Università degli Studi di Pavia

Bachelor degree in Artificial Intelligence

Cannon — Final exam project — A.Y. 2023/24

BOLIS ANDREA Mat.535315

Contents

The goal of the **Cannon** project is to design and implement a retro-style artillery game [1], inspired by classic games. The player controls cannon that fires projectile, with the objective of hitting a target while accounting for the physics of the world.

The game introduces a challenge through optional obstacles that can interact with the projectiles in various ways, altering their trajectories or block the shot.

The key elements of the game are:

- **Cannon**: the player controls angle and speed of projectile.
- **Target**: the player's objective is to hit the target by firing projectiles from the cannon.
- **Obstacles**: objects such as rocks, perpetio, mirrors, and wormholes are placed in the game to make hitting the target more challenging. These obstacles interact with projectiles in different ways, such as reflecting, absorbing, or redirecting them. The number of obstacle and their position will change at each level.

The project focuses on using **Kivy**, a Python framework for developing interactive applications, to create the game.

Assumption, methods, procedures

Cannon is a single-player game with a fixed-size game screen.

The player's goal is to hit the target to advance to the next level by adjusting the angle and velocity of the projectile fired from the cannon. During the game, the player must either avoid or hit the obstacles.

The game consists of 5 levels with increasing difficulty, determined by the position and number of obstacles. The game ends when the player runs out of projectiles or completes the fifth level. The ultimate goal is to finish the fifth level with the fewest shots possible. At the end of the game, the player's score determines their position on the hall of fame.

At the start of the game, the player has 12,000 points. For each level completed, 2,000 points are awarded, while each shot has a specific cost.

Available projectiles:

Bullet:

- 10 shots available
- Cost per shot: 100 points
- Follows a parabolic trajectory influenced by the mass of the projectile.

Bomb:

- 5 shots available
- Cost per shot: 400 points
- Follows a parabolic trajectory influenced by the mass of the projectile.
- Has a larger destruction radius compared to the bullet.

Laser:

- 3 shots available
- Cost per shot: 800 points
- Not affected by gravity, so it follows a straight trajectory and has a fixed speed.

Obstacles present in the game:

Rock:

- Composed of atomic elements.
- These elements can be destroyed when hit by a projectile.

Perpetio:

- Similar to Rock, but cannot be destroyed by any projectile.

Mirror:

- Reflects laser impulses.
- Cannot be destroyed by projectiles.

Wormhole:

- An indestructible obstacle consisting of two elements.
- When a projectile hits one of the two elements, its trajectory continues by exiting from the other element.

Logical structure of the solution

The game Cannon was developed following a modular and object-oriented architecture, which offers significant advantages in terms of flexibility, maintainability, and expandability. Each component of the system has specific and well-defined responsibilities, such as managing projectile movement, interacting with obstacles, and progressing through various levels. This organization ensures that each part of the game contributes consistently to the overall experience.

The user interface is divided into four main screens:

1. **Home**: acts as a navigation page, allowing users to easily access different sections of the game.
2. **Help**: offers a detailed description of the game rules and the functionality of the objects on the field.
3. **Record**: Allows players to view the general hall of fame and check their results.
4. **Game Page**: Serves as the core of the game, implemented using Kivy, and is responsible for coordinating all game elements, including the cannon, targets, obstacles, and projectiles, as well as managing the game's progression.

The **Game Page** plays a central role in managing the game dynamics. It is responsible for initializing and resetting the game, including transitioning from one level to another. Additionally, it handles user input, activating real-time updates based on the player's actions. The update cycle includes positioning the cannon, firing the projectile, and verifying interactions between projectiles, obstacles, and targets at each frame, ensuring a continuous and consistent responsiveness of the system.

The main responsibilities of the **Game Page** include:

- Initializing and resetting the game, with support for level changes.
- Managing user input.
- Coordinating transitions between game states, such as moving to the next level.
- Monitoring interactions between projectiles, obstacles, and targets, ensuring that the game's behaviour conforms to the established interaction rules.

The overall structure of the game is organized into the following main components:

Cannon

The cannon is the main tool of interaction in the game. It allows the player to aim, adjust the speed, and fire projectiles. The angle and speed are dynamically controlled by the user's input, influencing the trajectory of the projectiles. The player uses the mouse to define the firing angle, and by holding down the left mouse button, increases the projectile's speed, which is indicated by a progress bar. The projectile is fired when the left mouse button is released.

