Summary

1. Introduction

The Cannon Game is a single-player artillery game where the objective is to accurately strike moving targets within a dynamic landscape composed of three progressively challenging levels. The player controls a cannon that can fire three distinct types of projectiles—each with different physical properties and behaviors. Success depends on determining the optimal firing strategy by adjusting both the cannon's angle and the projectile velocity, making the game a rigorous test of both precision and strategic decision-making. Various obstacles are strategically placed throughout the levels to further increase the challenge, and a final score is recorded at the end of the game; if the player is eliminated before completing all levels, their score is saved in the Hall of Fame.

The game is built using an iterative, event-driven approach within the Kivy framework. Kivy's Clock schedules regular updates that continuously recalculate the physics of projectile motion, including gravity, inertia, and other forces, while also processing collisions between projectiles and obstacles. The program's design follows object-oriented principles, ensuring that each game component is encapsulated within its dedicated class. This approach enhances the clarity, maintainability, and extensibility of the code, as illustrated in the accompanying flowchart.

1.1 Libraries and Frameworks

- **Kivy**: Used for creating a dynamic, responsive user interface and managing event-driven programming aspects.
- **Python** (version 3.12): The primary programming language, chosen for its simplicity and powerful standard libraries, such as math.

2. Design Choices

The design choices for the Cannon Game are deeply rooted in the desire to create an engaging and interactive experience that challenges the player's strategic thinking and precision. As mentioned in the introduction, the game is structured into three levels with increasing difficulty. Each level introduces new obstacles, thereby progressively intensifying the challenge. The decision to incorporate three distinct projectile types—each with unique behaviors such as the bombshell's penetration effect, the bullet's gravity-influenced trajectory, and the laser's constant-speed motion—further enriches the gameplay.

User interactivity is enhanced through intuitive controls and a variety of game aids. The player can adjust the cannon's firing angle by using the Left Arrow and Right Arrow keys, while the Up Arrow and Down Arrow keys allow for fine-tuning of the projectile's velocity. This control scheme, managed within the main module's key event handler, reflects a commitment to providing precise and responsive gameplay mechanics. The objective is clear: hit all targets on the screen using a maximum of 10 projectiles. The challenge is compounded as the number of targets increases with each level, and a successful hit increments the player's score by 10 points. If the player runs out of projectiles, the game concludes, and the final score is automatically saved to the Hall of Fame.

In addition to core gameplay, several auxiliary buttons enrich the user experience. On the game screen, four buttons are provided to assist and customize the gameplay.

- **Help button**: which displays the game instructions, ensuring that players can quickly familiarize themselves with the controls and objectives.
- **Select Projectile button**: which enables the selection of different projectile types, allowing players to experiment with various strategies.

- **Reset button**: which resets the level by reinitializing the positions of obstacles and targets and restoring the projectile count to 10, albeit at a cost of 15 points deducted from the player's score.
- **Show Trajcetory button**: which visually projects the path a projectile will follow based on the current angle and velocity settings. This trajectory remains visible for 15 seconds with an accompanying countdown, and its use incurs a penalty of 10 points, introducing a strategic cost to accessing helpful visual cues.

These features are implemented through carefully designed methods in the main module, where each button press triggers a specific sequence of actions that modify the game state accordingly.

When all targets are hit, a congratulatory popup is displayed, leading the player to the next level. If the final level is completed, a final congratulations screen is shown, and the player's entry in the Hall of Fame is marked with "WINNER" next to their name and score.

In the event of a game over, the design offers three clear options:

- **Hall of fame button**: that access the Hall of Fame to review the saved score.
- **Restart button**: that restart the game with the previously chosen nickname.
- **Shutdown**: that exit the game.

The logical structure and implementation of these design choices are reflected in the modular organization of the code.

3. Code Structure and Module Organization

The project is organized into several well-defined modules, each encapsulating specific aspects of the game's functionality:

- **Main Module (Controller):** This is the central module that governs the entire game. It manages the overall game state, user interactions, and the main update loop. The main module coordinates game initialization, level progression, collision handling, score updates, and UI transitions (such as the main menu, pause screen, and game over popups). It achieves this by calling methods from other modules, ensuring a clear separation of concerns.

