Project Arcade Roblox

San

Goal: 100,000 players and an average session of 8 minutes

Contents

I Game Design **1**

1 General Summary **1**
1.1 Game Title . . . 1
1.2 Elevator Pitch . . . 1
1.3 Genre . . . 1
1.4 Platforms . . . 1
1.5 Target Audience . . . 1

2 Game Concept **2**
2.1 Main Mechanics . . . 2

3 Art Style **3**
3.1 The Shared Building Style . . . 3
3.2 How Each Building Is Recognized . . . 4
3.3 The Sign of Each Building . . . 4
3.4 Palette and Decoration of Each Building . . . 5

4 Music and Sound **5**
4.1 Environment Music and Sounds . . . 6

5 Game Features **6**
5.1 Player Actions . . . 6
5.2 Scenarios and Environments . . . 7
5.3 Death and Restart System . . . 7
5.3.1 Ways to Die . . . 7
5.3.2 Restart System . . . 8
5.4 Multiplatform . . . 8

6 Level Design **8**
6.1 Description . . . 8
6.2 Maps and Schemes . . . 9

II Engineering, Production and Launch **9**

7 Technical Design **9**
7.1 Game Engine . . . 9
7.2 Development Schedule . . . 9
7.3 Optimization . . . 11
7.4 System Requirements . . . 11
7.4.1 PC . . . 11
7.4.2 Mobile (Android and iOS) . . . 12
7.4.3 Xbox Series S/X . . . 12

7.4.4 PlayStation . . . 12
7.5 Testing and QA . . . 12

8 Marketing Plan **13**
8.1 Monetization . . . 13
8.2 Marketing Strategy . . . 13
8.2.1 Channels . . . 14
8.2.2 Content Plan . . . 14
8.2.3 Rollout Phases . . . 14
8.3 Why This Fits the Audience . . . 15

III Technical Implementation **15**

9 System Architecture Overview **15**

10 Roblox Structure **17**

11 Coding in Roblox by Game Type **22**
11.1 Oxybelis . . . 24
11.1.1 Rules of Oxybelis . . . 24
11.1.2 Project Structure . . . 25
11.1.3 Time Complexity . . . 26
11.1.4 Space Complexity . . . 26
11.1.5 Game Features and Implementation Status . . . 27
11.1.6 Game Results . . . 27
11.1.7 VIP and Global Scores . . . 28
11.2 Avis Toucan . . . 29
11.2.1 Rules of Avis Toucan . . . 29
11.2.2 Project Structure . . . 30
11.2.3 Time Complexity . . . 30
11.2.4 Space Complexity . . . 30
11.2.5 Game Features and Implementation Status . . . 31
11.2.6 Game Results . . . 31
11.3 Resilio . . . 33
11.3.1 Rules of Resilio . . . 33
11.3.2 Project Structure . . . 34
11.3.3 Time Complexity . . . 35
11.3.4 Space Complexity . . . 35
11.3.5 Game Features and Implementation Status . . . 36
11.3.6 Game Results . . . 36
11.4 Foxy Fruit . . . 38
11.4.1 Rules of Foxy Fruit . . . 38
11.4.2 Project Structure . . . 39
11.4.3 Time Complexity . . . 40
11.4.4 Space Complexity . . . 40

11.4.5 Game Features and Implementation Status . . . 41
11.4.6 Game Results . . . 41
11.5 Firmament Defender . . . 43
11.5.1 Rules of Firmament Defender . . . 43
11.5.2 Project Structure . . . 44
11.5.3 Time Complexity . . . 45
11.5.4 Space Complexity . . . 45
11.5.5 Game Features and Implementation Status . . . 46
11.5.6 Game Results . . . 47
11.6 Firmament Siege . . . 49
11.6.1 Rules of Firmament Siege . . . 49
11.6.2 Project Structure . . . 50
11.6.3 Time Complexity . . . 50
11.6.4 Space Complexity . . . 51
11.6.5 Game Features and Implementation Status . . . 51
11.6.6 Game Results . . . 52
11.7 Juice Cutter . . . 54
11.7.1 Rules of Juice Cutter . . . 54
11.7.2 Project Structure . . . 55
11.7.3 Time Complexity . . . 56
11.7.4 Space Complexity . . . 56
11.7.5 Game Features and Implementation Status . . . 57
11.7.6 Game Results . . . 57
11.8 Turris Casei . . . 59
11.8.1 Rules of Turris Casei . . . 59
11.8.2 Project Structure . . . 60
11.8.3 Time Complexity . . . 61
11.8.4 Space Complexity . . . 61
11.8.5 Game Features and Implementation Status . . . 62
11.8.6 Game Results . . . 63
11.9 Tap And Hold . . . 65
11.9.1 Rules of Tap And Hold . . . 65
11.9.2 Project Structure . . . 66
11.9.3 Time Complexity . . . 67
11.9.4 Space Complexity . . . 67
11.9.5 Game Features and Implementation Status . . . 68
11.9.6 Game Results . . . 69
11.10 ImNotASphere . . . 71
11.10.1 Rules of ImNotASphere . . . 71
11.10.2 Project Structure . . . 72
11.10.3 Time Complexity . . . 73
11.10.4 Space Complexity . . . 74
11.10.5 Game Features and Implementation Status . . . 75
11.10.6 Game Results . . . 76

