Cannon Game

Artificial Intelligence

Computer programming, algorithms and data structures

Beatrice Camera
February 25th, 2025

Summary

01 Introduction

A single-player artillery challenge built in Python with Kivy.

Players control a cannon, adjusting angle and velocity.

Three progressively challenging levels with moving targets and obstacles.

Emphasizes modular design and real-time gameplay.

GOAL: hit all targets on the screen using a maximum of 10 projectiles

02 Design Choices

Nickname

Before starting, players must enter a nickname. This name is used to track scores and display rankings in the Hall of Fame.

If no nickname is entered, the game prompts the player to input one. Once confirmed, the game transitions to the welcome screen

Welcome

After entering a nickname, players are greeted with a welcome message.

This screen introduces the game's objective and main controls. Players can proceed to the main menu, where they can start a new game, check the Hall of Fame, or view the help section

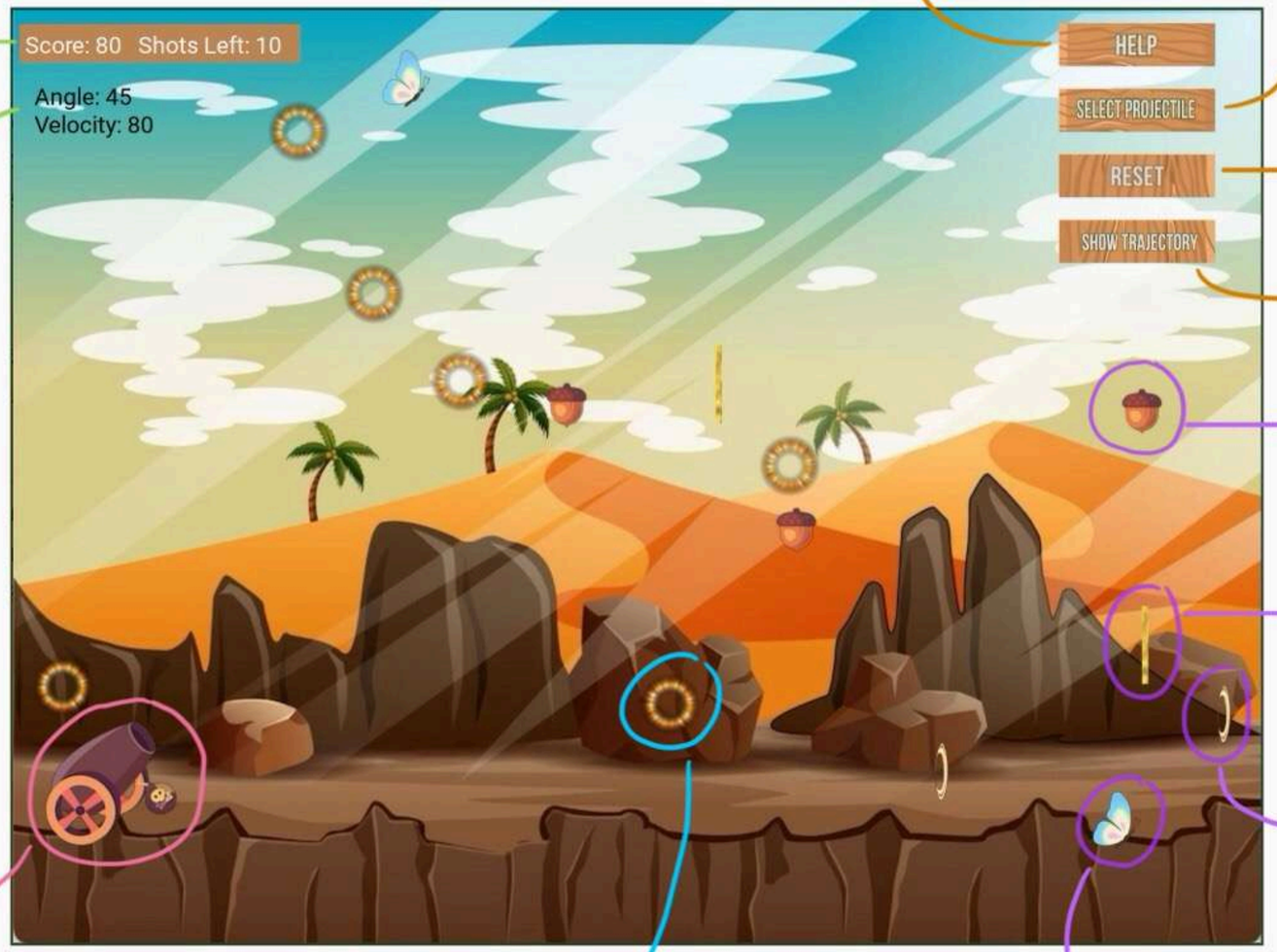
Projectile selection

Before playing, players choose a projectile type. Each projectile has different physics properties, affecting gameplay strategy:

- **Bullet**: Standard projectile, affected by gravity.
- **Bombshell**: Heavier, penetrates obstacles before exploding.
- **Laser**: Fast, moves in a straight line, unaffected by gravity.

Help Button
Opens a guide explaining game controls and objectives.

Score & Shots Left
Displays the player's current score and remaining shots



Select Projectile Button
Allows switching between bullet, bombshell, and laser

Reset Button
Restarts the level, repositioning obstacles and targets (-15 points)

Show Trajectory Button
Displays a predicted shot path for 15 seconds (-10 points)

Angle and velocity
Determines the projectile's trajectory; adjustable by the player.

Rocks
Obstacles that block projectiles and add difficulty

Bulletproof Mirror
Reflect certain projectiles, allowing for strategic bounces

Cannon
The player-controlled weapon used to fire projectiles by adjusting angle and velocity

Wormholes
Teleport projectiles to a linked location, altering their trajectory

Target
The main objective; hitting all targets advances the level

Perpetio
Indestructible obstacles that instantly destroy any projectile upon collision

Popups

When the player runs out of shots

- **Restart** : Try again with the same nickname.
- **Shutdown**: Close the game.
- **Hall of Fame** : View the leaderboard.