Attributes:

- **Angle**: Calculated based on the position of the pointer relative to the cannon.
- **Speed**: A dynamic value that increases with the left mouse button press, indicated by the progress bar.

Functionality:

- **Firing projectiles**: The projectile is launched with the set angle and speed when the player releases the left mouse button.

Projectiles

Projectiles are the moving objects fired from the cannon. Each projectile type has specific attributes, such as mass, radius, and behavior. The trajectory of the projectiles is determined by the angle and initial velocity defined by the gun, following a physical movement influenced by gravity.

Base class (BaseProjectile):

- Handles common projectile behaviours, such as updating position based on velocity and mass (to simulate gravity).
- Manages interactions with obstacles and targets.

Subclasses of projectiles (Bullet, Bombshell, LaserSegment):

- Each subclass customizes the projectile's attributes, such as radius and cost of the shot.

Trajectory calculation:

- The horizontal and vertical velocity of the projectile is calculated using trigonometric functions based on the angle and initial velocity.
- Gravity is simulated by reducing vertical velocity over time.
- At each frame, the trajectory and rotation of the projectile are updated following the tangent of the trajectory.

Obstacles

Obstacles add complexity and challenge to the game, interacting in various ways with projectiles. The interaction logic is handled in the `check_collision()` method of the BaseProjectile class, which detects collisions and identifies which obstacle was hit, executing the relevant methods for interacting with the projectile.

Collision

The collision detection system is essential for the interaction between projectiles, obstacles and targets. Projectiles check for collisions with obstacles or targets in each frame. The type of obstacle determines how the collision is handled:

- **Bounding box:** for rectangular objects (such as mirrors or pieces of rock), collisions are detected by checking whether the projectile's bounding box intersects with that of the obstacle.
- **Radial collision:** for circular objects, a distance check is used to check whether the projectile is within the radius of the object.

Target

The target is the player's main objective. The player must hit him with a bullet to complete the level. When the target is hit, an animation sequence is triggered and the game progresses to the next level or ends.

Attribues:

- **Impact Detection:** The `is_hit()` method detects bullet collisions and triggers the target removal animation.
- **Animation:** When hit, the target is animated to move off the screen before moving to the next level.

Game flow:

- **Initialization:** the game starts by positioning the cannon, target, and obstacles on the screen.
- **Player input:** the player adjusts the cannon's angle and speed using the mouse or keyboard.
- **Projectile firing:** the projectile is launched following a calculated trajectory.
- **Collision detection:** while the projectile moves, it checks for collisions with obstacles or targets. If a collision occurs, the appropriate interaction is triggered.
- **Level progression**: if the target is hit, the game advances to the next level.

Motivations of the design choices

The design choices of the Cannon game were aimed at balancing simplicity, performance, and extensibility—key elements to ensure a solid and flexible structure. Every decision was made with the goal of making the game easily maintainable and scalable without compromising the quality of the gaming experience.

Separation of responsibilities

A central aspect of the design was the clear separation of responsibilities between the various game components. This division led to the creation of distinct classes, each responsible for a specific task. This approach offers several advantages: firstly, it simplifies code maintenance, as each class has a well-defined and limited role.
As a result, making changes or updates is easier and does not risk breaking other parts of the system. Secondly, the game's extensibility is guaranteed; adding new elements, such as obstacles or projectiles, becomes a smooth process, as it only requires creating new classes without modifying existing code. For example, to introduce a new obstacle, it is sufficient to define a new class and add it to the game interface. Finally, this approach promotes code reuse.

Object-oriented design:

Object-oriented design (OOP) helped to organize the game more neatly. Encapsulation of data and functionality within objects like projectiles, targets, and obstacles made the system more robust, hiding the internal details of each object and allowing for cleaner interactions between components. This type of organization also simplifies managing different types of projectiles: thanks to polymorphism, they can be handled through a single interface, while maintaining specific behaviours for each type. The use of inheritance, in turn, allows sharing common functionalities among multiple objects, reducing code redundancy and improving the system's overall efficiency.

Use of constants for flexibility:

To enhance design flexibility, key game parameters were managed through constants. This approach makes it extremely easy to adjust aspects such as gravity, projectile speed, or screen dimensions, as all these variables are centralized in a dedicated file. This allows for changes in game difficulty or physics behaviour by simply modifying a few parameters, without needing to alter the actual code. Moreover, this configurability provides the possibility to easily create new game modes or difficulty levels by adjusting the parameters as needed.