12 Shared Systems **78**
12.1 Score Table System . . . 78
12.1.1 Architecture . . . 79
12.1.2 Data Flow . . . 79
12.1.3 Games Using This System . . . 80
12.2 Menu System . . . 81
12.2.1 Menu1to7 (Single Player Menu) . . . 82
12.2.2 Menu8to10 (Menu + Multiplayer Lobby) . . . 82
12.3 Universal Controller . . . 82
12.3.1 Menu Navigation Helpers . . . 83
12.3.2 Per Game Control Bindings . . . 83
12.3.3 Virtual Control Widgets . . . 84
12.4 Multiplayer Lobby System . . . 84

13 In Game Data Analytics System **86**
13.1 Architecture . . . 87
13.2 Data Collected Per Game . . . 87
13.3 Chart Engine . . . 88
13.4 Board Types . . . 88
13.5 Data Flow . . . 89

14 Fireworks System **89**
14.1 Architecture . . . 90
14.2 The Launch Economy . . . 91
14.3 Rendering Pipeline . . . 92
14.3.1 The Three Layer Burst . . . 93
14.4 Firework Kinds . . . 93
14.5 The Nineteen Fireworks . . . 94
14.5.1 Why Each One Looks Different . . . 94
14.6 Cinematic Camera . . . 95
14.7 Visibility Toggle . . . 95

Part I

Game Design

1 General Summary

1.1 Game Title

The game does not have a final name yet. The current working title is **2 Minutes**, but this may change before release. For now, the project is referred to internally as the arcade world.

1.2 Elevator Pitch

The player walks into a small world full of arcade machines. Each machine is a different game: Oxybelis, Avis Toucan, Resilio, Foxy Fruit, Firmament Defender, Firmament Siege, Juice Cutter, Turris Casei, Tap And Hold, and ImNotASphere. Each one has its own simple challenge, from steering a growing creature around a grid to stacking blocks or rolling a ball through a 3D course. The player is free to try any of them, beat their own score, climb a global top ten, or simply wander around and watch other people play. The whole idea is to spend a relaxed eight minutes or more moving from one machine to the next, with no pressure and nothing to grind.

1.3 Genre

This is not a game with a single genre. It is a **casual arcade collection** that brings several classic styles together in one place, so action, puzzle, reflex, and precision games all sit side by side. The common thread is that every game is short, easy to understand, and easy to pick up for a few minutes at a time.

1.4 Platforms

The game was made in Roblox Studio, so it runs anywhere Roblox runs. It was designed **mainly for PC**, and then adapted so the controls and the interface also work well on other devices:

- PC with keyboard and mouse, which is the main platform.
- Mobile (Android and iOS), with touchscreen controls.
- Xbox, with a gamepad.
- PlayStation, with a gamepad.

All ten games share the same input layer, so each game feels natural on every device without needing separate versions.

1.5 Target Audience

This game is made for **casual players** who simply want to have a good time. The ideal player enjoys wandering around a small world, trying a game for a few minutes, watching other players, or just

exploring without any pressure. There is no ranking stress, no grinding, and no complex mechanics to learn.

The project takes inspiration from games like *Are You OK?*, where the experience itself is the point and not the competition. A player can spend eight minutes jumping between arcade machines, or spend those same eight minutes just walking around the map. Both are perfectly fine.

This game is for:

- Casual players who enjoy simple and fun games.
- Explorers who like discovering small details in a game world.
- Players who enjoy weird and charming experiences.
- People who just like watching what happens around them.

This game is not for:

- Competitive players looking for ranked matches.
- Players who need a deep story or complex progression.
- Speedrunners or players who like to optimize every detail.

2 Game Concept

The main goal of the whole project is simple: **keep the player inside the world for at least eight minutes**, moving from one game to another instead of leaving after just one. Each game is a small arcade machine that the player can walk up to and use. There is no main story; the world itself and the variety of games are the experience.

2.1 Main Mechanics

Each of the ten games has its own core mechanic. They are kept short and readable on purpose, so a new player understands what to do within seconds.