- **Cannon Module:** It is responsible for rendering the cannon, controlling its rotation, and calculating the projectile launch position (the tip of the barrel). The class uses rotation transformations to simulate angle adjustments and ensures that the cannon's graphical representation remains in sync with its logical state.

- **Projectile Module:** This module manages the behavior and physics of the three distinct projectile types. The module handles launching projectiles, updating their positions based on physics calculations (including gravity and inertia), and implementing projectile-specific properties—such as the bombshell's penetration effect and the laser's constant speed and limited active duration.

- **Obstacle Module**: It manages various in-game obstacles by defining properties, movement, collision detection, and interactions with projectiles. It covers different obstacle types (rocks, perpetios, mirrors, wormholes) and includes specialized behaviors (like teleportation or reflection).

- **Target Module:** This represents scoring targets, which the player must hit to earn points.

- **Assets and External Resources:** All visual resources (backgrounds, buttons, cannon, projectile, and obstacle images) are stored in the image's directory. The game also utilizes an external text file ($hall\_of\_fame.txt$) to maintain and continuously update the Hall of Fame with players' scores.

Hence, the main module serves as the central hub that orchestrates the game's flow by invoking the specialized functions and classes defined in the Cannon, Projectile, Obstacle, and Target modules.

3.1 Main Module

The Main module serves as the central control unit for the game, orchestrating initialization, state management, and user interaction. Upon instantiation of the main game class, all essential variables are initialized. For instance, the player's nickname is set as an empty string, the score is set to zero, and the number of available projectiles is initialized to 10. Additionally, empty lists are created for storing active projectiles and obstacles. Other variables such as the selected projectile type (defaulted to "bullet"), the current level (starting at 1), the cannon's firing angle (set to 45 degrees), and its velocity (set to 50) are also established. The game state variable is initially set to "$enter\_nickname$," and flags for game over and pause are set to false.

Almost immediately, the main module schedules the primary update loop using Kivy's $Clock.schedule\_interval$ to call the update function 120 times per second. This high-frequency loop is critical for ensuring that animations, physics calculations, and state transitions are rendered smoothly. In parallel, the module defines the various background images corresponding to each level, loading a default background image for the initial screen.

When the game launches, the player is first presented with a nickname entry screen. This screen features a $TextInput$ widget, where the player is prompted to "Enter your nickname" and a continue button. Both widgets are carefully positioned using Kivy's $size\_hint$ property, that allows for responsive scaling relative to the parent container, and pos_hint property, which explicit sizes and positional hints ensure that the interface is visually balanced. If the player fails to enter a nickname, a red error label appears, prompting the player to provide the required input. Once a valid nickname is entered and the continue button is pressed, the $start()$ method is invoked. This method not only validates the nickname input but also displays a temporary welcome label before transitioning the game state to the welcome screen.

Following the welcome message, the $proceed\_after\_welcome()$ method is called, which clears the initial layout and updates the background image to that of the home screen. On this home screen, buttons for "Play," "Hall of Fame," and "Help" are displayed.

Each button is designed with a specific background image, explicit size, and position on the screen. Their functionality is bound to specific methods; for example, pressing the Play button leads to the projectile selection screen, while the Hall of Fame and Help buttons invoke methods that display corresponding popups.

The transition to active gameplay is handled by the $init\_game()$ method. This method first updates the background image to one that corresponds to the current level, then either creates or reinitializes the cannon object from the Cannon module. The cannon is positioned, and its angle is reset to its default values. The method then generates the obstacles for the level by calling initialize_$obstacles()$.

To keep the player informed, a score display is created along with a background rectangle (styled in a brown hue) to improve text visibility. Labels showing the current score and the number of shots remaining are positioned on the screen, as well as a label that dynamically displays the cannon's firing angle and the current velocity. Additionally, several buttons are created and added to the interface on the right side of the screen.

- **The help button**, when pressed, calls the $helpscreenshow()$ method. is responsible for providing players with quick access to the game's instructions and guidelines without interrupting the flow of gameplay. In $helpscreenshow()$, an Image widget is created, and it serves as the content for a Popup widget. The Popup is set to $auto\_dismiss$, meaning that it will close automatically when the player taps outside its boundaries, providing a seamless and user-friendly way to access and exit the help screen.