Animation and visual feedback:

Animations also play an important role in the game's design. The introduction of smooth movements and clear visual feedback makes the gaming experience more engaging. When a target is hit or a projectile interacts with an obstacle, the animation provides immediate feedback, helping players better understand the outcome of their actions. This not only improves game comprehension but also contributes to creating a richer and more satisfying visual experience.

Modular design of obstacles:

Finally, the design of the obstacles was conceived to be modular. Each obstacle, whether rocks, mirrors, or wormholes, has its own class and specific behavior. This allows for the introduction of unique behaviors for each obstacle: rocks, for example, shatter into smaller pieces, mirrors reflect projectiles, and wormholes teleport them. Thanks to this approach, it is possible to add new obstacles with different interactions without redesigning the game's core mechanics, making the system highly scalable.

Implementation

The **Cannon** game is implemented using the **Kivy** framework, a Python library for developing graphical user interfaces, with an emphasis on modular and object-oriented design.

Pages class

Each page is implemented as a separate class derived from the Kivy *Screen* object. The transition between pages is managed through the *ScreenManager*, which handles switching between different screens.

```
# Main App class for managing the screens
class MyCanun(App):
    def build(self):
        # Set the window size based on constants
        Window.size = (constants.SCREEN_WIDTH, constants.SCREEN_HEIGHT)

        # Create a ScreenManager and add screens to it
        sm = ScreenManager()
        sm.add_widget(HomePage(name="home"))  # Add HomePage screen
        sm.add_widget(GamePage(name="game"))  # Add GamePage screen
        sm.add_widget(RecordPage(name="record"))  # Add RecordPage screen
        sm.add_widget(HelpPage(name="help"))  # Add HelpPage screen

        return sm  # Return the ScreenManager
```

Game Page

The game consists of several key components, each implemented as independent classes.

The upper part of the screen is dedicated to game information, while the main section features the cannon on the left, the target on the right, and obstacles positioned between them.

The **Cannon** class, derived from the Kivy Image object, offers various methods to control its actions:

- **Aiming**: by moving the mouse within the game area, the player can adjust the cannon's angle and direction.

```
    def update_angle(self, pos):  # Update the rotation angle based on the mouse/touch position
        if not self.shooting:  # Only update angle when not shooting
            x = pos[0] - self.x  # Calculate the x offset
```

```
        y = pos[1] - self.y  # Calculate the y offset
        vect = sqrt(pow(x, 2) + pow(y, 2))  # Compute the vector magnitude
        angle = degrees(acos(x / vect))  # Calculate the angle in degrees
        self.angle = angle  # Update the cannon's angle
        self.rot.angle = angle  # Set the rotation angle
        self.rot.origin = self.center  # Ensure the origin of rotation is the center
```

- **Increasing projectile speed**:

```
def inc_velocity(self, dt):  # Increment the velocity when the mouse is held down
    self.velocity += 1  # Increase velocity
    if self.velocity > 100:  # Reset if velocity exceeds 100
        self.velocity = 0
    # Update the power cannon UI with the current velocity value
    self.parent.parent.parent.power_cannon.set_value(self.velocity)
```

- **Shooting projectiles:** through this method, the relevant class is instantiated based on the selected projectile before firing the projectile.

```
def start_shot(self):  # Start shooting based on the current muzzle type
    pos = self.initPos()  # Calculate the initial position for the projectile
    # Create a projectile based on the selected muzzle type
    if self.muzzle_type == 0:
        muzzle = Bullet(angle=self.angle, velocity=self.velocity, pos=pos)
    elif self.muzzle_type == 1:
        muzzle = Bombshell(angle=self.angle, velocity=self.velocity, pos=pos)
    elif self.muzzle_type == 2:
        muzzle = Laser(size=[100, constants.LASER_SIZE], angle=self.angle, pos=pos)

    self.shooting = True  # Set shooting flag to True
    return muzzle  # Return the created projectile
```

Obstacles in the game, such as **Rock**, **Perpetio**, **Mirror**, and **WarmHole**, are implemented as independent classes. These classes are responsible for defining only the graphical aspect of the obstacles.

The **BaseProjectile** class (along with its subclasses like **Bullet**, **Bombshell**, and **Laser**) manages the behaviour of the projectiles fired by the cannon. This includes calculating the projectile's movement based on its initial speed, angle, and mass, as well as handling collisions with obstacles and targets.