Game	Main Mechanic
Oxybelis	Steer a growing creature around a grid, eat food to grow, and avoid hitting the walls or your own body.
Avis Toucan	Tap to make a bird flap upward against constant gravity, and pass through the gaps between moving pipes.
Resilio	Move a paddle left and right to bounce a ball into the bricks and clear the stage; some bricks give power ups.
Foxy Fruit	Move through a maze eating raspberries while four bears chase you; eat a blueberry to turn the bears vulnerable for a few seconds.
Firmament Defender	Move a ship left and right and shoot a descending swarm of planes before they reach your row; hide behind destructible shields.
Firmament Siege	Fly a ship freely in 360 degrees with inertia, aim in any direction, and destroy drones that chase you while surviving as long as possible.
Juice Cutter	Swipe to slice the fruits named in the level recipe, while avoiding the wrong fruits that cost you a life.
Turris Casei	Time your drop so each swinging block lands on top of the previous one; any overhang gets sliced off.
Tap And Hold	Hold to charge a jump and release to launch your character in an arc onto the next platform without falling.
ImNotASphere	Roll a ball through 3D levels, collect coins, use power ups, avoid the kill zones, and reach the finish line.

Table 1: Core mechanic of each game.

3 Art Style

The whole game takes place in a small coastal town at sunset. The sky is a strong orange, the sea is light blue, the sand has a soft glow along its edges, and the streets are made of brick cobblestone that is designed to reflect the light so the ground stands out. White lamps light the town as the evening sets in. This warm and calm scene is the base that every building has to fit into.

Element	Color	Hex
Sunset sky	strong orange	`#FF7A18` to `#FF9E3D`
Sea	light blue	`#7FC9E8`
Sand (glowing edges)	warm pale with a soft glow	`#E8D2A0` / glow `#FFE9B0`
Brick cobblestone ground	warm tan that reflects light	`#B8A98C` / highlight `#D8CBB0`
Lamp light	warm white	`#FFF4D6`

Table 2: Color palette of the town and its environment.

3.1 The Shared Building Style

Every building uses the same medieval European look, similar to an old coastal town like the ones in northern Italy. They share the same materials so the whole town feels like one place: warm stone walls, dark wooden beams in the Tudor style, and cream plaster between the beams. Most importantly, every building has the same roof: red terracotta tiles, like brick. This single roof color was chosen on purpose, because a deep red stands out clearly against the orange sky and ties all the buildings together into one town.

Building material	Color	Hex
Stone walls	warm sandstone	`#D8C8A8`
Wooden beams (Tudor style)	dark oak	`#3A2A1C`
Plaster between beams	cream	`#EFE6D0`
Roof tiles (same on every building)	terracotta brick red	`#A23B22`

Table 3: Shared building materials, used by all ten buildings.

3.2 How Each Building Is Recognized

Since the roof is the same red on every building, the player tells the buildings apart in two other ways: the **accent colors** (the banners, shutters, door, and window frames, all painted in the game's own color) and a big **friendly themed decoration** on the outside. For example, the Oxybelis building has a giant but friendly green snake coiled around its tower and doorway, so a player can see from across the town that it is the snake game. Each accent color comes from the color that the game already uses inside its menus and leaderboard, so the building and the game match.

3.3 The Sign of Each Building

Every building has a hanging wooden sign with the name of the game. There is one simple rule for the colors of the sign, based on contrast with the orange sky:

- **Cool or already dark game colors** (blue, cyan, green, magenta, purple, rose): the **sign board is painted in the game's color** and the **letters are white or cream**. These colors are bold and read well from a distance.
- **Warm game colors that blend into the sky** (the cheese gold of Turris Casei and the amber of Foxy Fruit): the **sign board is dark** (brown or deep indigo) and the **letters are in the game's warm color**. The dark board gives the sign a clear shape against the orange sky, while the letters stay on theme.

As an extra touch, every sign has a soft edge of warm lamp light (`#FFF4D6`) around it, so the signs are easy to read at sunset.

3.4 Palette and Decoration of Each Building

Game	Accent colors	Themed decoration	Sign
Oxybelis	emerald `#10B981`, lime `#9AE600`	a giant friendly green snake coiled around the tower and doorway, smiling	emerald board `#059669`, cream letters, coiled snake
Avis Toucan	cyan `#06B6D4`, white trim	a big friendly toucan perched over the door with a bright orange beak, with tropical leaves	cyan board `#0891B2`, white letters, toucan
Resilio	magenta `#EC4899`	the wall looks like bricks with a few glowing broken ones, and a giant ball and paddle over the entrance	magenta board `#DB2777`, white letters
Foxy Fruit	amber `#FBBF24`, raspberry `#E11D48`, blueberry `#3B82F6`, indigo shutters `#3730A3`	a friendly fox sitting by the door, with berry bushes and a hedge maze pattern on the wall	dark indigo board `#3730A3`, amber letters, fox and a berry
Firmament Defender	violet `#A855F7`, cyan neon `#22D3EE`	small pixel style plane figures along the top edge and a little ship over the door	violet board `#7E22CE`, white letters
Firmament Siege	violet `#8B5CF6`, silver `#CBD5E1`	floating asteroid rocks and a triangular ship over the entrance, with stars	indigo board `#5B21B6`, silver letters
Juice Cutter	lime `#84CC16`, lemon `#FACC15`, strawberry `#EF4444`	giant fruit sculptures and a splash of sliced fruit above the sign	lime board `#65A30D`, white letters
Turris Casei	cheese gold `#EAB308`, dark timber	the wall looks like stacked blocks or cheese wheels, with a giant cheese wedge over the door	dark brown board `#4A2C1A`, gold letters, cheese wedge
Tap And Hold	rose `#FB7185`, crimson `#E11D48`	stacked platforms climbing the wall and a friendly character frozen in mid jump	crimson board `#BE123C`, cream letters
ImNotASphere	blue `#3B82F6`, sky blue `#93C5FD`	a giant blue sphere resting in a curved ramp on the wall	blue board `#2563EB`, white letters, sphere