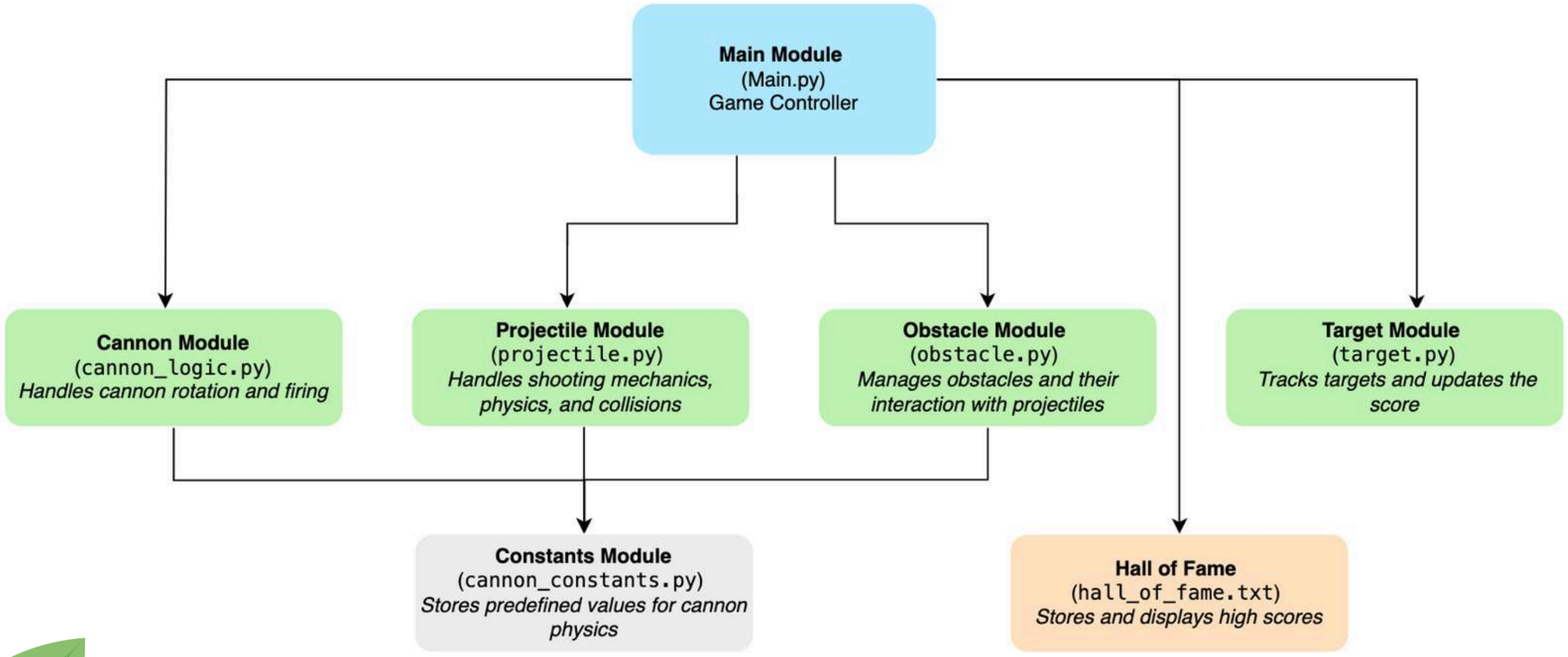
When the player hit all the targets

Displayed when all targets are successfully hit. The player can proceed to the next level,

The hall of fame

A leaderboard displaying the highest scores achieved by players

03 Code Structure and Module Organization



03 Main Module

The Main module serves as the central control unit for the game, orchestrating initialization, state management, and user interaction.

```python
class CanGame(Widget):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)

        # initialize game state variables
        self.state = "enter_nickname"
        self.nickname = ""
        self.score = 0
        self.shots_left = 10
        self.projectiles = []
        self.obstacles = []
        self.selected_projectile = "bullet"
        self.level = 1
        self.angle = 45
        self.velocity = 50
        self.cannon = None
        self.error_label = None
        self.game_over = False
        self.paused = False

        # schedule the main update loop
        Clock.schedule_interval(self.update, 1 / 120)

        # define level backgrounds
        self.lvl_bg = {
            1: "images/1_level.jpg",
            2: "images/cannon_africa.jpg",
            3: "images/cannon_america.jpg"
        }

        # initialize the default background
        with self.canvas.before:
            self.background = Rectangle(
                source="images/choice_background.png",
                pos=self.pos, size=self.size
            )
        # bind size and position changes to update the background
        self.bind(size=self.update_background, pos=self.update_background)
```

Main Module

This method initializes the main game variables:

- player name,
- score,
- available projectiles,
- level,
- cannon angle,
- velocity.

It also prepares the game state and schedules the primary update loop to run at 120 FPS using Kivy's Clock.schedule_interval

```python
class CanGame(Widget):
    def __init__(self, **kwargs):
        # create and add the main layout
        self.layout = FloatLayout(size=self.size)
        self.bind(size=self.resize)
        self.add_widget(self.layout)

        # create the nickname input widget
        self.nickname_input = TextInput(
            hint_text="Enter your nickname",
            size_hint=(None, None),
            size=(400, 50),
            pos_hint={"center_x": 0.5, "center_y": 0.6},
            multiline=False
        )
        # create the continue button for nickname entry
        self.bgbutton = Button(
            background_normal="images/buttons/immagine_continue.jpg",
            size_hint=(None, None),
            size=(230, 188),
            pos_hint={"center_x": 0.5, "center_y": 0.4}
        )
        self.bgbutton.bind(on_press=self.start)

        # add widgets to the layout
        self.layout.add_widget(self.nickname_input)
        self.layout.add_widget(self.bgbutton)

        # Debug
        self.deb_wid("Nickname Input", self.nickname_input)
        self.deb_wid("background Button", self.bgbutton)

        if self.cannon:
            print(f"Cannon initialized at position: {self.cannon.position}")
        else:
            print("Cannon not initialized yet.")
```

