Report: Cannon – Final Exam Project

Computer programming, Algorithms and Data Structures, Mod.1

Project: ToonTanks

Rebecca Bellaviti (Matr. 536905)

Abstract

This project is a 2D artillery game developed using Python and the Kivy framework. The game features three themed levels (*The Simpsons, Futurama, SpongeBob*) where the player controls a tank, aiming to hit a target. The design progressively introduces new mechanics, including dynamic obstacles, mirror-based laser reflections, and wormhole teleportation. The player has a choice of three weapons: a bullet, a bomb, and a laser. Key development goals included a modular screen/widget architecture, a stable 60 FPS update loop, and clear audio/visual feedback. This report provides an overview of the design, implementation, and key outcomes from testing and debugging.

Content

I. Aim of the Project

From a technical standpoint, I built this project with a clean, modular architecture that separates different components. Screens are responsible for managing flow and layout, while custom widgets handle specific entities like the tank, projectiles, mirrors, and wormholes. I set explicit engineering targets: reproducible projectile motion, segment-to-segment intersection for laser reflections, consistent teleport rules for wormholes. I also focused on robust hit detection against dynamic obstacles, all while ensuring minimal side effects and a clear flow of data throughout the system.

My second goal was to create a great player experience. The HUD provides real-time feedback on angle, power, ammunitions and shots. I also designed helpful on-screen overlays for each level and a simple level-select screen that tracks player progress.
The audio is designed to be purposeful, with themed looping background tracks for each level and one-shot sound effects for firing, winning, or losing to enhance the experience without overwhelming it.

Finally, the project aims to leave space for future growth—difficulty tuning, additional puzzle variants, and accessibility options—while delivering a coherent, shippable slice now. The result should feel like a small, replayable artillery puzzler and a teachable codebase that documents practical patterns for Kivy-based games.

When I started this project, my programming experience was limited. However, as I built it, I began to grasp concepts I'd only read about, like managing complex object interactions, ensuring data flows cleanly, and debugging issues. This project was a hands-on experience that solidified my theoretical knowledge into practical skills.

II. Gameplay and Game Design

The core idea of the game is to recreate the experience of a classic artillery challenge, enriched with progressively more complex elements. The player faces a series of themed worlds: “Simpson”, “Futurama” and “SpongeBob”. The gameplay is structured in short, self-contained levels that encourage experimentation with different weapons and trajectories. The goal is not only shooting accuracy, but also the ability to combine timing, resource management, and spatial reasoning.

Level designs follow gradual progression, with each scenario introducing new challenges and mechanics.

The first level has an introductory function: the player learns how to control the tank, adjusts the firing angle and familiarizes with the physics of the ammunitions. The obstacles are static and destructible and serve to guide the player the basic rules.

In level 2 the difficulty increases with the introduction of more dynamic obstacles, which force the player not to think only about the trajectory but also about the obstacles to avoid.

The third level represents the most complex stage: elements such as teleportation and multiple obstacles require planning and sometimes several attempts are needed to complete it. In this way, the game culminates in a challenge that integrates all the skills learned in previous levels.

The variety of weapons available allows for different approaches to the same situation, encouraging experimentation and comparison between attempts. The hall of fame, based on the numbers of shots used, adds a competitive element and rewards accuracy and optimization.

III. Project's file organization

The project is organized into different folders and files, to make the visualization easier and more comprehensible:

- **`main.py`**: the app's entry point. It defines the Kivy App, builds the ScreenManager, applies window/config settings, wires global key bindings, and starts the game loop.

- **`/screens`**: contains all the Kivy Screen classes. Each file defines a single screen with layout, UI logic, and callbacks to the ScreenManager and game modules (audio, saves, levels):
    - **`start_screen.py`:** Displays background, text field for username, and three buttons: "Play", "Hall of Fame", and "How to Play". Verifies that the name is valid and not already in the Hall of Fame before starting the game. Synchronizes the field with App.player_name, handles error popups and instructions, and redirects to level selection or hall of fame.

    - **`hall_of_fame_screen.py`**: Manages the Hall of Fame: saves/loads records from data/hof.json, records win, displays a scrollable table of scores, and clears the player's status when leaving the screen. Includes a Back button to return to Start screen.

    - **`screen_manager.py`:** Defines ScreenManagement, a subclass of ScreenManager. It registers and links all the main screens: `StartScreen, HallOfFameScreen, LevelSelectScreen`, and the three levels. It thus provides the central navigation of the app.

    - **`level_select_screen.py`**: displays variable background and three buttons (Play/Locked) based on progress. Allows access to levels 1 to 3 and updates the UI when levels 2 and 3 are unlocked.

