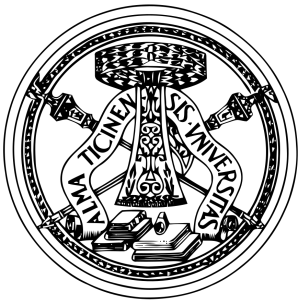

Università degli Studi di Pavia

Bachelor in Artificial Intelligence

Computer Programming, Algorithms and Data Structures
Module 1 Final Project

Cannon Game in Python Kivy Framework

Mucaj Deva e Gentile Martina

project name

| Invasion: Antibody Odyssey |

Contents

---

1. Goals of the project

Objective

The primary objective of this project is to develop a fully immersive, interactive, and engaging game, titled "Invasion: Antibody Odyssey", using the Kivy framework. The game is designed to provide players with a captivating experience through various gameplay mechanics, including projectile shooting, obstacle navigation, and strategic level progression.

Game Concept and Features

Gameplay Mechanics

1. Projectile Shooting:

Variety of Projectiles: players can choose from different types of projectiles, including standard projectiles, bombshells, and lasers. Each projectile type has unique properties that affect its behavior and interaction with the game environment.
Trajectory Calculation: the trajectory of each projectile is influenced by factors such as launch angle, speed, gravity, and external forces (e.g., gravitational fields from Gravitonio objects). Players must adjust their aim and force to hit targets accurately.

2. Obstacles and Challenges:

Static and Dynamic Obstacles: the game features a variety of obstacles that players must navigate or destroy to progress. Static obstacles remain fixed, while dynamic obstacles move or oscillate, adding complexity to the gameplay.
Mirror Mechanism: mirrors within the game can reflect laser shots. Players must strategically position their shots to utilize reflections and hit targets indirectly, or to avoid them.
Collidable Objects: obstacles, Gravitonios, Elastonios, Wormholes, Perpetios and Mirrors can collide with the different projectiles, causing deflections or destruction, requiring players to carefully plan their shots.

3. Interactive Elements:

Gravitonio Objects: these objects exert repulsive forces on projectiles, altering their trajectories. Players must account for these forces when aiming and shooting.
Explosions and Effects: bombshells explode upon impact. Visual effects enhance the sense of the impact.

User Interface and User Experience

1. Main Menu and Navigation:

Custom Background: the main menu features a visually appealing background that sets the game's theme and atmosphere.
Interactive Buttons: rounded buttons with integrated text labels provide a user-friendly interface. Buttons allow players to start the game, view the hall of fame, access help instructions, and navigate between screens.

2. In-Game Interface:

HUD Elements: the in-game interface includes a heads-up display (HUD) that shows essential information such as the current level, selected projectile type, remaining shots, and score.
Real-Time Feedback: players receive real-time updates and feedback on their actions, such as successful hits and remaining obstacles. This feedback helps players adjust their strategies and improve their performance.

Level Design and Progression

1. Level Structure:

Incremental Difficulty: the game is structured into two levels, each progressively increasing in difficulty.
Varied Environments: each level features a unique environment with distinct obstacles, layouts, and themes, keeping the gameplay fresh and engaging.

2. Victory and Progression:

Objective: the primary objective of each level is to hit all the targets using the least number of shots. Players must overcome obstacles and utilize their strategic thinking to achieve this goal.
Level Advancement: successfully completing a level unlocks the next one. Players are motivated to progress through levels to experience new challenges and environments.
Scoring System: a scoring system tracks players' performance based on accuracy, efficiency, and speed. High scores provide a sense of accomplishment and encourage replayability.

Audio and Visual Design

1. Graphics and Animation:

High-Quality Graphics: the game features detailed and vibrant graphics, including backgrounds, projectiles and obstacles images, and interactive elements. The visual design enhances the overall themed aesthetic and immersiveness of the game.
Smooth Animations: animations for projectile movements, explosions, and other interactions are smooth and fluid, providing a dynamic and engaging visual experience.

2. Sound and Music:

Background Music: carefully selected soundtracks play in the background, enhancing the game's atmosphere and providing an immersive auditory experience.