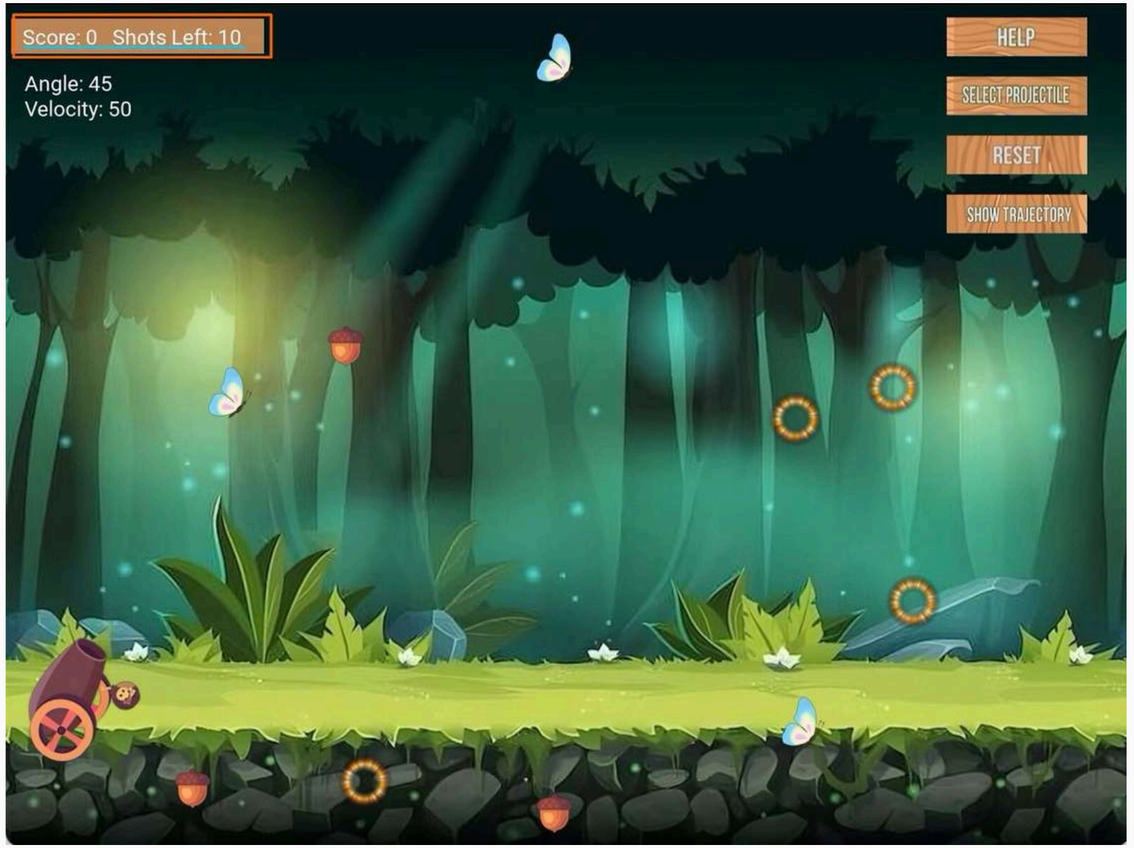
Main Module

Before starting the game, the player needs to enter a nickname. This section initializes the main interface and handles user input

- Creates the game layout
- Adds a nickname input field
- Adds a "Continue" button to start the game
- Debugs widget initialization
- Checks cannon initialization

Main Module

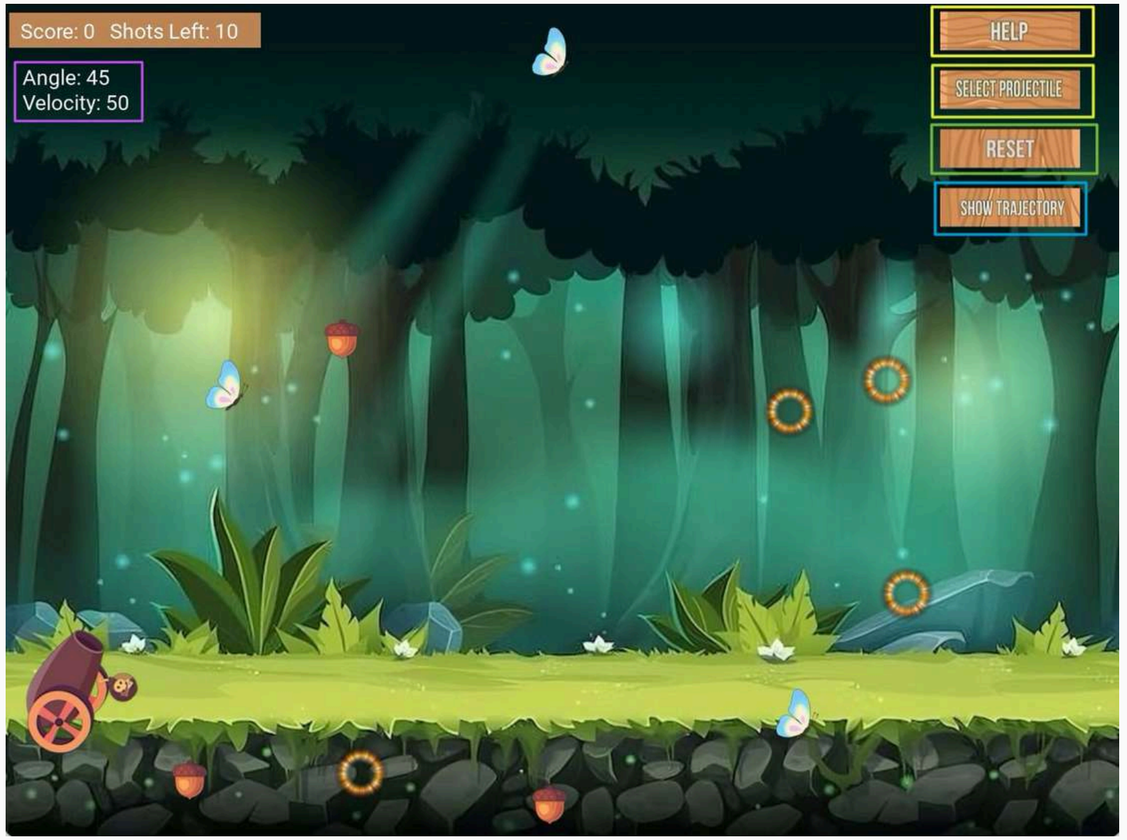
```
class CanGame(Widget):
    def init_game(self, instance=None):
        # initialize game level; update background and state accordingly
        print(f"backgrounding level {self.level}!")
        self.state = f"level_{self.level}"

        if hasattr(self, 'background'):
            self.background.source = self.lvl_bg.get(self.level, "")
        else:
            with self.canvas.before:
                self.background = Rectangle(
                    source=self.lvl_bg.get(self.level, ""),
                    pos=self.pos,
                    size=self.size
                )
            self.bind(size=self.update_background, pos=self.update_background)

        self.layout.clear_widgets()

        # initialize the cannon if not present
        if not self.cannon:
            self.cannon = Cannon(position=[100, 150], angle=45)
            self.add_widget(self.cannon)
            print(f"Cannon initialized at position: {self.cannon.position}")

        self.cannon.position = [100, 190]
        self.cannon.angle = 45

        # generate obstacles for the current level
        self.initialize_obstacles()

        # draw the score background and label
        with self.canvas.before:
            Color(0.74, 0.53, 0.33, 1)
            self.score_background = Rectangle(
                size=(360, 50),
                pos=(self.width * 0.005, self.height * 0.94)
            )

        self.score_text = f"Score: {self.score}   Shots Left: {self.shots_left}"
        self.score_label = Label(
            text=self.score_text,
            size_hint=(None, None),
            size=(200, 50),
            pos_hint={"x": 0.05, "top": 0.98},
            halign="left",
            valign="middle",
            color=(1, 1, 1, 1)
        )
        self.layout.add_widget(self.score_label)
```