- **The Select Projectile button**, when pressed, calls the pause functionality, encapsulated in the $toggle\_pause()$ method. When the game is paused, the update loop is unscheduled, and a semi-transparent overlay is created to dim the background. This overlay, which is a FloatLayout, displays projectile selection buttons (for bullet, bombshell, and laser) and a resume button. Each button is configured with specific properties and, upon being pressed, calls the method that either selects a new projectile or resumes the game, effectively reactivating the update loop and removing the pause overlay.

- **The Reset button**, when pressed, calls the $reset\_level()$ method. When triggered, it resets the positions of all obstacles (excluding rocks and targets, which are handled separately since during the game they can be destroyed), removes and recreates the rock and target obstacles based on their initial data, and resets the projectile count back to 10. However, this reset comes at a cost: 15 points are subtracted from the score that the player had before starting that level, and a temporary penalty label is displayed to indicate the deduction.

- **The Show Trajectory button,** when pressed, calls the $show\_trajectory()$ method. When invoked, it deducts 10 penalty points from the player's score and updates the display accordingly. A temporary penalty label is shown, and a countdown label with an initial value of 15 seconds is created. The method then calculates the trajectory parameters based on the current tip position of the cannon, its angle, and the velocity (modified by a multiplier that depends on the type of projectile selected). Depending on whether the projectile is a bombshell, laser, or bullet, the method computes the maximum time of flight (t_max) accordingly. A dedicated widget is then either created or cleared to display the trajectory preview. The drawing is updated at regular intervals to account for any changes in the cannon's settings, while a separate countdown method decrements the preview timer. Once the countdown reaches zero, the trajectory preview is automatically removed from the screen.

Each of these methods is tightly integrated into the main update loop, ensuring that state changes and visual updates occur in real time. The main module also handles key events via the $on\_key\_down()$ method, allowing the player to shoot (using the Spacebar) and adjust the cannon's rotation and velocity with the arrow keys. By combining a robust initialization process, responsive event handling, and modular integration of gameplay features, the main module lays the foundation for a dynamic and interactive gaming experience.

3.1.1 Projectiles Selection and Shooting

The projectile selection and shooting process in the Main module is designed to provide a seamless transition from the main menu into the gameplay experience. When the player opts to choose a projectile, the $go\_to\_projectile\_screen()$ method is called. This method changes the game state to "$choose\_projectile$" and clears the existing layout, then sets a new background image that visually indicates the projectile selection phase. At this point, three buttons are created corresponding to the available projectile types: bullet, bombshell, and laser. Each button is bound to a lambda function that calls the $sel\_proj()$ method with the appropriate projectile type as an argument.

The $sel\_proj()$ method is responsible for updating the selected projectile variable with the player's choice. If the game is currently paused during a level, this method will invoke $resume\_game()$ to continue gameplay with the newly selected projectile. Otherwise, if the state is still "$choose\_projectile$," the layout is cleared and $init\_game()$ is called to initialize the level with the chosen projectile. This mechanism ensures that the player's selection is properly integrated into the game flow.

Shooting itself is handled by the $shoot\_projectile()$ method. When the Spacebar is pressed, this method first checks whether any shots remain. If not, it triggers the game over sequence. Otherwise, it retrieves the cannon's tip position (using a method from the Cannon module), which serves as the starting point for the projectile. A new Projectile instance is then created using the currently selected projectile type and the computed starting position. The projectile's $launch()$ method is subsequently called with the cannon's current angle and velocity, which are key parameters in determining the projectile's trajectory. The projectile is added both to a list of active projectiles and to the game widget, and the available shot count is decremented accordingly.

3.1.2 Obstacles Generalization and Collision Handling

Obstacles are initialized and managed through a dedicated set of methods that ensure the game environment evolves as the player progresses. The $initialize\_obstacles()$ method first resets the lists for obstacles and projectiles, then configures the number of each obstacle type based on the current level. For example, in level one, there are a specific number of targets, rocks, perpetios, and so on, while later levels adjust these numbers to increase the challenge.