Conclusion

The goal of "Invasion: Antibody Odyssey" is to create an engaging and interactive gaming experience that challenges players' strategic thinking, precision, and problem-solving skills. By combining gameplay mechanics, an intuitive user interface, and high-quality audio-visual elements, the game aims to provide entertainment and a satisfying sense of progression and achievement for players. Through its design and dynamic gameplay, "Invasion: Antibody Odyssey" aspires to be an enjoyable experience.

2. Assumptions, Method and Procedures

Adaptable Design

A series of assumptions, methodologies and procedures were implemented to ensure the game's robustness and adaptability. Methodologically, the project leverages the Kivy framework, useful for its cross-platform capabilities and its efficacy in developing graphical user interfaces with Python.

Modular Game Development

Procedurally, the game development follows a modular approach. This involves breaking down the game into distinct sections: projectile dynamics, obstacle interactions, and game functionalities such as saving/loading progress and maintaining a Hall of Fame. Each module is designed with specific parameters—such as the mass and velocity of projectiles—to simulate different behaviors and interactions within the game environment.

Balancing Game Complexity

The game's design balances simplicity and complexity: while the core mechanics are straightforward, involving aiming and shooting at a target, the inclusion of various projectile types and interactive obstacles adds layers of strategy and challenge. Projectiles like bullets, bombshells, and lasers each have unique properties and effects, necessitating different approaches and tactics from the player. Obstacles, ranging from destructible rocks to reflective mirrors and gravitational fields, further enrich the gameplay.

3. Logical structure of the code

Code Analysis Report

The logical structure of the code can be divided into several components:

1. Configuration Setup

- *from kivy.config import Config*: sets the Kivy application to run in fullscreen mode.

2. Imports

Various Kivy modules and standard Python libraries are imported, including `math`, `random`, and `time`:

- *from kivy.app import App*: imports the App class from the Kivy library, which is the base class for creating Kivy applications.
- *from kivy.uix.screenmanager import ScreenManager, Screen, FadeTransition*: imports ScreenManager, Screen, and FadeTransition from Kivy.
    - ScreenManager manages multiple screens in an application.
    - Screen is a class for creating different screens within the ScreenManager.
    - FadeTransition provides a fade transition effect between screens.
- *from kivy.uix.widget import Widget*: imports the Widget class, which is the base class for creating UI elements in Kivy.
- *from kivy.graphics import Ellipse, Line, Color, Rectangle, PushMatrix, PopMatrix, Rotate, RoundedRectangle:* imports various graphical primitives and transformations from Kivy's graphics module.
    - Ellipse, Line, Rectangle, and RoundedRectangle are shapes that can be drawn.
    - Color sets the color for subsequent drawing.
    - PushMatrix and PopMatrix manage the transformation stack.
    - Rotate applies rotation to subsequent drawing commands.
- *from kivy.core.window import Window*: imports the Window class, which provides access to window properties and methods.

- *from kivy.clock import Clock*: imports the Clock class, which is used for scheduling tasks and events.
- *from kivy.core.audio import SoundLoader*: imports the SoundLoader class, which is used to load and play sound files.
- *from kivy.uix.label import Label*: imports the Label class, which is used to display text.
- *from kivy.uix.popup import Popup*: imports the Popup class, which is used to create popup dialogs.
- *from kivy.utils import get_color_from_hex*: import get_color_from_hex, a utility function to convert hex color strings to Kivy's color format.
- *from kivy.uix.behaviors import ButtonBehavior*: imports the ButtonBehavior class, which can be mixed into other widgets to give them button-like behavior.
- *from kivy.uix.boxlayout import BoxLayout*: imports the BoxLayout class, which arranges children in a horizontal or vertical box.
- *from kivy.uix.button import Button*: imports the Button class, which creates a clickable button widget.
- *from kivy.vector import Vector*: imports the Vector class, which provides vector operations.
- *from kivy.logger import Logger*: imports the Logger class, which provides logging functionality.
- *import math*: imports the standard Python math library for mathematical operations.
- *import random*: imports the standard Python random library for generating random numbers.
- *import time*: imports the standard Python time library for time-related functions.
- *from kivy.properties import NumericProperty, StringProperty*: imports NumericProperty and StringProperty, which are Kivy properties for creating reactive attributes.
- *from kivy.uix.textinput import TextInput*: imports the TextInput class, which creates an editable text input field.
- *import json*: imports the standard Python json library for parsing and manipulating JSON data.
- *import logging*: imports the standard Python logging library for configuring and using loggers.
- *import os*: imports the standard Python os library for interacting with the operating system, such as file and directory operations.