Level 1

Main Module

```
# display a label for current angle and velocity (color changes based on level)
if self.level == 1:
    param_color = (1, 1, 1, 1)
else:
    param_color = (0, 0, 0, 1)
self.param_label = Label(
    text=f"Angle: {self.cannon.get_angle()}\nVelocity: {self.velocity}",
    size_hint=(None, None),
    size=(200, 50),
    pos_hint={"center_x": 0.065, "top": 0.91},
    color=param_color,
)
self.layout.add_widget(self.param_label)

# create and add pause and help buttons
self.pause_button = Button(…
self.pause_button.bind(on_press=self.toggle_pause)
self.layout.add_widget(self.pause_button)

self.help_button = Button(…
self.help_button.bind(on_press=self.helpscreenshow)
self.layout.add_widget(self.help_button)

# create reset and trajectory preview buttons
self.reset_button = Button(…
self.reset_button.bind(on_press=self.reset_level)
self.layout.add_widget(self.reset_button)

self.trajectory_button = Button(…
self.trajectory_button.bind(on_press=lambda instance: self.show_trajectory())
self.layout.add_widget(self.trajectory_button)


# store the base score at level start for resets
self.level_base_score = self.score

Clock.schedule_interval(self.update, 1 / 120)
```



Level 1

Main Module

```
# display a label for current angle and velocity (color changes based on level)
if self.level == 1:
    param_color = (1, 1, 1, 1)
else:
    param_color = (0, 0, 0, 1)
self.param_label = Label(
    text=f"Angle: {self.cannon.get_angle()}\nVelocity: {self.velocity}",
    size_hint=(None, None),
    size=(200, 50),
    pos_hint={"center_x": 0.065, "top": 0.91},
    color=param_color,
)
self.layout.add_widget(self.param_label)

# create and add pause and help buttons
self.pause_button = Button(…
self.pause_button.bind(on_press=self.toggle_pause)
self.layout.add_widget(self.pause_button)

self.help_button = Button(…
self.help_button.bind(on_press=self.helpscreenshow)
self.layout.add_widget(self.help_button)

# create reset and trajectory preview buttons
self.reset_button = Button(…
self.reset_button.bind(on_press=self.reset_level)
self.layout.add_widget(self.reset_button)

self.trajectory_button = Button(…
self.trajectory_button.bind(on_press=lambda instance: self.show_trajectory())
self.layout.add_widget(self.trajectory_button)


# store the base score at level start for resets
self.level_base_score = self.score

Clock.schedule_interval(self.update, 1 / 120)
```

```
def toggle_pause(self, instance):
    # toggle the pause state of the game
    if self.paused:
        self.resume_game()
    else:
        self.paused = True
        Clock.unschedule(self.update)
        self.pause_overlay = FloatLayout(size=self.size)
        with self.pause_overlay.canvas.before:
            Color(0, 0, 0, 0.6)  # Black with 60% opacity
            self.dark_rect = Rectangle(size=self.pause_overlay.size, pos=self.pause_overlay.pos)

        def update_rect(instance, value):
            self.dark_rect.size = instance.size
            self.dark_rect.pos = instance.pos
        self.pause_overlay.bind(size=update_rect, pos=update_rect)

        # create projectile selection buttons
        bullet_button = Button(…
        bullet_button.bind(on_press=lambda x: self.sel_proj("bullet"))

        bombshell_button = Button(…
        bombshell_button.bind(on_press=lambda x: self.sel_proj("bombshell"))

        laser_button = Button(…
        laser_button.bind(on_press=lambda x: self.sel_proj("laser"))

        # Create a resume button that simply resumes the game without changing the projectile.
        resume_button = Button(…
        resume_button.bind(on_press=lambda instance: self.resume_game())

        self.pause_overlay.add_widget(bullet_button)
        self.pause_overlay.add_widget(bombshell_button)
        self.pause_overlay.add_widget(laser_button)
        self.pause_overlay.add_widget(resume_button)

        self.add_widget(self.pause_overlay)
        print("Game paused. Shots left:", self.shots_left)
```

Pause Button

Toggles the game’s pause state. If paused, it displays a semi-transparent overlay with options to change the projectile or resume. If resumed, it removes the overlay and restarts the update loop.

Main Module

```
# display a label for current angle and velocity (color changes based on level)
if self.level == 1:
    param_color = (1, 1, 1, 1)
else:
    param_color = (0, 0, 0, 1)
self.param_label = Label(
    text=f"Angle: {self.cannon.get_angle()}\nVelocity: {self.velocity}",
    size_hint=(None, None),
    size=(200, 50),
    pos_hint={"center_x": 0.065, "top": 0.91},
    color=param_color,
)
self.layout.add_widget(self.param_label)

# create and add pause and help buttons
self.pause_button = Button(…
self.pause_button.bind(on_press=self.toggle_pause)
self.layout.add_widget(self.pause_button)

self.help_button = Button(…
self.help_button.bind(on_press=self.helpscreenshow)
self.layout.add_widget(self.help_button)

# create reset and trajectory preview buttons
self.reset_button = Button(…
self.reset_button.bind(on_press=self.reset_level)
self.layout.add_widget(self.reset_button)

self.trajectory_button = Button(…
self.trajectory_button.bind(on_press=lambda instance: self.show_trajectory())
self.layout.add_widget(self.trajectory_button)


# store the base score at level start for resets
self.level_base_score = self.score

Clock.schedule_interval(self.update, 1 / 120)
```

```
def helpscreenshow(self, instance):
    # display the help screen popup
    background = Image(source="images/help.jpeg", size_hint=(1, 1), allow_stretch=True, keep_ratio=True)
    help_popup = Popup(
        title="",
        content=background,
        size_hint=(0.8, 0.8),
        auto_dismiss=True,
        separator_height=0
    )
    help_popup.open()
```