Table 4: Accent colors, themed decoration, and sign for each of the ten buildings.

A few notes guided these choices. The roof is never orange or yellow, because those colors would disappear against the sky; the deep red was chosen for exactly that reason. The two Firmament buildings are siblings: Defender is brighter (violet and cyan), while Siege is deeper (indigo and silver), so they look related but different. Avis Toucan (cyan) and ImNotASphere (blue) stand out the most, because blue and cyan are the opposite of orange on the color wheel.

4 Music and Sound

The project does not use voice acting or original composed music yet. Instead, each game uses short Roblox sound assets for feedback: a sound when the player scores, when they lose, when they select a menu option, and so on. Many of these sounds are shared across games because they play the same role (a menu click, a game over sound), and this keeps the whole arcade feeling consistent.

4.1 Environment Music and Sounds

The table below lists the actual sound asset IDs used in each game, taken directly from the game clients and the shared menu modules. Several IDs repeat across games because they are reused for the same purpose.

Game	**Sound (purpose)**	**Asset ID**
Shared menu	Hover (move selection)	rbxassetid://500472891
Shared menu	Select (confirm)	rbxassetid://70844388753756
Shared menu	Game over	rbxassetid://5855421995
Shared menu	Perfect / win	rbxassetid://89676801997437
Oxybelis	Eat food	rbxassetid://2940337254
Avis Toucan	Flap / score	rbxassetid://92876108656319
Resilio	Brick destroyed	rbxassetid://92876108656319
Foxy Fruit	Eat raspberry	rbxassetid://98007189797443
Foxy Fruit	Select	rbxassetid://70844388753756
Firmament Defender	Laser shot	rbxassetid://137510557013265
Firmament Defender	Explosion	rbxassetid://92876108656319
Firmament Siege	Laser shot	rbxassetid://137510557013265
Firmament Siege	Explosion	rbxassetid://92876108656319
Juice Cutter	Cut fruit	rbxassetid://6881026094
Juice Cutter	Wrong cut (error)	rbxassetid://1313873191
Juice Cutter	Select	rbxassetid://70844388753756
Turris Casei	Place block	rbxassetid://4612375233
Turris Casei	Perfect placement	rbxassetid://6895079853
Tap And Hold	Jump	rbxassetid://6895079853
ImNotASphere	Collect coin	rbxassetid://87437544236708
ImNotASphere	Power up	rbxassetid://500472891

Table 5: Sound asset IDs per game. Turris Casei, Tap And Hold, and ImNotASphere also reuse the shared hover, select, and game over sounds.

5 Game Features

5.1 Player Actions

The use case diagram below sums up everything a player can do in the world. Most of these actions are open to anyone at any time. Creating a private room and joining a room with a PIN are the two that lead to playing a game together with other people; the rest (playing alone, donating, reading a leaderboard or the analytics boards, or simply walking around) need only the single player.

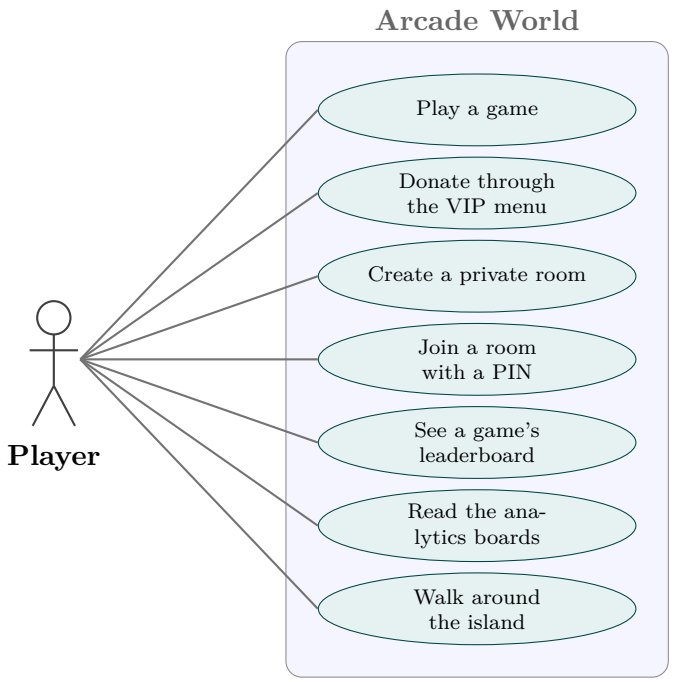


Figure 1: Use case diagram of the player's actions inside the arcade world.

