Introduction to Computer Science Lab II
Final Report Group 28

1st Yi-Jui, Huang, 112201529
Department of Mathematics
National Central University
Taoyuan City, Taiwan (R.O.C.)

2nd Guan-Sheng, Liu, 112201538
Department of Mathematics
National Central University
Taoyuan City, Taiwan (R.O.C.)

2nd Guan-Sheng, Liu, 112201538
Department of Mathematics
National Central University
Taoyuan City, Taiwan (R.O.C.)

June 12, 2025

Abstract

This document is an introduction to "Space Ball", a fast-paced, physics-driven game. Navigate through intricate obstacle courses, collect coins, grapple through the air, shrink or grow in size, and outsmart deadly lasers and missiles — all to reach the final goal. We will explain the game process, code, and production process of the game.

Contents

1 Introduction

This project presents the design and implementation of a 2D physics-based platformer game titled ”Space Ball”. The core objective is to create a modular, extensible game system that supports dynamic level construction through external data files and features diverse gameplay mechanics such as grappling, laser evasion, homing projectiles, and size manipulation.

The gameplay is structured around multi-stage levels, each composed of many sub-stages. The player must navigate through a variety of hazards and challenges to reach the final goal, which is revealed only in the last stage. The game emphasizes precise movement, timing, and spatial awareness, encouraging players to adapt quickly to new mechanics introduced gradually across levels.

The main objectives of this project are:

- To design a reusable and data-driven level system that loads stage layouts from structured text files.
- To implement a rich set of interactive elements including obstacles, items, triggers, and projectiles.
- To create a smooth and responsive player control system with physics-based interactions.
- To explore modular architecture that allows for easy expansion of new game features.

2 Implementation Details

2.1 Compilation and Execution

The project is developed in Java version 24.0.1 with JavaFX software platform version 24.0.1 SDK. The source code follows a modular architecture under the package com.binge, and we compile and execute it by using IntelliJ IDEA Community Edition 2025.1.2.

2.2 Operational Procedures

All buttons can be clicked with the left mouse button, or use the tab and arrow keys to choose the button and press the enter key.

When the players start the game, they'll have four different operations available:

- Move: Players can use arrow keys or WAD keys to make character jump and moving left or right. The principle is every time the player press the corresponding button, change the speed of character in the corresponding direction, but the character can only jump at most two times in the midair.
- Pause: Player can press P key to pause or unpause the game at any time. The principle is using the class ”Timeline” in the package ”javafx.animation”. When press P, run timeline.pause() to pause the game or run timeline.play() to unpause the game.
- Hook: Player can press space key when the character in the range of grapple point, then hook the grapple point to make the character moving along the hook direction. The principle is set the moving direction and speed of the character along the hook direction.
- Shoot: Player can press F key to shoot a bullet that can break the specific objects and collect the collectible objects. The principle is create the bullet as a class extend of the class Character to make it can trigger things just like the character.

3 Program Structure and Key Methods

3.1 Overall Program Architecture

Object	**Format Description**
`initial position`	`x y` — Player start position.
`CircleObstacle`	`x y radius`.
`RectangleObstacle`	`cx cy w h angle [fatal destroyable]`.
`Checkpoint`	`x y` — Checkpoint location.
`Coin`	`x y r value` — Collectible coin.
`Lock`	`lockX lockY keyX keyY [w h]`.
`SizeShifter`	`x y r sizeChange`.
`GrapplePoint`	`x y r`.
`Goal`	`x` — Goal x-position.
`LaserObstacle`	`y x1 x2 [offset] [pulse data]`.
`VerticalLaserObstacle`	Same as `LaserObstacle`, vertical.
`SpinningLaserObstacle`	`px py len angle speed [offset] [pulse]`.
`TrackingLaserObstacle`	`x y rot range len charge fire cd [angle]`.
`HomingMissileLauncher`	`x y rot range lock fire n spread cd speed turn life [angle]`.
`SpiralMissileLauncher`	`x y aim spiral range aimT fireT int cd speed turn life [angle]`.
`Text`	`x y message...`.