Help Button

Opens the help screen popup, displaying game instructions and mechanics to assist the player

Main Module

```
# display a label for current angle and velocity (color changes based on level)
if self.level == 1:
    param_color = (1, 1, 1, 1)
else:
    param_color = (0, 0, 0, 1)
self.param_label = Label(
    text=f"Angle: {self.cannon.get_angle()}\nVelocity: {self.velocity}",
    size_hint=(None, None),
    size=(200, 50),
    pos_hint={"center_x": 0.065, "top": 0.91},
    color=param_color,
)
self.layout.add_widget(self.param_label)

# create and add pause and help buttons
self.pause_button = Button(…
self.pause_button.bind(on_press=self.toggle_pause)
self.layout.add_widget(self.pause_button)

self.help_button = Button(…
self.help_button.bind(on_press=self.helpscreenshow)
self.layout.add_widget(self.help_button)

# create reset and trajectory preview buttons
self.reset_button = Button(…
self.reset_button.bind(on_press=self.reset_level)
self.layout.add_widget(self.reset_button)

self.trajectory_button = Button(…
self.trajectory_button.bind(on_press=lambda instance: self.show_trajectory())
self.layout.add_widget(self.trajectory_button)


# store the base score at level start for resets
self.level_base_score = self.score

Clock.schedule_interval(self.update, 1 / 120)
```

```
def reset_level(self, instance):
    # Reset positions for obstacles that are not "rock" or "target"
    for obstacle in self.obstacles:
        if obstacle.obstacle_type not in ["rock", "target"] and hasattr(obstacle, 'initial_pos'):
            obstacle.pos = obstacle.initial_pos[:]
            obstacle.position = obstacle.initial_pos[:]

    # remove and recreate obstacles of type "rock" (non-scoring obstacles)
    rocks_to_remove = [obs for obs in self.obstacles if obs.obstacle_type == "rock"]
    for rock in rocks_to_remove:
        if rock in self.obstacles:
            self.obstacles.remove(rock)
        if rock.parent:
            rock.parent.remove_widget(rock)
    if hasattr(self, 'initial_rock_data'):
        for pos, size in self.initial_rock_data:
            new_rock = Obstacle("rock", self, image="images/small_images/immagineghianda.png", pos=pos, size=size)
            new_rock.initial_pos = pos[:]
            new_rock.position = pos[:]
            new_rock.pos = (pos[0] - size[0] // 2, pos[1] - size[1] // 2)
            self.obstacles.append(new_rock)
            self.layout.add_widget(new_rock)
            if new_rock.parent is None:
                self.layout.add_widget(new_rock)

    # remove and recreate obstacles of type "target" (scoring targets)
    targets_to_remove = [obs for obs in self.obstacles if obs.obstacle_type == "target"]
    for target in targets_to_remove:…
    if hasattr(self, 'initial_target_data'):…

    # reset projectiles and adjust score based on the level's base score minus penalty
    self.shots_left = 10
    self.score = self.level_base_score - 15
    self.level_base_score = self.score
    self.score_label.text = f"Score: {self.score}   Shots Left: {self.shots_left}"

    # Display a temporary penalty label
    penalty_color = (0, 0, 0, 1) if self.level > 1 else (1, 1, 1, 1)
    penalty_label = Label(
        text="As a penalty, 15 points have been removed.",
        size_hint=(None, None),
        size=(400, 50),
        pos_hint={"center_x": 0.5, "top": 0.85},
        color=penalty_color,
        font_size='20sp'
    )
    self.layout.add_widget(penalty_label)
    Clock.schedule_once(lambda dt: self.layout.remove_widget(penalty_label), 3)
```

Reset Button

Resets the current level by restoring obstacles and targets to their initial positions. It also resets the number of shots left and applies a score penalty.

Main Module

```
# display a label for current angle and velocity (color changes based on level)
if self.level == 1:
    param_color = (1, 1, 1, 1)
else:
    param_color = (0, 0, 0, 1)
self.param_label = Label(
    text=f"Angle: {self.cannon.get_angle()}\nVelocity: {self.velocity}",
    size_hint=(None, None),
    size=(200, 50),
    pos_hint={"center_x": 0.065, "top": 0.91},
    color=param_color,
)
self.layout.add_widget(self.param_label)

# create and add pause and help buttons
self.pause_button = Button(…
self.pause_button.bind(on_press=self.toggle_pause)
self.layout.add_widget(self.pause_button)

self.help_button = Button(…
self.help_button.bind(on_press=self.helpscreenshow)
self.layout.add_widget(self.help_button)

# create reset and trajectory preview buttons
self.reset_button = Button(…
self.reset_button.bind(on_press=self.reset_level)
self.layout.add_widget(self.reset_button)

self.trajectory_button = Button(…
self.trajectory_button.bind(on_press=lambda instance: self.show_trajectory())
self.layout.add_widget(self.trajectory_button)


# store the base score at level start for resets
self.level_base_score = self.score

Clock.schedule_interval(self.update, 1 / 120)
```

```
def show_trajectory(self):
    # if a trajectory is already being shown, do nothing
    if hasattr(self, 'traj_event') and self.traj_event:…

    print("Showing trajectory for level", self.level)

    # deduct penalty points and update the score label
    self.score -= 10
    self.score_label.text = f"Score: {self.score}   Shots Left: {self.shots_left}"

    # display a temporary penalty label
    penalty_color = (0, 0, 0, 1) if self.level >= 2 else (1, 1, 1, 1)
    penalty_label = Label(…
    self.layout.add_widget(penalty_label)
    Clock.schedule_once(lambda dt: self.layout.remove_widget(penalty_label), 3)

    # create a countdown label and add it to the layout
    countdown_label = Label(…
    self.layout.add_widget(countdown_label)

    # compute initial trajectory parameters based on the cannon's tip and current angle
    start_pos = self.cannon.get_tip_position()
    angle_rad = math.radians(self.cannon.get_angle())
    if self.selected_projectile == "bombshell":
        multiplier = 2
    elif self.selected_projectile == "laser":
        multiplier = 1
    else:  # bullet
        multiplier = 5
    v0 = self.velocity * multiplier
    v0x = v0 * math.cos(angle_rad)
    v0y = v0 * math.sin(angle_rad)

    # calculate t_max depending on the projectile type
    if self.selected_projectile == "bombshell":
        t_natural = 0.0
        cumulative = 0.0
        prev_point = start_pos
        step = 0.1
        while cumulative < const.BOMB_DRILL and t_natural < 10.0:
            t_natural += step
            x = start_pos[0] + v0x * t_natural
            y = start_pos[1] + v0y * t_natural - 0.5 * 9.8 * t_natural * t_natural
            current_point = (x, y)
            cumulative += math.dist(prev_point, current_point)
            prev_point = current_point
        t_max = t_natural
    elif self.selected_projectile == "laser":
        t_max = const.LASER_DIST / const.LASER_VEL
    else:  # bullet
        t_max = 100000 / abs(v0x) if abs(v0x) > 0 else 15.0

    print("Computed t_max:", t_max)
    print("Cannon tip position:", start_pos)

    # set the countdown time
    self.trajectory_time = 15.0
    countdown_label.text = str(int(self.trajectory_time))
```