    - **`level1_screen.py`**: Core gameplay prototype. Manages tank controls, two ammunition types (bullet and bomb), a single static Perpetio obstacle, destructible rocks, target hit logic, HUD, background music, and win/lose flows. Handles precise SFX (shot, bomb, win/lose), input binding, and a per-frame update() loop with cleanup via reset_level().

- **`level2_screen.py`** — Adds complexity: multiple Perpetio obstacles, reflective Mirrors (for laser interactions), and tri-ammo cycling (bullet, bomb and laser). Implements segment math for laser lines, reflection with jitter, and the same HUD/music/game-state lifecycle as Level 1.

- **`level3_screen.py`** — Puzzle layer. Spawns three vertically oscillating MovingPerpetio near the target, reflective Mirrors, and a Wormhole pair (entry random, exit fixed near target) that can route a well-aimed shot to victory. Includes ammunitions cycling (bullet/bomb/laser), teleport checks, bounce limits for lasers, and robust start/stop of the main loop.

- **`/images`**: All visual assets used throughout the game, including level backgrounds, sprites (targets, tank, explosions), icons, and other imagery for effects and menus.

- **`/sounds`**: All audio assets used throughout the game, including level background music, sound effects for shots, explosions and laser and win/lose jingles.

- **`/fonts`**: Custom font assets used in the game interface (HUD, buttons, popups and labels) to ensure a consistent retro visual identity across all screens.

- **`/constants`**: which holds the file `/screen_constants.py` and contains global screen constants used by all screens to maintain consistent resolution and refresh rate.

- **`physics.py:`** It collects constants and utilities of game physics, such as gravity, velocity, bounces, and functions to update trajectories and collisions (reused by bullets, bombs, and lasers).

I chose to integrate the code for the tank, obstacles, and ammunition directly within each level's logic rather than creating separate classes for them. This was because implementing them within each level was straightforward. Furthermore, a unified tank class was impractical as the tank's color varies between levels. The obstacles also differ as they are uniquely designed to match each level's layout. While the ammunition is consistent across Level 2 and 3, Level 1 notably lacks the laser type, making a shared ammunition module less necessary.

IV. Details of implementation

Ammunitions:

- **`Bullet`**: It's a class derived from Image that represents a standard bullet in the game, with mass BULLET_MASS and impact radius BULLET_RADIUS. The initial velocity is calculated using `get_initial_velocity(angle, speed)` in horizontal (vx) and vertical (vy) components. The trajectory is parabolic, subject to constant acceleration GRAVITY.
  The `move(dt)` method updates position and velocity at each frame, while `impact()` sets

has_impacted=True, stopping the motion.

- **Bomb**: It's a class derived from Image that shows high-mass projectile (BOMB_MASS) with an impact radius BOMB_RADIUS and vertical penetration BOMB_DRILL. The initial velocity is calculated similarly to Bullet.
  The move(dt) method updates position and velocity according to the gravitational parabola; upon contact with the ground, y is decremented by drill before activating impact().

- **Laser**: It's a class derived from Widget that represents an energy beam in the game, its graphics are drawn by draw_laser() and updated via update_graphics(). It is initialized with a position (x, y) and a firing angle.
  The main attributes include:
  - angle: firing angle of the laser.
  - velocity_x, velocity_y: velocity components calculated from the direction and constant speed LASER_VEL.
  - damage_radius and impulse: laser effect radius (LASER_DIST) and force of the shot (LASER_IMPULSE).
  - _distance_travelled and _max_distance: keep track of the distance traveled and the maximum distance allowed.
  - bounces and max_bounces: number of bounces allowed against obstacles.
  - has_impacted: flag indicating whether the laser has reached the end of its useful life.

  The move(dt) method advances the laser along a straight trajectory, tracking the distance traveled. Upon exceeding window bounds or maximum range, impact() sets has_impacted=True and removes the widget. Unlike Bullet and Bomb, the laser ignores gravity, moves linearly, and includes bounce handling and canvas-based visual effects.

Obstacles:

- **Rock**: Simplest obstacle; implemented as:
  - RockBlock: It is a class derived from Image that represents a fixed obstacle in the game scenario. Each block has predefined dimensions and a logically destroyed status that indicates its presence or removal. The destroy() method marks the block as destroyed and removes it from the parent widget. The implementation of the class remains unchanged between the different levels, while the associated image (source) changes to match the graphic theme of the level.
  - RockField: is a class derived from Widget that manages the generation and positioning of multiple RockBlocks within the scenario. The generate_blocks() method creates a specified number of blocks and distributes them randomly throughout the game space, avoiding overlaps between obstacles or predefined areas. The class also integrates a check_collision(projectile) method that

checks for intersections between a projectile and the blocks present: in the event of a collision, the block that is hit is removed.