A helper method, $get\_random\_position()$, generates random positions within a predefined area, ensuring that obstacles are distributed evenly across the screen. Additionally, $get\_valid\_target\_position()$ ensures that targets are placed at a minimum distance from the cannon and are not too close to one another. This method iteratively generates random positions and checks the constraints before assigning a valid location.

For each type of obstacle—whether it be a target, wormhole, mirror, perpetio, or rock—the method creates an instance with appropriate attributes and stores its initial position. This stored data is later used to reset the level if needed, maintaining consistent gameplay and structured obstacle placement.

Collision handling is orchestrated by the $handle\_collisions()$ method, which iterates over each obstacle and checks for collisions with each active projectile using the obstacle's $collision()$ method. The response to a collision varies according to the type of obstacle involved. For instance, if a projectile collides with a wormhole and has not yet been teleported, the code locates a paired wormhole, calculates an appropriate exit position using an offset based on the wormhole's radius, and teleports the projectile to this new location while marking it as having been teleported. In the case of a mirror, the obstacle's $projectile\_reflection()$ method is called, which reflects the projectile's velocity; if the projectile is a bullet or bombshell, it is subsequently removed from play. If the projectile collides with a perpetio, it is immediately destroyed. Additionally, when an obstacle's $on\_hit()$ method returns $True$—such as when a target is hit—the method may update the score, remove the obstacle, or, in the case of a bombshell, remove multiple obstacles within a specific radius due to its penetrating effect. Finally, if there are no remaining targets after processing all collisions, the game triggers a congratulations popup, indicating that the level has been completed successfully.

Throughout this process, the $on\_key\_down()$ method monitors for key presses to ensure that shooting, cannon rotation, and velocity adjustments are performed in real time. The Spacebar is used to fire a projectile, the Left and Right arrow keys rotate the cannon, and the Up and Down arrow keys modify the firing velocity. This

integration of projectile selection, shooting, obstacle management, and collision handling creates a cohesive and interactive gameplay experience that is both challenging and rewarding.

3.1.3 Level Progression and Popups

The level progression in the game is managed by a series of methods that work together to control the flow of gameplay, from advancing to new levels to handling the end-of-game scenarios. The $next\_level()$ method serves as the primary gateway for moving from one level to the next. When called, it increments the current level and resets the number of available shots to ten. If the current level exceeds the maximum level (the third level), the method invokes $final\_screen()$ to display the final congratulatory interface; otherwise, it reinitializes the game state for the new level by calling $init\_game()$. This approach ensures a smooth transition between levels, with each level introducing its own configuration of obstacles and targets to increase the challenge progressively.

The $update()$ method forms the core of the game loop, and it is responsible for refreshing the game state at a fixed interval. It iterates through every obstacle and projectile, calling their respective update methods to animate movement and ensure that all physics calculations remain current. Additionally, $update()$ monitors collisions by invoking $handle\_collisions()$, which processes interactions between projectiles and obstacles. When the number of shots falls to zero and no active projectiles remain, $update()$ schedules a delayed check (via $check\_last\_projectile()$) to confirm whether the game should transition to a game-over state. This real-time updating mechanism is crucial for maintaining an engaging and responsive gaming experience.

When the game reaches its conclusion—either through a loss of shots or by clearing all targets—a series of popups guide the player through the next steps. The $finished()$ method is triggered when the player has no shots left. It halts the main update loop, saves the final score to the Hall of Fame, and displays a Game Over popup. This popup, which is built using a FloatLayout and a BoxLayout for its buttons, presents three options: restart the game (while preserving the player's nickname), view the Hall of Fame, or shut down the game entirely. The clear and immediate feedback provided by this popup ensures that the player understands the outcome and the available choices for how to proceed.

If the player successfully destroys all targets before exhausting their shots, the $congrat\_sc()$ method is called to display a congratulatory popup. This method changes the game state to “congratulations,” clears any remaining projectiles, and then creates a popup that contains a “Next” button. When the player presses this button, the popup dismisses, and $next\_level()$ is invoked to advance the player to the next level. The congratulatory popup not only celebrates the player’s success but also seamlessly integrates level progression by prompting the transition to a new set of challenges.