Table 1: Stage file format per section

3.2 Key Classes

3.2.1 Class Hierarchy and Abstraction Design

The following diagram illustrates the relationship between major classes used in the obstacle system of the game engine:

- `Obstacle` is an `abstract class` that serves as the superclass for all obstacles in the game. It defines an interface for:
  - detecting collisions with the player character
  - updating internal state over time
- `CircleObstacle` and `RectangleObstacle` are concrete subclasses of `Obstacle`, each implementing the abstract methods using geometry specific to their shapes.
- `Collectible` is another `abstract class` representing objects the player can collect. It holds position and radius data.
- `Sublevel` acts as a container for game entities. It stores:
  - a list of `Obstacle` objects (both circles and rectangles)
  - a list of `Collectible` items
  - other components such as `Displacer`, `Lock`, `Checkpoint`, and `Goal`

3.2.2 Abstract Class: Obstacle

The `Obstacle` class is defined as an abstract base class for all types of geometric barriers the player can collide with. It holds common fields such as:

- `Point2D pos` — center position of the obstacle

- `Shape body` — the JavaFX shape used for rendering
- `boolean fatal, destroyable` — flags to define behavior on collision
- `abstract boolean checkCollision(...)` — for detecting collisions
- `abstract void update(...)` — for behavior updates (e.g., animation)

3.2.3 Derived Classes

CircleObstacle: A circular obstacle with a radius, centered at `pos`. Implements its own collision detection logic using circle-point geometry. Upon collision, velocity is reflected and position corrected.

RectangleObstacle: A rotated rectangle obstacle defined by center position, width, height, and angle. Collision handling involves rotated coordinate frames and axis-aligned projections.

Purpose of Abstraction :
Abstraction allows:

- Multiple obstacle types (e.g., circle, rotated rectangle) to share a common interface
- Easy extensibility: additional obstacles can be added without modifying existing code
- Delegating specific collision logic to the geometric nature of each shape
- Ensuring that every obstacle defines how it checks and handles collisions

3.2.4 Code Example

```java
abstract class Obstacle {
    Point2D pos;
    Shape body;
    boolean fatal, destroyable;
    double epsilon = 1e-5;

    abstract boolean checkCollision(Character c, double dispX, double dispY, double
    deltaTime);
    public abstract void update(double deltaTime);
}

class CircleObstacle extends Obstacle {
    private int radius;

    CircleObstacle(Pane pane, double x, double y, int r, Color color) {
        this.pos = new Point2D(x, y);
        this.radius = r;
        this.body = new Circle(r);
        this.body.setCenterX(x);
        this.body.setCenterY(y);
        this.body.setFill(color);
        pane.getChildren().add(this.body);
    }

    @Override
    boolean checkCollision(Character c, double dx, double dy, double dt) {
        // Collision logic here
    }

    @Override
    public void update(double deltaTime) {
        // Animation or other updates
    }
}
```

3.3 Key Methods

3.3.1 checkCollision

This game contains two main obstacles `CircleObstacle` and `RectangleObstacle`, each with a distinct collision detection method.

1. `CircleObstacle`: Let:

   - The character (circle) have:
     - Center coordinates: (x_c, y_c)
     - Radius: r_c
     - Predicted world position: $(x_p, y_p) = (x_c + \Delta x, y_c + \Delta y)$
   - The circle obstacle have:
     - Center coordinates: (x_o, y_o)
     - Radius: r_o

   Then since both the character and `CircleObstacle` are of circle shape, the distance between them is then given by

   $$d = \sqrt{(x_p - x_o)^2 + (y_p - y_o)^2}$$

   We then check whether $d < r_c + r_o$, if so, then the character collides with the circle obstacle.

2. `RectangleObstacle`: Detecting collision between the character (a circle) and a `RectangleObstacle` (a rectangle) is more involved, especially when the rectangle can be rotated. The approach typically involves transforming the circle's position into the rectangle's local coordinate system.

   Let:

   - The character (circle) have:
     - Center coordinates: (x_c, y_c)
     - Radius: r_c
     - Predicted world position: $(x_p, y_p) = (x_c + \Delta x, y_c + \Delta y)$
   - The rectangle have:
     - Center coordinates: (x_R, y_R)
     - Width: W_R
     - Height: H_R
     - Rotation angle (clockwise from positive x-axis): θ_R

   The collision detection proceeds as follows:

   1. Transform the character's predicted world position into the rectangle's local coordinate system. First, translate the character's position so the rectangle's center is the origin. Then, rotate this translated position by $-\theta_R$ to align with the rectangle's axes.