5.2 Scenarios and Environments

The game takes place in a single connected world, not in separate level screens. Inside this world, the player finds the ten arcade machines, each one a physical object they can walk up to and activate. The world also holds the global leaderboard boards and a bank themed building that shows live data about how every game is being played (this is explained later in the data analytics section). Everyone shares the same world, so one person can play a game while others watch or play on nearby machines.

5.3 Death and Restart System

Because the project is a collection of ten different games, there is no single rule for losing. Each game decides on its own when the run ends, but they all bring the player back to the same kind of game over screen afterwards, so the experience feels consistent across the whole arcade.

5.3.1 Ways to Die

Game	**How the run ends**
Oxybelis	Crashing into a wall or into its own body.
Avis Toucan	Hitting a pipe or the ground.
Resilio	Losing all 3 lives when the ball falls below the paddle.
Foxy Fruit	Being caught by a bear; the run ends after all 3 lives are lost.
Firmament Defender	Losing all 4 lives to enemy fire, or any plane reaching the ship's row.
Firmament Siege	Losing all 3 lives to drone collisions.
Juice Cutter	Losing all lives by cutting the wrong fruits.
Turris Casei	A block missing the platform entirely.
Tap And Hold	Falling off a platform.
ImNotASphere	Losing all 8 lives by falling into a kill zone or off the level.

Table 6: What ends the run in each game.

5.3.2 Restart System

When a run ends, the player sees a game over screen with their final score and two options: play again or go back to the arcade. Playing again restarts that same game right away, so there is no loading and no need to walk back to the machine. Games that use lives (such as Resilio, Foxy Fruit, the Firmament games, and ImNotASphere) let the player respawn and keep going until all the lives are gone, and only then show the game over screen.

In the three multiplayer games (Turris Casei, Tap And Hold, ImNotASphere), a versus match does not end the moment one player dies. Instead, players who have been eliminated switch to spectator mode and can watch the others until the match finishes and a winner is chosen.

5.4 Multiplatform

All ten games run on PC, mobile, Xbox, and PlayStation from the same code. This works because every game sends its input through one shared controller layer. On PC it reads the keyboard and mouse, on console it reads the gamepad, and on mobile it shows touch controls on the screen that are built for that specific game (a direction pad, a single button, a joystick, and so on). The menus follow the same logic, so moving the selection and confirming a choice work the same way with a keyboard, a gamepad, or a tap on the screen. The player never has to pick a "mobile version" or a "console version"; the game simply adapts to whatever device is being used.

6 Level Design

6.1 Description

The ten games do not all handle levels in the same way. Some have a fixed number of handmade levels, some run forever and get harder over time, and some have no levels at all and simply keep going until the player loses. This variety is on purpose, because it keeps each machine feeling different from the one next to it.

Game	**Level Type**	**Description**
Oxybelis	Endless	One single grid; the game keeps going until the player crashes. The only goal is the score.
Avis Toucan	Infinite	Endless pipes that never stop; the run lasts as long as the player survives.
Resilio	Limited (5)	Five handmade brick layouts; clearing the last one is a perfect game.
Foxy Fruit	Limited (3)	Three handmade maze maps that get more complex as the player advances.
Firmament Defender	Limited (3)	Three waves that get faster each time.
Firmament Siege	Infinite	Endless survival; drones keep appearing and the game has no win state.
Juice Cutter	Limited (5)	Five levels, each one with its own fruit recipe and difficulty.
Turris Casei	Endless	Stack blocks until one misses the platform.
Tap And Hold	Infinite	Platforms generated on the fly that never end.
ImNotASphere	Limited (3)	Three handmade 3D levels with platforms, coins, and a finish line.

Table 7: Level structure of each game.

6.2 Maps and Schemes

To be updated. The detailed maps and layout schemes for each game will be added in a future revision of this document.

Part II

Engineering, Production and Launch

7 Technical Design

7.1 Game Engine

The game was built entirely in **Roblox Studio**, using the Luau language. Server logic lives in ServerScriptService, shared game logic and physics live in ReplicatedStorage modules, and all interface, controls, and rendering live in StarterPlayer client scripts. The project is synced between Visual Studio Code and Roblox Studio using Rojo, which keeps the source code organized in a normal folder structure outside the engine.