In addition to these methods, the $final\_screen()$ method handles the scenario when the final level is completed. It writes a winner entry to the Hall of Fame and then presents a full-screen popup with navigation buttons that allow the player to either view the Hall of Fame or shut down the game. This final screen serves as both a celebration of the player’s achievement and a clear end point to the game, marking the culmination of the gameplay experience.

Overall, these methods work in concert to manage the progression through levels, provide timely feedback through popups, and maintain a fluid and interactive gaming experience. The design ensures that players are constantly informed about their progress and are given clear options whether they succeed or fail, all while seamlessly transitioning between different stages of the game.

3.1.4 Hall of Fame

The Hall of Fame functionality is managed by two key functions: one for displaying the saved scores and another for saving new entries.

The $show\_hall\_of\_fame()$ method is designed to present a popup window that lists all the Hall of Fame entries in a sorted, descending order based on score. When this method is called, it attempts to open the "$hall\_of\_fame.txt$" file to read all stored entries. If the file does not exist, a default message ("No Hall of Fame data found.") is used. For each line read from the file, the method processes the string by splitting it to extract the score, which is then used as the key for sorting. The entries are sorted in descending order so that the highest scores appear at the top. Once sorted, the entries are concatenated into a single string with appropriate spacing. A Label widget is then created with this text, set to use white color and a readable font size, and is configured to handle text wrapping (with a defined text size and padding) to ensure clarity. This Label is embedded inside a $ScrollView$ to accommodate a potentially long list of entries, and finally, the ScrollView is set as the content of a Popup widget. The popup, complete with a custom background image, is then opened, providing the user with an interactive view of the Hall of Fame.

The $save\_to\_hall\_of\_fame()$ method is responsible for recording the current player's performance. It constructs a string entry containing the player's nickname, score, and level. The method first attempts to read existing entries from "$hall\_of\_fame.txt$". If the file is not found, it assumes there are no previous entries. Before appending the new entry, the method checks whether the entry already exists in order to avoid duplicates. If the entry is unique, it appends the entry to the file. The method also includes error handling to catch any Error that might occur during file operations, ensuring that any issues in saving the data are logged. This approach guarantees that the player's performance is recorded accurately and that the Hall of Fame reflects the latest game outcomes without redundancy.

Together, these methods provide a robust system for maintaining and displaying the Hall of Fame, offering players both a record of past performances and a sense of achievement as they see their scores alongside previous entries.

3.2 Cannon Module

The Cannon class is a dedicated component responsible for both rendering the cannon's visual representation and managing its rotational behavior, which directly affects the projectile's launch parameters. Upon initialization, the class accepts a starting position (representing the logical center of the cannon), an initial angle (defaulting to 30 degrees if not specified) and defines a fixed barrel length of 60 units. These parameters are essential as they determine both the initial orientation of the cannon and the precise starting point from which projectiles are fired.

In the $init$ method, the cannon sets up its visual appearance using Kivy's canvas instructions. The class makes use of a $PushMatrix/PopMatrix$ block to ensure that the rotation transformation is applied only to the cannon image. Specifically, a Rotate instruction is used with the current angle and the specified origin (the cannon's center position). The cannon is then drawn as a Rectangle with a defined source image, size, and position. Notice that the position of the Rectangle is adjusted (by subtracting fixed offsets) to properly center the image around the given logical position.

The $rotate()$ method allows the cannon's orientation to be dynamically adjusted. Depending on whether the direction parameter is "up" or "down," the method increases or decreases the cannon's angle by 5 degrees, while also ensuring the angle remains within defined limits (not exceeding 360° when increasing, and not dropping below -90° when decreasing). After modifying the angle, the method updates the Rotate

transformation accordingly and requests a canvas redraw via $canvas.ask\_update()$. This functionality is crucial since the main module calls the $rotate()$ method in response to key events (specifically, when the user presses the left or right arrow keys), thereby modifying the firing trajectory in real time.

The $get\_angle()$ method provides a simple interface to retrieve the current rotation angle of the cannon, ensuring that other parts of the program, such as the projectile launching mechanism, have access to the accurate orientation of the cannon.