   - Relative position to rectangle center:

   $$(x_{\text{rel}}, y_{\text{rel}}) = (x_p - x_R, y_p - y_R)$$

   - Rotated local position:

   $$\begin{pmatrix} x_{\text{local}} \\ y_{\text{local}} \end{pmatrix} = \begin{pmatrix} \cos(-\theta_R) & -\sin(-\theta_R) \\ \sin(-\theta_R) & \cos(-\theta_R) \end{pmatrix} \begin{pmatrix} x_{\text{rel}} \\ y_{\text{rel}} \end{pmatrix} = \begin{pmatrix} \cos(\theta_R) & \sin(\theta_R) \\ -\sin(\theta_R) & \cos(\theta_R) \end{pmatrix} \begin{pmatrix} x_{\text{rel}} \\ y_{\text{rel}} \end{pmatrix}$$

   2. Find the closest point on the rectangle (in its local coordinates) to the character's local position. Given the rectangle's half-width $W_{R,\text{half}} = W_R/2$ and half-height $H_{R,\text{half}} = H_R/2$, the closest point $(x_{\text{clamp}}, y_{\text{clamp}})$ on the rectangle to $(x_{\text{local}}, y_{\text{local}})$ is found by clamping the local coordinates to the rectangle's bounds:

   $$x_{\text{clamp}} = \max(-W_{R,\text{half}}, \min(x_{\text{local}}, W_{R,\text{half}}))$$
   $$y_{\text{clamp}} = \max(-H_{R,\text{half}}, \min(y_{\text{local}}, H_{R,\text{half}}))$$

3. Calculate the squared distance from the character's local position to this closest point. This distance represents how far the character's center is from the closest edge/corner of the rectangle in its local space.

$$d_{\text{sq}} = (x_{\text{local}} - x_{\text{clamp}})^2 + (y_{\text{local}} - y_{\text{clamp}})^2$$

4. Check for collision. A collision occurs if the squared distance from the character's center to the closest point on the rectangle is less than the squared radius of the character.

$$d_{\text{sq}} < r_c^2$$

If this condition is met, a collision is detected.

3.3.2 handleCollision

After checking whether the character collides with any obstacle, we then update the position and the velocity of the character.

Let $\mathbf{p}$ be the position vector of a character (or projectile), and let $\mathbf{v}$ be its velocity vector. Let $\mathbf{n}$ denote the unit normal vector of the collision surface (pointing outward from the obstacle), and let d denote the penetration depth. Define a small constant $\varepsilon > 0$ to ensure numerical separation after correction.

To resolve interpenetration, the character is moved along the normal direction by a distance of $(d+\varepsilon)$:

$$\mathbf{p} \leftarrow \mathbf{p} + (d + \varepsilon)\mathbf{n}$$

This ensures that the character is displaced just outside the obstacle, avoiding repeated collisions due to residual overlap.

The velocity is decomposed into components normal and tangential to the collision surface:

$$\mathbf{v}_n = (\mathbf{v} \cdot \mathbf{n})\mathbf{n}, \quad \mathbf{v}_t = \mathbf{v} - \mathbf{v}_n$$

1. `CircleObstacle` If $\mathbf{v} \cdot \mathbf{n} < 0$ (object is moving into the surface), we apply a bounce using a restitution coefficient r:
$$\mathbf{v} \leftarrow \mathbf{v} - (1 + r)(\mathbf{v} \cdot \mathbf{n})\mathbf{n}$$

2. `RectangleObstacle` In the more general case, we reflect the normal component and dampen the tangential component using a friction coefficient μ:
$$\mathbf{v} \leftarrow -r\mathbf{v}_n + (1 - \mu)\mathbf{v}_t$$

   Where:

   - $r \in [0, 1]$ is the coefficient of restitution (lower for less bounce).
   - $\mu \in [0, 1]$ is the surface friction coefficient (higher for less sliding).

3.3.3 Loading Level from Files

Loading game stage data from external .in files is achieved through a common pattern in Java for file input and string processing. The core idea is to read the file line by line, interpret the content of each line based on a predefined format, and then use that interpreted data to construct game objects.

The general steps involved are:

1. **File Access:** The process begins by opening the target '.in' file for reading. This is typically done using Java's java.io package. The BufferedReader class, wrapped around a FileReader, is commonly used for efficient line-by-line reading of text files.

```java
BufferedReader br = new BufferedReader(new FileReader(filename));
```

2. **Line-by-Line Reading:** The file is then read sequentially, line by line, until the end of the file is reached. A 'while' loop is used for this purpose, where 'br.readLine()' retrieves each line as a 'String'.

```java
while ((line = br.readLine() != null)
```

3. **Line Cleaning and Filtering:** Each read line is usually "cleaned" by trimming leading/trailing whitespace (‘line.trim()‘) and skipping empty lines (‘line.isEmpty()‘) to handle formatting variations in the input file.