7.2 Development Schedule

The project was built almost entirely alone and with no real prior experience in Roblox, so the pace was about one game per week. The focused work began on the first of February 2026, and the ten games were built one after another, each in roughly a week (around three days for the logic and three days to fix bugs and make it playable). After the ten games came one week of refactoring the code, one week writing this documentation, three weeks to design the island and its art, a longer stretch of testing and bug fixing across PC, mobile, and console, and finally one week to plan the marketing and record a trailer. With this pace, the planned release is the second week of July 2026. The Gantt chart below shows the order and the rough length of each part.

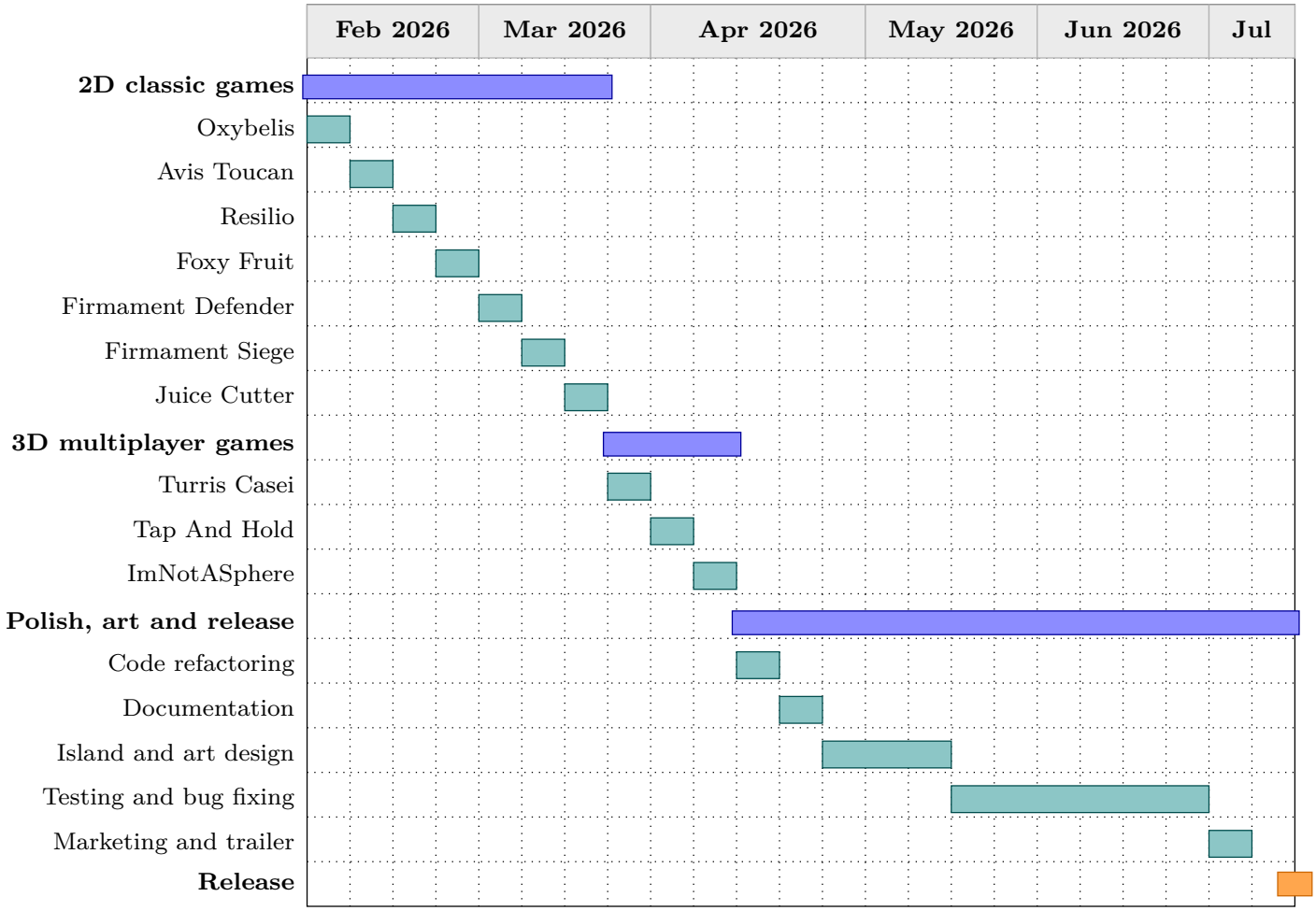


Figure 2: Development schedule, from the first game in February 2026 to the planned release in the second week of July 2026.

The next chart breaks down the work for each game into the two phases: the days spent on the logic and the days spent polishing and fixing bugs. It makes clear that the 3D multiplayer games (Turris Casei, Tap And Hold, and ImNotASphere) were the most expensive, while the first and simplest games were the quickest.