- **Trajectory calculation**:

```
# Move the projectile and check for collisions
def move(self):
    stateCollision = self.check_collision(self.parent)  # Check if there's a collision
    if stateCollision > 0:
        return stateCollision

    # Calculate new position based on velocity and gravity
```

```
    x = self.velx * self.time
    y = self.vely * self.time - 0.5 * self.mass * pow(self.time, 2)

    # Update trajectory and position
    self.calcTangente(x, y, self.velx, self.vely, self.time, self.angle)
    self.pos = x + self.initialPos[0] + self.inc[0], y + self.initialPos[1] + self.inc[1]

    self.time += constants.FPS * 5  # Increment time
    # Check if projectile goes out of bounds
    if (self.x + self.size[0] > constants.SCREEN_WIDTH) or (self.y < 0):
        return 3
    return 0

# Calculate tangent angle of the trajectory for visual rotation
def calcTangente(self, oldx, oldy, velx, vely, time, _angle):
    tmpTime = time + 1
    x = velx * tmpTime
    y = vely * tmpTime - 0.5 * self.mass * pow(tmpTime, 2)
    vectx = x - oldx
    vecty = y - oldy
    vect = math.sqrt(pow(vecty, 2) + pow(vectx, 2))
    angle = math.degrees(math.acos(vectx / vect))

    # Rotate projectile based on the tangent of the trajectory
    if vecty >= 0:
        self.rot.angle = angle - 90
    else:
        self.rot.angle = -angle - 90
    self.rot.origin = self.center
```

- **Collision handler**: determine what type of object was hit and call the appropriate collision handler to perform different actions

```
# Check for collision with various objects
def check_collision(self, root):
    if self.collide_widget(root.target):
        return 1  # Collision with the target

    # Iterate over objects in the environment and check for collisions
    for obj in root.objects:
        if self.collide_widget(obj):
            if isinstance(obj, Rock):
                return self.manage_collision_rock(obj)  # Handle rock collision
            elif isinstance(obj, Mirror):
                return self.manage_collision_mirror(obj)  # Handle mirror collision
            elif isinstance(obj, Perpetio):
                return self.manage_collision_perpetio()  # Handle Perpetio collision
            elif isinstance(obj, WarmHole):
                if obj.is_in:
```

```
                return self.manage_collision_wormhole(obj)  # Handle wormhole entry

        return 0  # No collision
```

Both the methods `move()` and `check_collision()` are called every frame.

Each subclass of **BaseProjectile** can define additional behaviour:

- **Bullet**: a basic projectile with standard mass and size.
- **Bombshell**: a larger projectile with more destructive power.
- **Laser**: a segmented projectile that can reflect off mirrors.

The **Target** class is an Image class and contain the method for create animation when the target is hit, it moves off-screen through an animated sequence and then triggers the game's level progression system.

```
# Method to be called when the target is hit
def is_hit(self):
    # Define the first animation: move the target down to a position near the bottom of the screen
    animation1 = Animation(pos_hint={'y': 0.1}, duration=1)

    # Define the second animation: move the target completely off the screen
    animation2 = Animation(pos_hint={'y': -1}, duration=1)

    # Chain the animations: once the first animation is complete, start the second animation
    animation1.bind(on_complete=lambda *args: animation2.start(self))

    # After the second animation completes, update the level in the game
    animation2.bind(on_complete=lambda *args: self.parent.parent.parent.updateLevel(0))

    # Start the first animation
    animation1.start(self)
```

The **GamePage** class manages the overall state of the game, including progressing to the next level when the target is hit.

```
# Setup the objects and obstacles for the current level
def setup_level(self):
    self.clearObject()  # Clear previous level's objects
    self.objects = level.setup_level(self.level)  # Get the new objects for the current level
    for obj in self.objects:
        self.layoutGame.add_widget(obj)  # Add each object to the game layout
```

The function level.setup_level(self.level) return all the obstacle instance in the new level.

Record

At the end of the game, it is possible to save the score achieved with a nickname in the general hall of fame. This feature is managed by the function `saveResult()`, which inserts the player's name and score into the leaderboard in the correct position.

```
# Function to save a result (name and points) to the leaderboard
def saveResult(name, points):
    leaderBoard = readResult()  # Read the current leaderboard
    data = {"name": f"{name}", "score": points}  # Create a new entry with name and score
    leaderBoard.append(data)  # Add the new entry to the leaderboard
    # Sort the leaderboard in descending order of score
    leaderBoard.sort(key=lambda x: x['score'], reverse=True)
    # Write the updated leaderboard to the file
    with open(constants.PATH_FILE_RECORDS, 'w') as file:
        json.dump(leaderBoard, file, indent=4)  # Save the leaderboard to a JSON file
```

At the end, the hall of fame can be viewed in a table format within the **Record** page. This allows players to see the ranking and compare scores in an organized manner.