Show Trajectory Button

Main Module

```
# display a label for current angle and velocity (color changes based on level)
if self.level == 1:
    param_color = (1, 1, 1, 1)
else:
    param_color = (0, 0, 0, 1)
self.param_label = Label(
    text=f"Angle: {self.cannon.get_angle()}\nVelocity: {self.velocity}",
    size_hint=(None, None),
    size=(200, 50),
    pos_hint={"center_x": 0.065, "top": 0.91},
    color=param_color,
)
self.layout.add_widget(self.param_label)

# create and add pause and help buttons
self.pause_button = Button(…
self.pause_button.bind(on_press=self.toggle_pause)
self.layout.add_widget(self.pause_button)

self.help_button = Button(…
self.help_button.bind(on_press=self.helpscreenshow)
self.layout.add_widget(self.help_button)

# create reset and trajectory preview buttons
self.reset_button = Button(…
self.reset_button.bind(on_press=self.reset_level)
self.layout.add_widget(self.reset_button)

self.trajectory_button = Button(…
self.trajectory_button.bind(on_press=lambda instance: self.show_trajectory())
self.layout.add_widget(self.trajectory_button)


# store the base score at level start for resets
self.level_base_score = self.score

Clock.schedule_interval(self.update, 1 / 120)
```

```
# ensure the trajectory widget covers the entire layout
if not hasattr(self, 'traj_widget'):
    self.traj_widget = Widget(size=self.layout.size, pos=self.layout.pos)
else:
    self.traj_widget.canvas.clear()
    self.traj_widget.size = self.layout.size
    self.traj_widget.pos = self.layout.pos

# remove and re-add the trajectory widget so it sits on top
if self.traj_widget.parent:
    self.traj_widget.parent.remove_widget(self.traj_widget)
self.layout.add_widget(self.traj_widget)

self.traj_segments = []

# define the function to update the trajectory preview
def update_trajectory(dt):
    self.traj_widget.canvas.clear()
    self.traj_segments = []
    current_start = self.cannon.get_tip_position()
    current_angle = math.radians(self.cannon.get_angle())
    if self.selected_projectile == "bombshell":
        mult = 2
    elif self.selected_projectile == "laser":
        mult = 1
    else:
        mult = 5
    current_v0 = self.velocity * mult
    current_v0x = current_v0 * math.cos(current_angle)
    current_v0y = current_v0 * math.sin(current_angle)
    t = 0.0
    step = 0.1
    traj_points = []
    while t <= t_max:
        if self.selected_projectile != "laser":
            x = current_start[0] + current_v0x * t
            y = current_start[1] + current_v0y * t - 0.5 * 9.8 * t * t
        else:
            x = current_start[0] + current_v0x * t
            y = current_start[1] + current_v0y * t
        traj_points.append((x, y))
        t += step
    print("Trajectory points count:", len(traj_points))
    dash_length_points = 5
    gap_length_points = 3
    i = 0
    line_color = (1, 1, 1, 1)
    with self.traj_widget.canvas:
        Color(*line_color)
        while i < len(traj_points) - 1:
            seg_points = []
            for p in traj_points[i:min(i + dash_length_points, len(traj_points))]:
                seg_points.extend(p)
            seg_line = Line(points=seg_points, width=1.5)
            self.traj_segments.append(seg_line)
            i += dash_length_points + gap_length_points
    return True
```

Show Trajectory Button

```python
def sel_proj(self, projectile_type):
    # set the selected projectile type and initialize or resume the game accordingly
    self.selected_projectile = projectile_type
    print("Projectile selected:", projectile_type, "in state:", self.state)
    if self.state.startswith("level_") and self.paused:
        self.resume_game()
    elif self.state == "choose_projectile":
        self.layout.clear_widgets()
        self.init_game()
```

```python
def shoot_projectile(self):
    # fire a projectile from the cannon if shots remain
    if self.shots_left <= 0:
        print("No shots left! Game over.")
        self.finished()
        return
    tip_position = self.cannon.get_tip_position()
    print(f"Launching projectile from {tip_position}")
    projectile = Projectile(
        projectile_type=self.selected_projectile,
        start_position=tip_position
    )
    projectile.launch(
        angle=self.cannon.get_angle(),
        power=self.velocity
    )
    self.projectiles.append(projectile)
    self.add_widget(projectile)
    self.shots_left -= 1
    print(f"Shots left: {self.shots_left}")
    if self.shots_left == 0:
        print("No shots remaining!")
```

Main Module

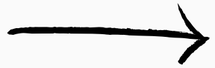
This method updates the selected projectile and ensures that the game starts or resumes with the chosen type

Once the projectile is selected, the player can shoot it by setting the angle and velocity.

- Retrieves the cannon's tip position
- Computes angle and velocity before launch
- Creates and launches the projectile
- Reduces the remaining shots count
- Handles special cases (no shots left, game over)

Projectile Module

```python
def shoot_projectile(self):
    # fire a projectile from the cannon if shots remain
    if self.shots_left <= 0:
        print("No shots left! Game over.")
        self.finished()
        return
    tip_position = self.cannon.get_tip_position()
    print(f"Launching projectile from {tip_position}")
    projectile = Projectile(
        projectile_type=self.selected_projectile,
        start_position=tip_position
    )
    projectile.launch(
        angle=self.cannon.get_angle(),
        power=self.velocity
    )
    self.projectiles.append(projectile)
    self.add_widget(projectile)
    self.shots_left -= 1
    print(f"Shots left: {self.shots_left}")
    if self.shots_left == 0:
        print("No shots remaining!")
```



```python
def launch(self, angle, power):
    radian_angle = math.radians(angle)
    if self.projectile_type == "laser":
        # for lasers, ignore 'power' and use a fixed speed
        self.velocity = [
            const.LASER_VEL * math.cos(radian_angle),
            const.LASER_VEL * math.sin(radian_angle)
        ]
    elif self.projectile_type == "bombshell":
        # bombshells use a lower multiplier to simulate greater mass
        self.velocity = [
            power * math.cos(radian_angle) * 2,
            power * math.sin(radian_angle) * 2
        ]
    else:
        # bullets use the standard multiplier
        self.velocity = [
            power * math.cos(radian_angle) * 5,
            power * math.sin(radian_angle) * 5
        ]
    print(f"Projectile launched with velocity: {self.velocity}")
```