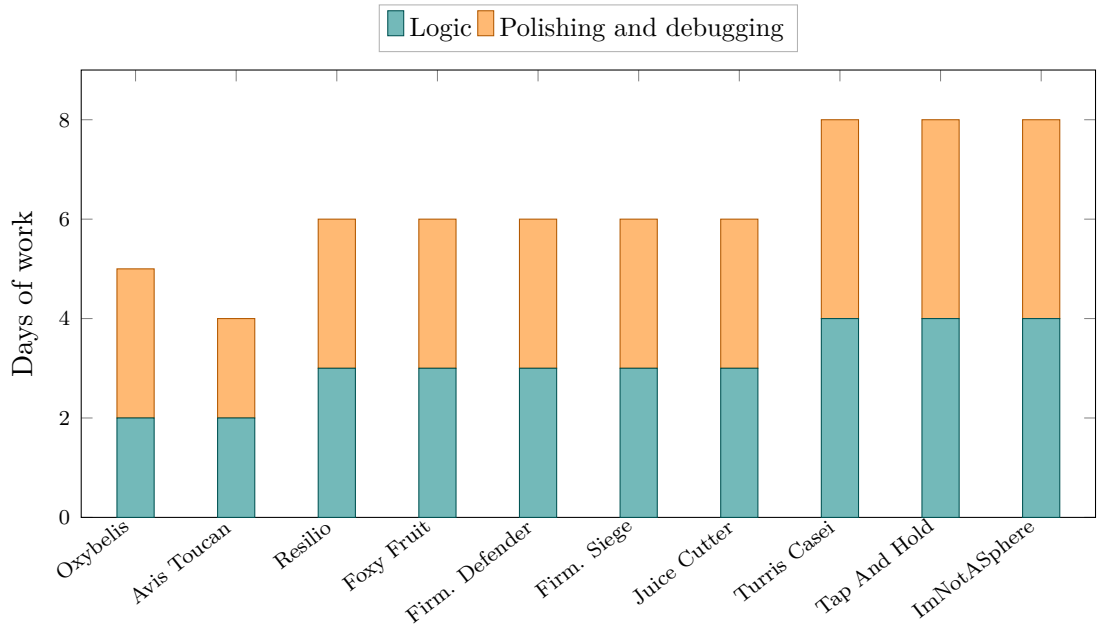


Figure 3: Days of work per game, split into logic and polishing or debugging.

7.3 Optimization

Because the arcade has to run smoothly on phones and consoles, every game was written with its performance cost in mind. Most game loops were kept at a constant or almost constant cost per frame, and memory use was always kept within a fixed limit, so nothing grows out of control during a session. The table below gives a short summary for each game; the detailed analysis is in each game's own section earlier in this document.

Game	**Cost per tick**	**Memory**
Oxybelis	$O(1)$ via a circular buffer	$O(N)$, fixed grid (192 cells)
Avis Toucan	$O(P)$ with $P \leq 3$ pipes, so $O(1)$	$O(P)$, bounded
Resilio	$O(B)$ bricks, decreasing per level	$O(B)$, $B \leq 120$
Foxy Fruit	$O(E \times N)$ with $E = 4$, $N = 192$ fixed	$O(N)$, fixed grid
Firmament Defender	$O(N)$ planes, $N \leq 30$, decreasing	$O(N)$, bounded
Firmament Siege	$O(N \times M)$ with $N \leq 7$, $M \leq 10$	$O(N + M)$, bounded
Juice Cutter	$O(F)$ fruits, $F \leq 10$, so $O(1)$	$O(F)$, bounded
Turris Casei	$O(1)$ per block drop	$O(1)$ game logic
Tap And Hold	$O(1)$ per jump	$O(1)$, max 4 platforms
ImNotASphere	$O(1)$ per physics tick (stateless module)	$O(1)$ game logic

Table 8: Optimization summary for each game. The three multiplayer games also keep $O(P)$ lobby data for up to $P \leq 50$ players (10 for ImNotASphere).

7.4 System Requirements

Because the game runs on the Roblox platform, the requirements match Roblox's own minimum requirements rather than custom hardware targets.

7.4.1 PC

- Windows 10 or newer (or macOS 10.13+).

- A processor from 2015 or later, with at least 4 cores (around 2.5 GHz or faster).
- At least 8 GB of RAM.
- An internet connection (the game is online and uses DataStore for leaderboards).
- Keyboard and mouse.

7.4.2 Mobile (Android and iOS)

- Android: any device running Android 10 or newer with at least 8 GB of RAM.
- iOS: iPhone 14 or newer.
- A touchscreen (the touch controls are built automatically).
- An internet connection.

7.4.3 Xbox Series S/X

- An Xbox console that supports the Roblox app (Xbox One and Series S/X).
- An Xbox controller (the gamepad layout is supported in every game).
- An internet connection.

7.4.4 PlayStation

- A PlayStation console that supports the Roblox app (PS4 and PS5).
- A DualSense or DualShock controller (same gamepad layout as Xbox).
- An internet connection.