4. **Section Identification (State Machine):** The parsing logic often operates like a simple state machine. Specific keywords in the file (e.g., "initial position", "CircleObstacle") indicate the start of a new data section. A ‘String‘ variable (‘section‘ in your code) is updated to keep track of the current context. This allows the parser to know what type of data to expect on subsequent lines.

```java
if (line.equals("Keyword")) section = line;}
```

5. **Tokenization:** Once a line containing parameters for a game object is read, the line needs to be broken down into individual data units, or "tokens." This is commonly done using the ‘split()‘ method of the ‘String‘ class, often with a regular expression for whitespace as the delimiter.

```java
String[] tokens = line.split("\\s+");
```

   Each element in the ‘tokens‘ array then represents a piece of data (e.g., a coordinate, a radius, a boolean flag).

6. **Data Type Conversion:** The tokens, which are initially ‘String‘s, must be converted to their appropriate numerical (e.g., ‘double‘, ‘int‘) or boolean types. This is achieved using methods from the wrapper classes:

   - For floating-point numbers: ‘Double.parseDouble(String s)‘
   - For integers: ‘Integer.parseInt(String s)‘
   - For booleans: ‘Boolean.parseBoolean(String s)‘

```java
double x = Double.parseDouble(tokens[0]);
```

7. **Object Instantiation:** With the converted data, the corresponding game object (‘CircleObstacle‘, ‘RectangleObstacle‘, ‘Coin‘, etc.) is instantiated using its constructor.

```java
RectangleObstacle ro = new RectangleObstacle(pane, cx, cy, width, height, angle,
    Color.GRAY, fatal, destroyable);
```

8. **Adding to Data Structures:** Finally, the newly created game object is added to the appropriate collection (e.g., ‘ArrayList<Obstacle>‘, ‘ArrayList<Item>‘) within the ‘Sublevel‘ or ‘Level‘ data structure, making it available for game logic and rendering.

3.3.4 Laser Implementation Details

The overall program architecture for lasers can be described with the following sequence of steps:

1. Defined the `LaserObstacle` class in `LaserObstacle.java` with initial properties, constructor, and placeholder methods for:
   - `update()`
   - `checkCollision()`
   - `handleCollision()`
2. Implemented the blinking logic for `LaserObstacle` by:
   - Adding an `update()` method to the obstacle hierarchy.
   - Calling it from the main game loop.
   - Updating laser visibility based on a timer.
3. Implemented the `checkCollision()` method in `LaserObstacle.java`.
4. Verified the existing `handleCollision(Character c)` method in `LaserObstacle.java` correctly implements the fatal logic by calling `c.revive()` when the laser is fatal and `isOn` is true. No changes were necessary.

5. Confirmed that `LaserObstacle` instances are integrated into the game loop:
   - `update()` is invoked by the new loop in `Main.java`.
   - `checkCollision()` is called through polymorphism by the existing obstacle collision loop.
6. Added parsing and instantiation syntax for laser obstacles in `Pageloader.java` to support future use.

Key Modules and Methods for Lasers

- `update(double deltaTime)` method
- `checkCollision()` method
- `handleCollision()` method

Relevant Classes

- `LaserObstacle` (`LaserObstacle.java`)
- `TrackingLaserObstacle` (`TrackingLaserObstacle.java`)
- `PageLoader` (`PageLoader.java`)

The `LaserObstacle` class follows a similar design philosophy to the `TrackingLaserObstacle`, but with a simpler, static behavior and pulsing visual effects. While it lacks dynamic tracking, it shares core collision logic and state-based activation. Therefore, I will introduce the Tracking Laser first and talk about difference with `LaserObstacle` Class later.