Equally important is the $get\_tip\_position()$ method. This method calculates the exact tip of the cannon's barrel, which is used as the starting position for all fired projectiles. The method converts the current angle from degrees to radians using Python's $math.radians()$ method. It then applies basic trigonometric functions—$math.cos()$ and $math.sin()$—to determine the horizontal and vertical offsets from the cannon's central position, factoring in the barrel length plus an additional offset (10 units) for a precise launch point. The resulting tip position is returned as a two-element list representing the x and y coordinates.

The Cannon class integrates several mathematical operations—such as degree-to-radian conversion and the application of cosine and sine functions—to accurately compute the cannon's orientation and firing position. These computations directly feed into the main module, where the cannon's current angle and tip position are used to launch projectiles with the correct trajectory. The modular design of the Cannon class, with its clear separation of rendering, rotation, and position calculation responsibilities, ensures that the game's firing mechanics remain both accurate and easily maintainable.

3.3 Target Module

The Target class is a custom widget that represents a destructible target in the game. Designed to challenge the player, the class encapsulates both the visual and logical behavior of a target, ensuring it interacts dynamically with projectiles and the game environment. When a Target is instantiated, it is initialized with several key attributes: a reference to the main game, an image path for its visual representation, a starting position, and a size. Additionally, a movable flag indicates whether the target can move dynamically within the game space.

At initialization, the Target class converts the provided position into a mutable list and stores a copy as the initial position for later resets. If the target is set as movable, it receives random velocity components in both the horizontal and vertical directions; these velocities are adjusted to ensure minimal movement is noticeable, which makes the target more challenging to hit. The base collision radius is defined, and an image widget is created using the provided asset. This image widget is then centered on the target's logical position and added as a child, ensuring that the target is properly rendered on the screen.

The update method is an integral part of the class, called repeatedly within the game's main update loop. This method adjusts the target's position by incrementing it based on its velocity and the elapsed time (dt). It also includes boundary checking: if the target reaches any edge of the game window, its corresponding velocity component is inverted so that it bounces back into view. After updating the logical position, the method synchronizes the image widget's center with the new position, ensuring visual consistency.

Collision detection is handled by the collision method. This method computes the Euclidean distance between the target's center and a projectile's current position. By comparing this distance with the sum of an effective collision radius and the projectile's radius, the method determines if a collision has occurred. If the distance is less than this sum, a collision is registered.

When a collision is detected, the on_hit method is called to process the impact. Each hit reduces the target's health by one, and if the health falls to zero or below, the method returns True to indicate that the target has been destroyed and should be removed from the game. Otherwise, the target remains active, allowing for multiple hits before being eliminated.

Overall, the Target class provides a robust framework for creating dynamic, interactive targets within the game. Its careful integration of movement, collision detection, and hit processing contributes significantly to the gameplay experience, making it a critical component of the game's design.

3.4 Projectiles Module

The Projectile class is a central component that governs the behavior of all fired projectiles in the game, adapting its properties and physics based on the type of projectile selected by the player. When a Projectile instance is created, it is initialized with a specified type ("bullet", "bombshell", or "laser") and a starting position, which is usually determined by the tip of the cannon. The constructor sets up several key state variables: the projectile's current position (stored as a mutable list), an initial velocity vector (starting at [0, 0]), and flags for its active status and teleportation state. These variables ensure that each projectile maintains its own independent state during gameplay.

Based on the projectile type, the constructor configures specific parameters.

- For the **bullet**, the projectile is assigned a collision radius defined by a constant ($const.BULLET\_RADIUS$) and a corresponding image asset is loaded to visually represent it, along with a designated size.
- The **bombshell** is treated as a heavier projectile with a smaller size but with additional properties such as $bomb\_mass$ (set from $const.BOMB\_MASS$) and a penetration parameter ($bomb\_drill\_remaining$) that determines how far it can travel before being nullified by obstacles.
- The **laser** projectile is configured with a larger size, since it is longer, and its radius is calculated as half the width of the size to ensure a center-based collision detection. Unique to the laser is a timer ($laser\_timer$) that defines its duration in play (based on $const.LASER\_IMPULSE$), along with variables to track the total distance it has traveled.

