settings we exploited BoxLayout, a Kivy's property that let us display on the screen all the elements by moving them with specific coordinates in x and y directions. We inserted the enemies by adding them in the BoxLayout of each level as Images which we represented with AI generated pictures.
To all of them, we also added an animation which made the enemies move in a hovering way, and we used Animation, a Kivy's class.

The **HELP BUTTON** was added as a button to each level screen and in the general levels screen as well. On each screen there is a different description of the Help instruction, depending on what level we are. And in the general levels screen, the help button provides general rules of each level. In the levels file we defined a function **on_help** that allowed us to open every time the button and see different instructions.

II.IIII. SCREENS

The ***screens/*** folder is responsible for managing the game's main user interface. Within this folder, two main components are defined and organized:

1. **Main Game Screens**
   These are based on Kivy's 'Screen' class and represent the navigable flow of the game (such as menu, gameplay, story, etc.).
2. **Modal Popups**
   These are based on the 'ModalView' class and appear above the current screen to manage temporary interactions, such as timeouts or victory notifications.

The screens are structured according to the Kivy MVC (Model-View-Controller) pattern:

- The '***.kv***' files define the graphical interface.
- The '***.py***' files contain the behavioral logic.

Screen Classes

Each screen is implemented as a class that extends Kivy's 'Screen'. These screens form the main pages of the application and are loaded, managed, and switched using the 'screen_manager.py'.

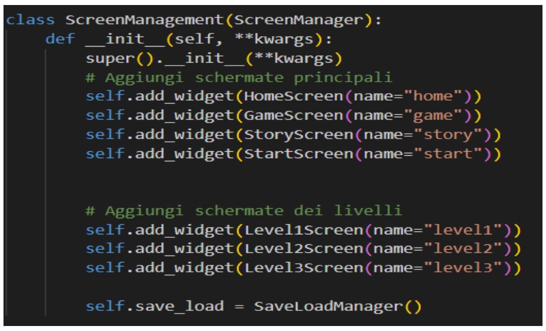
The key screens and popups include:
- **HomeScreen**: The main menu of the game, which allows the user to start the game or access other sections.
- **StartScreen**: The screen where the player enters their name.
- **StoryScreen**: Presents the game's narrative introduction.
- **GameScreen**: Acts as a dynamic container for the actual game levels (such as `level1`, `level2`, and `level3`), which are loaded within it.
- **Congratulations**: A popup that appears when the player successfully completes a level, displaying stats such as time taken, bullets used, and providing access to the Hall of Fame.
- **TimeUpPopup**: A popup that appears when the player runs out of time during a level, offering options to retry or return to the menu.

<u>The role of 'screen_manager.py'</u>

This file is central to the management of the application's navigation. Its main responsibilities are:

- Import all the screen classes (such as 'HomeScreen', 'StartScreen', etc.).
- Create a subclass of 'ScreenManager' to coordinate the transitions between different screens.

- ‘ScreenManagement’ extends ‘ScreenManager’ to centrally and efficiently manage multiple screens ‘scenes’ within the app. It adds each screen to the manager at startup and enables seamless transitions between them using unique names. It may also integrate additional features, such as handling save/load functionality for game data.

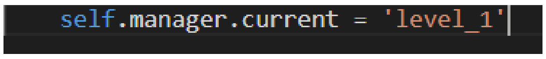


```python
class ScreenManagement(ScreenManager):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        # Aggiungi schermate principali
        self.add_widget(HomeScreen(name="home"))
        self.add_widget(GameScreen(name="game"))
        self.add_widget(StoryScreen(name="story"))
        self.add_widget(StartScreen(name="start"))


        # Aggiungi schermate dei livelli
        self.add_widget(Level1Screen(name="level1"))
        self.add_widget(Level2Screen(name="level2"))
        self.add_widget(Level3Screen(name="level3"))

        self.save_load = SaveLoadManager()
```

- Add each ‘Screen’ to the manager using ‘self.add_widget(...)’.
- Define the logic for changing the current screen by setting ‘self.current’ to the desired screen’s name.



```python
self.manager.current = 'level_1'
```

This structure ensures a clear separation between permanent screens (handled by ‘Screen’) and temporary components (handled by ‘ModalView’), which greatly improves the clarity, reusability, and scalability of the project’s codebase.

III. TOOLS, RESOURCES AND REFERENCES

- **Python(v. 3.12.0)**
- **Kivy(v. 2.1.0)**
- **VS Code**
- **Chat GPT 3.5:** we used AI in the cases where we wanted to better understand Kivy’s properties and tools and how to properly use them. We mostly used it to generate the images of some elements of our game, like the bullets (berries) and bombshells (cupcakes), the backgrounds and the enemies (Evil Crow, Rattlesnake, Wolf).
- **Pinterest:** we used it to find the starting images of some of our game elements, which we then fed to the AI in order to create a similar picture.

IV. TESTING AND RESULTS

The game was thoroughly tested on both **Windows** and **macOS** platforms to ensure cross-platform compatibility and consistent behavior across different operating systems.
During testing, all core features of the game performed reliably and as expected:

- **Cannon Rotation**: The cannon responds smoothly to directional key inputs, allowing the player to aim accurately. The rotation is fluid and free from lag or stuttering across both platforms.

- **Shooting Mechanism**: Projectiles are launched upon pressing the spacebar, alternating correctly between different types, and originating from the cannon's current angle and position.

- **Collision Detection**: Interactions between projectiles and game elements (such as obstacles and targets) are accurately detected. Different behaviors such as explosions, bouncing, or disappearing are triggered appropriately based on the type of object hit.

- **Score Saving and Hall of Fame Integration**: Upon level completion, the player's performance (including time and player name) is correctly saved and displayed in the Hall of Fame. This functionality works without issue on both tested platforms.

- **User Input:** The game controls are intuitive and responsive. Keyboard inputs (arrow keys for aiming, spacebar for shooting) are handled efficiently, providing a satisfying gameplay experience with minimal input delay.

Overall, the application demonstrates a stable and consistent user experience, meeting the expectations for usability and functionality on both Windows and macOS environments.

V. POSSIBLE IMPROVEMENTS

- Sound Effects & Music: Add background music and sound effects to enhance immersion, sound effects and visual feedback when hitting targets, winning, or failing.
- Menù settings: Allow players to adjust sound, graphics quality, and control sensitivity.
- Rewards: when winning fast there could be a reward in matter of how many bullets you have at your disposal, or different projectile type that is more powerful and could be used for an amount of time at each level.
- Add Easy, Medium, and Hard modes with different time limits, obstacle speeds, or enemy behavior.

Technically, the game can be improved by turning repeated code into shared functions or helper classes, keeping all score and game data in one place, and using common base classes to avoid rewriting the same code making the project easier to update, expand, and keep organized.