Tracking Laser Class

The `update()` method is the core logic of the tracking laser obstacle, handling state transitions, laser behavior, and visual updates. The laser cycles through the following states:

- **IDLE (scanning for player)**
  The laser scans for the player within its `detectionRange`. If the player enters range, it transitions to the `TRACKING` state.
- **TRACKING (rotating toward player)**
  Calculates the angle to the player using `Math.atan2`. The laser smoothly rotates toward the player using `rotationSpeedRadiansPerSec`. If the player is aimed at (within a $5°$ tolerance or after 0.5 seconds), it transitions to `CHARGING`. If the player leaves detection range, it reverts to `IDLE`.
- **CHARGING (preparing to fire)**
  The laser waits for `chargeDurationSecs` before firing. Once charging completes, it transitions to `FIRING`.
- **FIRING (emitting laser beam)**
  While firing, the laser continues to track the player. It updates the laser beam's endpoint (`laserFireTargetPoint`) based on the current angle. After `fireDurationSecs` elapses, the laser transitions to `COOLDOWN`.
- **COOLDOWN (recovery after firing)**
  The laser waits for `cooldownDurationSecs` before resetting. It then transitions back to `IDLE` (or `TRACKING` if the player is still in range).

Collision Detection

The `checkCollision()` method determines whether the laser beam (when active) collides with the player. It uses line-circle intersection logic for precise detection:

- `lineVec` = `p2 - p1` (laser direction vector)

- `startToChar` = `playerPos - p1` (vector from emitter to player)
- `t` = `(startToChar · lineVec) / ||lineVec ||`^{2} (projection parameter)
- `closestPoint` = `p1 + (lineVec * clamped_t)`
- If `distance(closestPoint, playerPos) < playerRadius`, a collision is detected.

Collision Handling

The `handleCollision(Character c)` method defines the consequences of a laser-player collision:

- If the laser is fatal (`this.fatal = true`), it calls `c.revive()` to respawn the player.

Integration

The `Pageloader.java` file is responsible for parsing and instantiating the laser obstacle for future use in level design and loading.

LaserObstacle Class

The `LaserObstacle` class adheres to the same foundational principles as `TrackingLaserObstacle`, including collision detection, fatal interactions, and visual state synchronization. However, it trades dynamic tracking for efficiency, using timer-based activation and axis-aligned bounding boxes for performance-critical scenarios. Pulsing effects provide visual polish without complex state machines.

An `enum` is implemented for `LaserOrientation` to define the direction of the laser in a clean and readable manner.

For implementation details, refer to the `TrackingLaserObstacle` analysis—core methodologies apply analogously.

Shared Core Logic

Feature	**TrackingLaserObstacle**	**LaserObstacle**
Collision Detection	Line-circle intersection	Axis-aligned bounding box (AABB)
State Management	FSM (IDLE, TRACKING, etc.)	Binary (ON/OFF timer)
Fatal Interaction	Instantly revives player	Same behavior
Visual Feedback	Color changes + beam visibility	Pulsing thickness + visibility

Spinning Laser Obstacle

Design Concept	The SpinningLaserObstacle combines elements from both static LaserObstacle and dynamic TrackingLaserObstacle, creating a balanced middle-ground between static and tracking lasers. While it lacks target-seeking behavior, its constant rotation creates predictable yet challenging patterns ideal for arena-style gameplay.
Technical Highlights	
• ***Efficient Rotation***	Only 2 trigonometric calls per frame ($\cos\theta$ /$\sin\theta$)
• ***Collision System***	Projects player position onto laser segment
• ***Inheritance***	Reuses line-circle collision from TrackingLaserObstacle
• ***Visuals***	Maintains LaserObstacle's pulsing effects

Table 2: Technical specification of SpinningLaserObstacle implementation

Feature	Static LaserObstacle	TrackingLaserObstacle	SpinningLaserObstacle
Movement	None	Player-tracking	Constant rotation
Collision Method	AABB	Line-circle	Line-circle
State Management	Timer-based	FSM	Timer-based
Visual Complexity	Pulsing only	Color changes	Pulsing + rotation
Use Case	Environmental hazards	Smart enemies	Dynamic arena hazards

Table 3: Comparison of different laser obstacle implementations

Homing Missile Implementation Overview

Core Components

- `HomingLaserProjectile.java` - Projectile behavior
- `HomingMissileLauncherObstacle.java` - Emitter system
- `Main.java` - Projectile management