Once the projectile's parameters are set, an Image widget is created and added to the projectile instance to render the appropriate visual asset on screen. The position of this image is carefully adjusted so that it remains centered on the projectile's logical coordinates.

The $launch()$ method is then used to set the projectile in motion. It converts the provided firing angle from degrees to radians using $math.radians()$, and then calculates the velocity components using the cosine and sine functions. Notably, the velocity is scaled by a multiplier that depends on the projectile type: lasers use a fixed speed determined by $const.LASER\_VEL$ (ignoring the power parameter altogether), bombshells multiply the power by 2 to simulate their greater mass, and bullets use a standard multiplier to produce a faster projectile. These multipliers and constants ensure that each projectile type behaves differently in terms of speed and trajectory, which is reflected in the game's physics.
The $update()$ method is crucial for simulating the projectile's motion over time. It first checks whether the projectile is still active, and then processes any teleportation cooldowns (relevant for interactions with wormholes). For non-laser projectiles, gravity is applied by reducing the vertical component of the velocity (using a gravitational constant of 9.8) on each update cycle. In contrast, the laser's timer is simply decremented to track its limited duration. The method then updates the projectile's logical position by scaling its velocity with the elapsed time (dt) and adding these displacements to its current coordinates.

Furthermore, the $update()$ method calculates the distance traveled during each update cycle by comparing the new position with the old one. For laser projectiles, this distance is accumulated, and if either the laser's timer expires or its total traveled distance exceeds a predefined constant ($const.LASER\_DIST$), the projectile is removed from play. Similarly, for bombshells, the remaining penetration distance ($bomb\_drill\_remaining$) is decremented by the distance moved, and once it reaches zero, the projectile is deactivated. The method also contains logic to remove the projectile if it exits an extended boundary of the game area, ensuring that off-screen objects do not consume resources.

Additional utility methods in the class include $is\_active()$, $get\_position()$, and $get\_radius()$, which provide external modules with access to the projectile's current state, position, and collision dimensions. Finally, $remove\_projectile()$ handles the deactivation process by removing the projectile's image widget from its parent and setting its active flag to False.

The Projectile class leverages a combination of mathematical functions (such as trigonometric calculations and Euclidean distance measurement) and constant parameters defined in a separate constants module to ensure that each projectile type exhibits distinct and realistic behavior. This detailed implementation not only governs the projectile's motion and interactions with obstacles but also directly influences the player's strategic decisions during gameplay, as each projectile type has unique advantages and limitations that are clearly reflected in the game mechanics.

3.5 Obstacles Module

The Obstacle class is designed to manage all non-target dynamic elements in the game's environment. When an Obstacle is created, it is assigned a specific type ("rock," "wormhole," "mirror," or "perpetio") and linked to the main game instance. In the constructor, several key parameters are initialized. First, the obstacle's logical center position is randomly determined within a designated portion of the screen, ensuring that obstacles appear in the appropriate region of the game. This position is stored as the initial position for later use, such as when the level is reset. Additionally, each obstacle is given a default collision radius of 30 units; however, this value can be effectively modified during collision detection depending on the obstacle type. For movable obstacles, random horizontal and vertical velocity components are also assigned, which allows them to traverse the game area and bounce off window edges.

The $update()$ method is responsible for continuously modifying the obstacle's position based on its velocity. With every call to update (which is tied to the game's main loop), the obstacle's x and y coordinates are incremented by the product of their respective velocity components and the time delta (dt). The method also includes logic to detect when an obstacle reaches the edge of the game window. In such cases, the corresponding velocity component is inverted, causing the obstacle to bounce off the boundary. Finally, the obstacle's visual representation is kept in sync with its logical position by re-centering its image widget after each update.

Collision detection is handled by the $collision()$ method. This method calculates the Euclidean distance between the center of the obstacle and the center of a projectile. A collision is detected if the computed distance is less than the sum of the obstacle's effective radius and the projectile's own collision radius. This straightforward mathematical approach ensures reliable detection of impacts between projectiles and obstacles.