Main Module

```
def initialize_obstacles(self):
    # generate obstacles for the current level based on level-specific configuration
    self.obstacles = []
    self.projectiles = []

    if self.level == 1:
        num_targets = 4
        num_rocks = 3
        num_perpetio = 3
        num_mirrors = 0
        num_wormholes = 0
    elif self.level == 2:
    elif self.level == 3:
    else:

    screen_padding = 50

    def get_random_position():

    def get_valid_target_position(self, min_distance=200, min_separation=150):

    # create targets
    for _ in range(num_targets):
        target = Target(self, image="images/small_images/cursor_image.png",
                        pos=get_valid_target_position(self), size=(80, 80), movable=True)
        if target.parent is None:
            self.layout.add_widget(target)
        self.obstacles.append(target)

    # create wormholes in pairs
    for _ in range(num_wormholes // 2):
        wormhole1 = Obstacle("wormhole", self, image="images/small_images/immagine_wormhole.png",
                             pos=get_random_position(), size=(50, 100))
        wormhole1.initial_pos = wormhole1.position[:]
        wormhole2 = Obstacle("wormhole", self, image="images/small_images/immagine_wormhole.png",
                             pos=get_random_position(), size=(50, 100))
        wormhole2.initial_pos = wormhole2.position[:]
        self.obstacles.extend([wormhole1, wormhole2])

    # create mirrors
    for _ in range(num_mirrors):

    # create perpetio
    for _ in range(num_perpetio):

    # create rocks
    for _ in range(num_rocks):

    # add every generated obstacle to the layout
    for obstacle in self.obstacles:
        if obstacle.parent is None:
            self.layout.add_widget(obstacle)

    # save initial rocks and target data for resets
    self.initial_rock_data = [(rock.initial_pos, rock.size)
                              for rock in self.obstacles if rock.obstacle_type == "rock"]
    self.initial_target_data = [(obs.initial_pos, obs.size)
                                for obs in self.obstacles if obs.obstacle_type == "target"]
```

Level 3

Main Module and Obstacles Module

```python
def handle_collisions(self):
    # process collisions between projectiles and obstacles
    for obstacle in self.obstacles[:]:
        for projectile in self.projectiles[:]:
            if obstacle.collision(projectile):
                # wormhole logic: teleport the projectile using the paired wormhole
                if obstacle.obstacle_type == "wormhole" and not projectile.just_teleported:…

                # mirror logic: reflect the projectile
                elif obstacle.obstacle_type == "mirror":…

                # perpetio logic: destroy the projectile
                elif obstacle.obstacle_type == "perpetio":…

                # on-hit logic for obstacles
                elif obstacle.on_hit(projectile):…

    # if no targets obstacles remain, trigger the congratulations popup
    if not any(o.obstacle_type == "target" for o in self.obstacles):…
```

```python
def collision(self, projectile):
    # Determine if a projectile collides with this obstacle. Uses an effective radius (larger for rocks) plus the projectile's radius.
    proj_x, proj_y = projectile.get_position()
    effective_radius = self.radius
    distance = math.sqrt((proj_x - self.position[0]) ** 2 + (proj_y - self.position[1]) ** 2)
    return distance < effective_radius + projectile.get_radius()
```

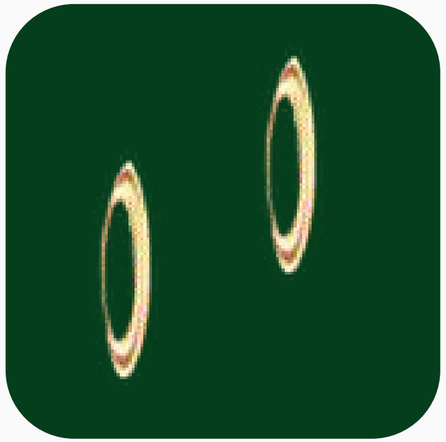
The first method hecks for projectile-obstacle collisions and applies effects:

- teleportation (wormholes),
- reflection (mirrors),
- destruction (perpetios),

or standard hit logic. Triggers a win popup if all targets are destroyed.

The second method calculates the Euclidean distance between a projectile and an obstacle to detect collisions based on their radii.

Main Module



```python
def handle_collisions(self):
    # process collisions between projectiles and obstacles
    for obstacle in self.obstacles[:]:
        for projectile in self.projectiles[:]:
            if obstacle.collision(projectile):
                # wormhole logic: teleport the projectile using the paired wormhole
                if obstacle.obstacle_type == "wormhole" and not projectile.just_teleported:
                    paired_wormhole = next(
                        (o for o in self.obstacles if o.obstacle_type == "wormhole" and o != obstacle), None
                    )
                    if paired_wormhole:
                        print(f"Paired wormhole found: {paired_wormhole}")
                        offset = paired_wormhole.radius + 10
                        exit_position = [
                            paired_wormhole.position[0] + offset,
                            paired_wormhole.position[1] + offset
                        ]
                        projectile.position = exit_position
                        projectile.image_widget.pos = (
                            exit_position[0] - projectile.radius,
                            exit_position[1] - projectile.radius
                        )
                        projectile.just_teleported = True
                        projectile.teleport_cooldown = 0.5
                        print(f"Exit position calculated: {exit_position}")
                        print(f"Projectile velocity after teleport: {projectile.velocity}")
```

Wormholes

What does it do?