3. Widget Definitions

- **Projectile**: represents a basic projectile with properties like size and velocity, and methods to update its graphical representation.

- **Bombshell**: inherits from `Projectile`, adding properties like mass, speed, and angle. It includes methods for trajectory calculation, explosion behavior, and gravitational effect handling.
- **Laser**: represents a laser shot with properties like speed, and angle. It includes methods to update its graphical representation and trajectory.
- **MirrorBulletproof**: represents a reflective mirror that can reflect lasers and cannot be destroyed by projectiles and bombshells.
- **Elastonio**: represents an object that can collide with projectiles, bombshells, and lasers. Projectiles and Bombshells are "reflected" by it, while Lasers are able to destroy it.
- **Obstacle**: represents an obstacle with properties for oscillation that can be destroyed by every type of projectile.
- **Perpetio**: represents a fixed obstacle that can't in any way be destroyed.
- **Gravitonio**: represents a gravitational object that affects projectiles' and bombshells' trajectories.
- **MainMenuBackground**: represents the background of the main menu.
- **RoundedButton**: a custom button with rounded corners and a label.
- **GameWidget**: the main game logic is implemented here. It includes keyboard and mouse event handling, game object creation, and updates for projectiles, lasers, bombshells, obstacles, and other elements.
- **Target**: represents a moving target in a game.
- **Obstacle**: provides a framework for creating oscillating obstacles in a game.
- **Wormhole**: create a pair of interconnected wormholes that can transport objects between them.

4. Screen Definitions

- **MainMenu**: represents the main menu screen with buttons for playing the game, viewing the hall of fame, and help instructions. It handles button events and displays a popup with instructions.
- **GameScreen**: represents the game screen and includes the `GameWidget`.
- **HallOfFame**: represents the Hall of Fame, where the 10 best scores of the game are saved and shown.
- **StorylineScreen**: the class sets up a list of images that constitute the storyline, where the player can press anywhere in the window (or press enter) to navigate through the storyline. It also includes a "Skip" button that allows users to bypass the storyline and navigate directly to the main menu.
- **SplashScreen**: contains a welcome label to the screen and an initial splash image. It also plays a background music file for the splash screen, set to loop continuously, providing an engaging audio backdrop while the splash screen is visible.

5. Pop Up Class

- **HallOfFamePopup**: display a popup where players can enter their name and save their score to a Hall of Fame after achieving a complete score in the game.
- **PauseMenuPopup**: designed to create a popup menu that appears when the game is paused when the player presses "ESC" button during the game.

6. Application Class
- **MyApp**: the main application class that sets up the screen manager with the main menu and game screens. It runs the application.

Logical Structure and Flow

1. Initialization
- The application initializes by configuring the screen to full screen mode.
- Imports necessary Kivy modules and Python libraries.

2. Class Definitions
- Defines various widgets for game elements like `Projectile`, `Bombshell`, `Laser`, `MirrorBulletproof`, `Elastonio`, `Perpetio`, and `Gravitonio`.
- Defines the main menu background and custom buttons.

3. Main Menu
- The main menu screen (`MainMenu`) is created with buttons for navigation. Each button has a specific action, such as starting the game, viewing the hall of fame, or showing help instructions.

4. Game Logic
- The `GameWidget` class handles the core game logic, including:
    - Keyboard and mouse event handling for player input.
    - Updating game elements like projectiles, lasers, bombshells, and obstacles.
    - Managing game levels and progression.
    - Collision detection and response for various game objects.

5. Screen Management
- The `ScreenManager` switches between the main menu and game screens.

6. Running the Application

- The `MyApp` class initializes the application and runs it.