The $on\_hit()$ method defines how an obstacle responds when struck by a projectile. For destructible obstacles like rocks, this method decrements the obstacle's health by one. When the health reaches zero or falls below, the method signals that the obstacle should be removed from play. For other obstacle types that are not meant

to be destroyed, the method simply returns False, indicating that no removal should occur. This logic allows the game to differentiate between obstacles that are intended to be temporary challenges and those that persist as part of the environment.

Mirror obstacles implement a different interaction through the $projectile\_reflection()$ method. When a projectile collides with a mirror, the method checks the type of projectile involved. For bullets and bombshells, the mirror causes the projectile to be removed from the game by deleting its image widget and marking it as inactive. However, for laser projectiles, rather than removing them, the method inverts both the horizontal and vertical components of their velocity. This inversion causes the laser to reflect off the mirror, continuing its trajectory in the opposite direction.

4. Difficulties and Solutions

During development, I encountered several technical issues that required modifications to the code.

- First, when I pressed the Reset button, I observed that the targets and rocks were not returning to their initial positions. The image widgets were not aligning with the logical positions of the obstacles. I determined that this issue was due to the lack of synchronization between the stored initial positions and the positions of the image widgets. To resolve this, I modified the reset logic so that both the obstacle's logical coordinates and its image widget's center are updated based on the initially stored positions.

- Next, I noticed that the trajectory preview was not being displayed in the second and third levels. Although the countdown label appeared correctly, the dashed trajectory line did not render. After analysis, I identified that widget layering, and coordinate synchronization were the likely causes. I addressed this by ensuring that the trajectory widget is re-added to the layout at the highest index and explicitly setting its size and position to match the layout. This change ensured that the trajectory drawing is rendered above all other widgets.

- Another issue was that sometimes only the visual image of a projectile was removed, while the projectile itself remained active within the game logic. This discrepancy led to unwanted behavior during collision detection. I resolved this by modifying the removal method so that it both removes the image widget from its parent and marks the projectile as inactive, ensuring that no residual projectiles continue to affect the game state.

- I also encountered a problem with the premature removal of projectiles due to the boundary-checking logic. The original implementation removed projectiles as soon as they reached specific points in the game map. I addressed this by introducing a much larger margin beyond the visible game area. This modification prevents the projectile from being eliminated before it has fully left the extended boundaries of the game.

- Finally, the game-over condition was being triggered immediately after the last projectile was fired—even if that final shot successfully hit the last target—and it was also erroneously firing in subsequent levels. To address this, I revised the logic so that the game-over check is delayed until all active projectiles have been fully processed and verified that no targets remain, while also canceling any pending game-over events during level transitions. This ensures that the game-over state is reached only when truly appropriate.

Throughout the development process, I integrated extensive print statements to log key actions and state changes in the terminal. This practice was invaluable for verifying that each component of the game was functioning as expected. For example, I printed messages when a projectile was launched, when collisions were detected, and when objects such as targets or obstacles were reset or removed. These debug outputs provided real-time feedback on the game's internal state, which allowed me to trace the sequence of events and identify any discrepancies between the visual behavior and the underlying logic.

Using print statements for logging is a fundamental debugging technique, especially in complex systems like a game where multiple modules interact dynamically. It enabled me to confirm that methods were being called at the right moments and that the calculated values (such as positions, velocities, and collision distances) were within expected ranges. Moreover, these logs helped pinpoint the exact location and cause of issues, such as when the trajectory preview was not appearing in certain levels or when objects were not being properly reset. By monitoring the terminal output, I could rapidly adjust and refine the code, ultimately ensuring a more robust and stable game environment.

5. Conclusion

This project exemplifies the practical application of object-oriented programming principles in a structured and modular development environment. The integration of event-driven programming, real-time physics calculations, and user interaction mechanisms required a methodical approach to both software architecture and algorithm optimization. The use of a layered modular design not only ensured code maintainability but also facilitated the scalability of the system, reinforcing best practices in software engineering.

Beyond the technical implementation, this work highlights the challenges inherent in developing interactive systems where computational efficiency, responsiveness, and precision are critical. The debugging process underscored the importance of systematic testing and real-time logging, demonstrating how iterative refinement contributes to software robustness. Ultimately, this project serves as a case study in the intersection of computational methods and interactive system design, offering valuable insights for both game development and broader applications in real-time simulation.