- **`Perpetio`**: is a class derived from Image that represents an indestructible obstacle. It maintains fixed dimensions and the attribute indestructible = True, which prevents it from being removed following impacts. It's present in Level 1 and Level 2 with the same implementation: in the first level it is unique and fixed in position, while in the second there are many in random spots.

- **`Mirror`**: is a class derived from Image that represents a reflective obstacle which only interacts with Laser, cannot be destroyed by any ammunition and is only present in levels 2 and 3, where it introduces the mechanics of shot reflection. It's not perfectly deterministic but includes a random component that introduces variability in the angle of rebound: `_reflect_laser_angle_random` calculates the angle at which the laser is reflected off the mirror, applying the geometric law of reflection and introducing random jitter to make the bounce unpredictable.

- **`MovingPerpetio`**: is a subclass of Perpetio that introduces dynamic behavior: the obstacle is not static but oscillates vertically along a sinusoidal trajectory.
  The oscillation is parameterized by amplitude, frequency, and phase, which determines its vertical movement over time. This makes the obstacle a more complex and unpredictable variant, capable of increasing the difficulty of the level thanks to its motion.

- **`Wormhole`**: is a class derived from Image present only in Level 3 that represents a teleportation system based on two portals each derived from Image. Both portals are indestructible but have different positioning logic: the entrance portal is generated in a random position on the screen, while the exit portal is always located in a fixed position near the target. When a projectile passes through the entrance portal, it is instantly transferred to the exit portal, maintaining its original speed and direction.

- **`Tank`**: is a class derived from Image that represents the player. The implementation is identical in all levels but changes the image source based on the level theme. It can move forward and backward to the middle of the screen, fire ammunition, and visually update the angle of the cannon based on the direction of fire.

- **`Target`**: is not a class but is defined directly in the LevelScreen class of each level. It's an image themed for the level and must be hit to win. It has a fixed position and size. Together with the tank, it does not risk overlapping obstacles: during level generation, the areas occupied by the tank and target are treated as "areas to avoid" by the `generate_blocks()` function, which places blocks while avoiding these areas.

V. Most important functions

In the project, various functions and methods were implemented that form the logical backbone of the game:

- **`move():`** used by all types of projectiles. Updates the position taking into account speed and trajectory. For Bullets and Bombs, the trajectory follows a parabolic path influenced by gravity, while for Lasers, the movement is linear and reflection is managed in the event of collision with Mirror-type obstacles.

- **`fire():`** method associated with the tank, responsible for instantiating the correct projectile based on the weapon selected by the player. This ensures consistent firing logic, regardless of the type of weapon used.

- **`check_collision`**: function that checks whether a projectile encounters an obstacle or a target. The logic varies depending on the object hit: some obstacles are indestructible, others not and others reflect, while target determines the victory of the level.

- **`update_canvas:`** Responsible for graphics. It redraws the main game elements on the canvas based on their updated status.

VI. Testing and debugging

The game underwent a series of tests to ensure that all levels, obstacles, and weapons functioned correctly. The behavior of projectiles (parabolic and laser), collisions with obstacles, laser rebounds on mirrors, and the correct recording of scores in the Hall of Fame were all verified.

During development, several bugs were found, such as projectiles passing through obstacles, occasional crashes during level selection and errors in laser reflection management. These issues have been solved by updating collision logic, managing screen boundaries and synchronizing the canvas with the status of widgets.

At the end of the testing process the game guarantees smooth and constant updating, without visible slowdowns, with all levels functioning correctly and stable flow between screens.

VII. Libraries and References used

I used Python (v. 3.12.10) and Kivy (2.3.0).

The following references include the main documentation, resources, and assets consulted during the development of the project:

- Python official documentation
- Kivy official documentation

- Visual Studio Code
- AI assistant (such as Chat GPT and Copilot) to understand and fix difficult bugs
- Canva to create images of level selection, help center and hall of fame background.

VIII. Possible Improvements

The current version of the game offers a complete and playable experience, but it has some limitations. The tank mechanics are simplified and do not include advanced controls. Level design is limited to three scenarios, which may reduce long-term replayability. Additionally, the Hall of Fame is based solely on the number of shots used, without considering other performance metrics.

Possible future developments include adding new levels with different puzzle mechanics and introducing advanced scoring systems that also reward speed. Other improvements could include customizable difficulty settings and a greater variety of assets (sounds, themes, and special effects) to enrich the player's experience.

During the execution of the code a warning message appeard: "`Deprecated property "<BooleanProperty name=allow_stretch>" of object "<kivy.uix.image.Image object at 0x0000027621089A20>" has been set, it will be removed in a future version`". This warning relates to the allow_stretch property of Kivy's Image class, which is deprecated. I am aware that this feature will be removed in future versions; however, with the version used, it does not cause any functional or graphical problems in the game. A possible improvement could be to update the code by replacing allow_stretch with the new supported properties to ensure compatibility and maintainability.