Interaction Flow

1. User starts the application: the main menu is displayed.
2. User interacts with the main menu: selects to play the game, view the hall of fame, or read help instructions.
3. Game starts: `GameWidget` initializes and sets up the game environment.
4. Gameplay: user controls the cannon, shoots projectiles, lasers, or bombshells. The game updates object positions, checks for collisions, and manages game levels.
5. Game progression: the game advances levels as obstacles are cleared.
6. End of the game: upon completing the second level, a victory popup is displayed.

Conclusion

The code defines a complete game with a structured flow from the main menu to gameplay, including various game elements and interactions. It uses Kivy's widget system to create and manage game objects, handle user input, and update the game state. The use of object-oriented principles allows for modular and maintainable code.

4. Implementation

The implementation of the project involves several key steps, including designing the game architecture, developing game functionalities, and ensuring proper visualization.

Game Architecture

It is grounded in a modular approach that separates the game's core components into distinct, manageable sections. The game's architecture includes modules for handling the physics of projectile motion, collision detection with obstacles, and the dynamics of different projectile types such as bullets, bombshells, and lasers. Each projectile type has specific properties and behaviors that are managed within their respective modules. The cannon's controls, which allow the player to adjust the angle and velocity of the shots, are another critical part of the architecture. Additionally, the game features a robust system for managing game states, including saving and loading game progress, displaying the Hall of Fame, and providing help instructions. Visualization is achieved through a single-screen interface that can be extended to larger spaces if necessary, ensuring that all game elements are cohesively integrated and easily accessible to the player

Game functionalities

Key functionalities include the ability to save and load game progress, ensuring that players can resume their game at any point during the play without losing their achievements. The "Help" button is designed to assist players in understanding game mechanics and controls, ensuring accessibility for all users. These core functionalities are integrated into the game using the Kivy framework, which allows for smooth execution and interaction. The focus on these features ensures that the game is not only enjoyable but also user-friendly.

Visualization

The game is designed to be displayed in a single-screen view, ensuring that the entire game field is visible to the player at all times. This design choice helps players easily track the cannon, projectiles, and target within the game environment. The Kivy framework is employed to handle the graphical user interface, utilizing its tools to render smooth animations and interactive elements. This structured approach to visualization ensures that the game remains accessible and visually appealing across different devices and screen sizes

Tools and Resources

Various tools and resources were crucial in the development and implementation of the game. The primary tool utilized was the Kivy framework, a versatile and open-source Python library designed for developing multi-touch applications. Kivy flexibility was essential for ensuring that the game could run on any system equipped with a Python 3 interpreter and the Kivy framework. Comprehensive documentation and community support for Kivy were available through its official website and GitHub repository, facilitating a smoother development process. These resources collectively supported the creation of a robust and interactive gaming experience for the Cannon project.

5. Libraries

A diverse set of libraries from the Kivy framework was utilized to build the application's graphical user interface and interactive features.

- The kivy.config module was employed to configure the application settings, while kivy.app provided the core application management functionalities.
- The kivy.uix.screenmanager was instrumental in managing multiple screens within the app, using ScreenManager and Screen classes, with transitions such as FadeTransition to enhance user experience.
- The kivy.uix.widget and kivy.graphics modules facilitated the creation and manipulation of custom graphical elements, including shapes like Ellipse, Line, and Rectangle, and transformations such as Rotate.

- For audio functionalities, kivy.core.audio allows integration of sound effects, enhancing the interactivity of the application.
- To handle user input and display, widgets such as Label, TextInput, and Button from kivy.uix were utilized.
- The kivy.uix.popup module provided a means to display popup messages for user notifications.
- Additionally, kivy.utils helped with color management, and kivy.vector was used for vector operations.
- The kivy.logger was employed to handle and log application events, aiding in debugging and maintenance. Supporting libraries such as json, os, and time were also used for data handling and time-related functionalities.

Together, these libraries facilitated the development of a dynamic and responsive application interface.