Tools and Resources

The **Cannon** game project was built using a variety of tools and resources, ensuring smooth development, testing, and deployment. Below is an overview of the key tools and resources used for the project.

Development Tools

Python 3.12

- **Version**: Python 3.12
- **Key Features Utilized**:
  - Object-oriented programming for structuring game components.
  - Built-in libraries for mathematical operations (e.g., `math` for trigonometry and physics simulations).
  - JSON support for saving and loading player results.

Kivy Framework

- **Version**: Kivy 2.0.0
- **Key Features Utilized**:
  - **UI Components**: The game layout (cannon, projectiles, obstacles, target) is rendered using Kivy's `Widget`, `BoxLayout`, and `FloatLayout` components.
  - **Animation Support**: Kivy's `Animation` module was used to animate target movements and provide visual feedback when a projectile hit the target.
  - **Touch Input Handling**: Kivy's touch event system was used to handle user input for aiming and firing the cannon.
  - **Screen Management**: Kivy's `ScreenManager` was used to transition between different game states (home page, game page, record page, etc.).

IDE

- **VS Code**: Visual Studio Code was the primary text editor used for writing and managing the source code. It provides integrated terminal support, syntax highlighting, and extensions for Python and Kivy.

Libraries

Kivy [2]

- **Version**: 2.0.0
- **Purpose**: Kivy served as the main library for rendering the game and managing user interactions. It provided the tools needed to manage both the graphical elements and the animations of the game.

Math Library (Python Standard Library)

- **Purpose**: Used to perform mathematical calculations necessary for projectile motion, including trigonometry functions (sine, cosine) and vector calculations.
- **Functions Used**:
  - `math.sin()` and `math.cos()` for calculating velocity components based on the cannon's firing angle.
  - `math.sqrt()` for distance calculations in collision detection.
  - `math.degrees()` and `math.radians()` for converting between angles and radians.

JSON (Python Standard Library)

- **Purpose**: The JSON library was used for saving and loading player results (name, score) in a structured format. This allowed the game to persist high scores between sessions and display the Hall of Fame in the records screen.
- **Key Functions Used**:
  - `json.load()` to read player scores from a file.
  - `json.dump()` to write updated player scores to the file.

Testing Tools

The game was manually tested by playing it repeatedly, focusing on projectile behaviour, obstacle interaction, collision detection, and overall gameplay experience.

The focus during the test was in:

- Accuracy of projectile trajectories.
- Responsiveness of user input.
- Correct behaviour of obstacles when interacting with projectiles (e.g., reflecting, breaking).

Resources

Constants

All the game parameters are written in `constant.py` this provided flexibility in tweaking game mechanics without modifying the core logic, for example**:**

- `BULLET_RADIUS`, `BOMB_MASS`, and other projectile-related constants for adjusting gameplay difficulty.
- `SCREEN_WIDTH` and `SCREEN_HEIGHT` for determining the size of the game window.

Images and Graphics

All the icon and image are stored in path `/assets/images/`, all the paths are store in constants.py.

Reference

[1] “Artillery game — wikipedia.” https://en.wikipedia.org/wiki/Artillery_game.

[2] “Kivy: Cross-platform Python framework for GUI apps development.” https://kivy.org/.

[3] Kivy tutorial – techwithtim “https://www.techwithtim.net/tutorials/python-module-walk-throughs/kivy-tutorial”

Possible Improvements

Physics Enhancements:

- **Air resistance simulation:** Currently, projectile movement only considers gravity and velocity. Adding air resistance would improve the realism of the physics simulation.
- **More detailed collision detection**: Improve collision accuracy by considering more complex shapes (such as polygons or curves) instead of just bounding boxes or radial collisions, for more precise interactions between projectiles and obstacles.

Level Design and Progression Enhancements:

- **Procedural level generation**: Instead of predefined levels, introduce procedural generation that creates new levels based on parameters like difficulty or player style.
- **Different game modes**: Add various game modes, such as a timed mode or survival mode, where the player must hit as many targets as possible before time or ammo runs out.

AI and Adaptive Difficulty:

- **Adaptive difficulty**: Implement a difficulty system that adjusts to the player's skill level, making the game easier or harder based on current performance.
- **Dynamic targets**: Instead of stationary targets, introduce targets that move or require multiple hits to destroy.