- Find the twin wormhole and teleport the projectile there.
- The projectile maintains its speed and direction.
- Avoid teleporting the projectile twice in a row with a cooldown

Main and Obstacles Module

```
# mirror logic: reflect the projectile
elif obstacle.obstacle_type == "mirror":
    obstacle.projectile_reflection(projectile)
    if projectile.projectile_type in ["bullet", "bombshell"]:
        self.remove_projectile(projectile)
```

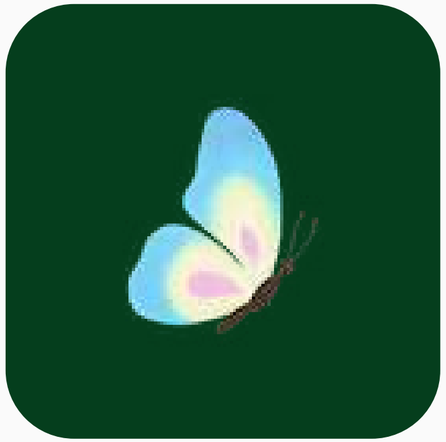
```
def projectile_reflection(self, projectile):
    # Reflect a projectile when it hits a mirror. Bullets and bombshells are removed upon impact; lasers are reflected by inverting their velocity.
    if self.obstacle_type == "mirror":
        if projectile.projectile_type in ["bullet", "bombshell"]:
            if hasattr(projectile, 'image_widget') and projectile.image_widget.parent:
                projectile.image_widget.parent.remove_widget(projectile.image_widget)
            projectile.active = False  # Mark as inactive to remove from updates
            print(f"{projectile.projectile_type.capitalize()} disappeared upon hitting the mirror!")
        elif projectile.projectile_type == "laser":
            projectile.velocity[0] = -projectile.velocity[0]
            projectile.velocity[1] = -projectile.velocity[1]
            print("Laser reflected by the mirror!")
```

Bulletproof Mirror

What does it do?

- Handles projectile interactions with mirrors.
- Lasers are reflected, changing direction.
- Bullets and bombshells are destroyed on impact.

Main Module



```
# perpetio logic: destroy the projectile
elif obstacle.obstacle_type == "perpetio":
    print(f"{projectile.projectile_type} destroyed upon hitting perpetio.")
    if projectile in self.projectiles:
        projectile.remove_widget(projectile.image_widget)
        self.projectiles.remove(projectile)
```

Perpetio

What does it do?

- Any projectile that hits it is destroyed immediately.
- Does not affect the score.

Main Module

```
                else:
                    if obstacle.obstacle_type == "target":
                        self.score += 10
                    if hasattr(obstacle, 'image_widget'):
                        obstacle.remove_widget(obstacle.image_widget)
                    if hasattr(projectile, 'image_widget'):
                        projectile.remove_widget(projectile.image_widget)
                    if obstacle in self.obstacles:
                        self.obstacles.remove(obstacle)
                    if projectile in self.projectiles:
                        self.projectiles.remove(projectile)

        # if no targets obstacles remain, trigger the congratulations popup
        if not any(o.obstacle_type == "target" for o in self.obstacles):
            print("Congratulations! All targets destroyed.")
            if self.state != "congratulations":
                self.congrat_sc()
```

Target

What does it do?

- Destroys targets when hit by a projectile.
- Increases score when a target is eliminated.
- If the projectile is a bombshell, it can destroy multiple obstacles
- When all the targets are eliminated it calls the congratulation popup.

```python
def show_hall_of_fame(self, instance):
    # display the Hall of Fame popup with sorted entries
    try:
        with open("hall_of_fame.txt", "r") as f:
            lines = f.readlines()
    except FileNotFoundError:
        lines = ["No Hall of Fame data found."]
    if lines and "No Hall of Fame data found." not in lines[0]:
        entries = []
        for line in lines:
            line = line.strip()
            parts = line.split(',')
            if len(parts) >= 2:
                try:
                    score = int(parts[1].split(':')[1].strip())
                except (IndexError, ValueError):
                    score = 0
                entries.append((line, score))
            else:
                entries.append((line, 0))
        # sort the entries by score descending
        entries.sort(key=lambda x: x[1], reverse=True)
        sorted_hof_data = "\n".join(entry[0] for entry in entries)
    else:
        sorted_hof_data = lines[0] if lines else "No Hall of Fame data found."
    sorted_hof_data = "\n\n" + sorted_hof_data

    # create a label with white text
    popup_content = Label(…
    popup_content.text_size = (620, None)
    popup_content.bind(texture_size=popup_content.setter('size'))
    scroll_view = ScrollView(size_hint=(1, 1))
    scroll_view.add_widget(popup_content)
    popup = Popup(…
    popup.open()

def save_to_hall_of_fame(self):
    # save the current player's entry to the Hall of Fame if not already present
    entry = f"Nickname: {self.nickname}, Score: {self.score}, Level: {self.level}"
    try:
        with open("hall_of_fame.txt", "r") as f:
            lines = [line.strip() for line in f.readlines()]
    except FileNotFoundError:
        lines = []

    if entry in lines:
        print("Duplicate entry found. Not saving to Hall of Fame.")
        return
    try:
        with open("hall_of_fame.txt", "a") as f:
            f.write(entry + "\n")
        print("Player's score saved to Hall of Fame.")
    except IOError as e:
        print(f"Failed to save to Hall of Fame: {e}")
```

Main Module

These two methods manage the Hall of Fame feature, storing and displaying the highest scores achieved by players

Show Hall of Fame:

- Reads from hall_of_fame.txt to get past scores.
- Handles missing files, displaying a default message if no data is found.
- Extracts player scores, sorting them in descending order.
- Formats the data and displays it in a scrollable popup.

Save Player Score:

- Creates an entry using the player's nickname, score, and level.
- Checks for duplicates to avoid saving the same score multiple times.
- Writes the new score to hall_of_fame.txt if it's unique.
- Handles errors.

This system ensures that high scores are stored and presented neatly, adding a competitive element to the game!

04 Difficulties and solutions

During development, I encountered several technical challenges that required debugging and modifications to the code. To resolve these challenges, I introduced modifications to the game logic and debugging techniques.

Challenges faced

Projectile Persistence Bug

Some projectiles were visually removed but remained active in the game logic, affecting collisions

Solution

In remove_projectile, I now remove the projectile both from the UI (parent.remove_widget) and from the projectiles list, ensuring it’s truly gone.

Trajectory Preview Missing

The dashed trajectory line did not render in later levels due to widget layering issues.

Solution

In show_trajectory, I bring the traj_widget to the top of the layout and clear it at each level initialization, avoiding layering conflicts.

Challenges faced

Premature Projectile Removal

Projectiles were disappearing too soon due to boundary-checking logic

Solution

In the Projectile.update method, I increased extended_margin to 1000, ensuring projectiles remain until they’re truly off-screen.

Game Over Triggered Too Early

The game over state was activated before verifying if the last shot hit a target

Solution

In update, I schedule a delayed check_last_projectile(dt) call and only trigger game over if the last shot truly misses all targets

Thank you for the attention!

Beatrice Camera

Bachelor of Science in Artificial Intelligence