Implementation Details

Component	Implementation
1. Projectile Class	• Properties: position, velocity, speed, turn rate, lifespan, target • Visual: Circle body with color • Methods: update(), collision detection, activation control
2. Update Logic	• Lifespan countdown and auto-deactivation • Homing behavior: angle calculation + clamped turning • Position and visual updates
3. Collision System	• Circle-to-circle detection with player • Automatic deactivation on hit • Returns true for successful collision
4. Main Integration	• `activeProjectiles` ArrayList • Update loop for all active projectiles • Automatic cleanup of inactive projectiles
5. Launcher Class	• State machine (IDLE, TRACKING, LOCKON, FIRING, COOLDOWN) • Projectile configuration parameters • Visual emitter body
6. Launcher Update	• State transitions and timing • Player detection and aiming • Projectile instantiation and volley control • Visual feedback (color changes)

Table 4: Detailed implementation breakdown

Key Technical Features

Class Interactions

- Launcher creates projectiles → added to `Main.activeProjectiles`
- Main game loop updates all active projectiles

Aspect	Solution
State Management	Finite State Machine pattern
Collision Detection	Circle-to-circle for projectiles
Projectile Steering	Clamped angular adjustment
Performance	Only 2 trig calls per frame
Visual Feedback	Color changes and pulsing effects

Table 5: Technical highlights

- Projectiles handle own collision detection
- Launcher body is non-fatal (collision returns false)

Collision Detection Logic

Collision with Player(projectile logic)

- Compute squared distance between projectile and player center positions.
- Sum the radii of both projectile and player.
- If distance squared < (sum of radii)^2, a collision has occurred.
- On collision:
  - Set projectile as inactive.
  - Remove it from the pane (handled via `setActive()`).
  - Return `true`.

Collision with Obstacles(porjectile logic)

- Loop through all obstacles and skip invisible ones.
- **For Circular Obstacles**:
  - Use radius-based collision: check if distance squared < (sum of radii)^2.
- **For Rectangular Obstacles**:
  - Use Axis-Aligned Bounding Box (AABB) intersection via `getBoundsInParent()`.
- On any collision:
  - Set projectile as inactive.
  - Remove it from the pane.
  - Return `true`.
- If no collision is detected with any obstacle, return `false`.

4 Challenges and Solutions

4.1 Penetration of Character into Obstacles

4.1.1 Problem Statement

The very first challenge we encountered was that the character would sometimes penetrate through obstacles. This problem happened because the collision response failed to adequately separate the ball from the obstacle after impact. As a result, the ball may remain partially embedded within the obstacle, leading to repeated or simultaneous collision detections over successive frames, even if the ball is not moving at high speed.

4.1.2 Solution

To address the issue of repeated collisions caused by insufficient post-collision separation, a predictive collision handling method is employed. Instead of resolving collisions reactively after they occur, this approach anticipates potential collisions before the ball fully penetrates an obstacle. By analyzing the ball's current trajectory and velocity, the system determines whether a collision is imminent in the upcoming frame.

If a collision is predicted, the position that the ball should occupy immediately after bouncing is computed in advance, and the ball is directly placed at this post-collision location in the next frame. This predictive repositioning ensures that the ball is never allowed to remain in an overlapping state with an obstacle, thereby eliminating the possibility of redundant or chained collision detections. The bounce direction and magnitude are derived based on the expected collision normal and restitution properties, ensuring physically plausible responses.

This method effectively transforms the collision response from a reactive correction into a proactive adjustment, reducing computational overhead and improving stability. Moreover, it prevents the ball from entering ambiguous configurations where multiple simultaneous collisions would otherwise occur due to persistent interpenetration.

4.2 Level design process

4.2.1 Problem Statement

During the level design process, a key challenge arose from the **discrepancy between initial design ideas and actual gameplay behavior**. While level layouts and obstacle sequences were planned with specific intentions, the in-game outcomes often failed to match these expectations.

Small variations in object positions, movement patterns, or activation timing frequently resulted in issues such as:

- Unintended difficulty spikes
- Player movement being blocked unexpectedly
- Interactions between elements not behaving as expected

This mismatch was largely caused by the complex interplay of **physics calculations**, **pixel-based placements**, and **timing-sensitive mechanics**, all of which were configured manually through text files.

4.2.2 Solution

To resolve this issue, an **iterative trial-and-error approach** was adopted. Levels were tested repeatedly with minor parameter adjustments made between each run.

This involved:

- Tweaking the exact coordinates of objects pixel by pixel
- Adjusting speeds, angles, and durations
- Observing how changes affected player experience

Although this process was time-consuming, it ultimately led to better-balanced stages with smoother pacing. Over time, the team developed a more intuitive understanding of how each setting impacted gameplay, allowing for faster and more confident design decisions in later stages.