6. References

The development of "Invasion: Antibody Odyssey" involved the exploration of various resources and literature to inform the design and implementation of the game.
Key references are:

1. Kivy Documentation: the official Kivy documentation was crucial for understanding the framework's capabilities and functionalities. It provided detailed guidance on using Kivy modules and classes effectively, which was pivotal in implementing the game's graphical user interface and interactive elements. The documentation is available at Kivy Documentation.
2. Kivy GitHub Repository: the GitHub repository for Kivy offered valuable insights and updates on the latest developments and community-contributed features. It also served as a source of troubleshooting and examples for common issues encountered during development. Access the repository at Kivy GitHub.
3. Python Standard Library Documentation: for general programming and integration tasks, the Python Standard Library documentation was referenced. It provided essential information on modules such as json, os, and time, which were used for data handling and time-related functionalities. This documentation is accessible at Python Standard Library.
4. Textbooks: relevant textbooks on game design and development principles were reviewed to guide the creation of engaging and well-balanced gameplay mechanics. These resources offered theoretical foundations that aided the game's implementation strategies.

Book used: *Think Python: "How to Think Like a Computer Scientist by Allen B. Downey", 2015*

These references collectively supported the project by providing essential knowledge, practical examples, and community support.

7. Testing and Results

The testing phase for the game was crucial to ensure its functionality, performance, and user experience. A multi-faceted approach was employed to validate different aspects of the game:

Unit Testing
Individual components of the game, including projectiles, obstacles, and interactive elements, were tested separately to verify their behavior and interaction. Unit tests confirmed that each component adhered to its defined properties and functions, such as projectile trajectory and obstacle collision detection.

Integration Testing
The integration of various game elements was examined to ensure that they worked seamlessly together. This involved checking interactions between projectiles and obstacles, the correct update of game state, and the accurate rendering of visual and audio effects. Any issues detected were addressed to prevent disruptions in gameplay.

Functional Testing
Functional testing focused on verifying that the game operates according to its design specifications. This included confirming that core gameplay mechanics, such as projectile shooting and trajectory calculation, worked as intended. We also checked the game's progression system to ensure that levels unlocked properly and that transitions between different stages of the game occurred smoothly. Additionally, we examined the user interface to ensure that all elements—such as buttons, the heads-up display (HUD), and menus—responded correctly to user interactions and displayed accurate information.

Performance Testing
Performance testing aimed to assess how well the game performs under various conditions. Load testing was conducted to observe how the game handled multiple projectiles and obstacles on screen simultaneously, ensuring that there were no significant performance degradations. We monitored the frame rate to ensure smooth gameplay and responsive controls, addressing any issues that caused frame rate drops or lag.

Usability Testing

Usability testing evaluated how easy the game was to use and how enjoyable the overall experience was for players. This feedback led to several adjustments aimed at improving usability.

Regression Testing

Regression testing ensured that new updates or fixes did not negatively impact existing functionalities. We tested all previously implemented features after making updates to verify that they continued to function correctly. This included checking that any bugs or issues previously fixed remained resolved and that no new problems had been introduced. We also ensured that the game remained compatible with different operating systems and devices after updates, addressing any new compatibility issues that arose.

8. Possible Improvements

Enhanced Graphics and Animation

a. High-Resolution Assets: upgrading to higher resolution textures and sprites could improve the visual appeal and detail of the game.
b. Advanced Animation Techniques: incorporating more sophisticated animation techniques, such as frame-by-frame animations or skeletal animations, could provide smoother and more dynamic visual effects.

Gameplay Depth

c. Additional Levels and Challenges: expanding the number of levels and introducing new types of obstacles and projectiles could add more variety and extend gameplay.
d. Difficulty Levels: implementing multiple difficulty settings would cater to a broader range of skill levels, making the game more accessible and challenging for different types of players.

User Interface Enhancements

e. Customizable Controls: allowing players to customize control schemes or use alternative input methods could improve accessibility and user satisfaction.
f. Improved HUD: enhancing the in-game heads-up display (HUD) to include more detailed statistics and dynamic feedback could provide players with better insights into their performance.

Bug Fixes and Stability:

g. Comprehensive Testing: it may be possible for the bulletproof mirror to not function always correctly. Specifically, lasers may not be reflected in the correct way, especially in level 2, where two mirrors are present. This problem may be due to an incorrect specification of the mirror class, or of the definition of how the laser trajectory is affected by the mirror. Nonetheless it functions almost perfectly, following the specifications and physical implications of the reflection phenomenon, but could ultimately be improved to perform always as expected.
h. User Feedback Integration: gathering and incorporating feedback from players to address common issues and improve overall gameplay experience.