7.5 Testing and QA

The testing was done by hand, one piece at a time, since the project was built alone. There was no automated test suite; instead, each part was checked directly inside Roblox Studio while it was being built, and again after each change. The focus was always the same: make sure the game stays playable and that nothing breaks the calm feeling of the world.

The main things that were checked, game by game, were:

- **Menus:** that every screen opens and closes correctly, that the PLAY, VIP, EXIT, and MENU buttons go where they should, and that the donation menu shows the right products.
- **Sounds:** that each sound plays at the right moment (scoring, losing, selecting), and that the mute button turns all audio on and off.
- **Buttons and input:** that every button reacts to a click and a tap, and that the keyboard controls (WASD, arrow keys, Space) work in each game.
- **Mobile:** using the device emulator inside Roblox Studio to act like a phone, checking that the on screen touch controls appear, that they are placed well, and that the games can be fully played with touch.

- **Gamepad:** that the menus and the games respond to a controller, so the Xbox and PlayStation layouts behave like the keyboard one.
- **Leaderboards and saving:** that a finished game reports its score, that the top 10 boards update, and that audio settings come back correctly after rejoining.

Each game was only marked as working once it had passed these checks on PC and in the mobile emulator. The known gaps and anything still pending are listed in the per game feature tables earlier in this document.

8 Marketing Plan

This game is a portfolio and learning project, not a commercial product, so the marketing plan has two simple goals: cover the running costs of the game, and reach the kind of player who would really enjoy it. The audience is very specific. These are solo and casual players who are not competitive and do not need to play with friends. For that reason, the plan skips the usual tactics aimed at hardcore or social audiences and focuses on simple discovery instead: a person sees the game in under a minute, learns its name, and decides to give it a try.

8.1 Monetization

All the money side of the game is optional and never makes the game easier to play. Nothing can be bought to gain an advantage, and all the income is meant to cover the running costs and help fund a future project. There are four ways the game can earn money, three of them already working and one still being studied:

- **Developer product donations (VIP menu).** Every game has a VIP menu with several donation tiers (from 5 up to 1000 Robux). These are pure donations: the player gets nothing that changes the gameplay, they simply support the project if they want to. This is already working in all ten games.
- **VIP rooms (single purchase).** A VIP area inside the map, sold as a single purchase instead of a monthly subscription. Once a player buys it, they keep access forever. This gives supporters a special space without splitting the players apart or locking any game behind a paywall.
- **Private server rental.** Players can rent a private server through the system that Roblox already provides. This fits the private style of the game's multiplayer (rooms protected by a PIN), giving small groups their own space to play together.
- **Immersive advertising (being studied).** Roblox has an immersive ad system that places ad surfaces inside the 3D world. It is being considered as an extra source of income, but it is not active yet, because it first has to be tested to make sure it does not hurt the calm and relaxed feeling of the world.

8.2 Marketing Strategy

The marketing only starts once the game is **100% finished**. Promoting an unfinished game to a casual audience usually backfires: a new player who runs into a bug or a missing image simply leaves and does

not come back. So the plan is to wait until the experience is complete and polished, and only then push hard with short videos.

The whole strategy rests on one idea: **a person should understand the game, see its name, and want to try it, all in under one minute.** Casual and solo players do not find games through tournaments, ads, or friends telling them about it. They find them by scrolling through short videos and thinking "that looks fun, let me try it." The campaign is built entirely around that moment.

8.2.1 Channels

- **Short videos** (TikTok, YouTube Shorts, Instagram Reels): the main channel. Vertical clips under 60 seconds, each one showing a single game in action with the game name on the screen the whole time.
- **YouTube (longer videos):** a second channel for longer content, such as full gameplay and videos aimed at other developers. Every YouTube description links to the **GitHub repository**, so anyone who wants the full source code and documentation can find it. This reaches a second, smaller audience: people who are learning to make games in Roblox.

8.2.2 Content Plan

- **One video per game.** At least ten short clips, one for each game, so every machine gets its own moment. The first three seconds show the most striking part, and the name of the game stays on the screen the whole time so viewers can search for it.
- **Hook first, name always.** Every clip opens with the most fun or surprising moment, never a slow intro. The name of the game and the name of the world are always visible, because the goal is for people to remember it, not just to watch it.
- **Lean into the relaxed, playful tone.** The audience includes people who enjoy weird, charming, low pressure games. The clips should feel calm and a little playful, matching the tone of the world instead of hyping up competition.
- **Compilations.** Short montages that show several games one after another make the main selling point clear: ten games in one place, eight minutes of variety.
- **Familiar arcade styles.** Many of the games are classic arcade styles rebuilt inside a 3D world. The clips can use that familiarity, since players often already know how to play, which makes them more willing to try it.

8.2.3 Rollout Phases

1. **Before launch (teasers).** A few short clips with quick glimpses of the world and some of the games, to build a little curiosity before release.
2. **Launch.** When the game is fully finished, release the ten clips (one per game) close together across TikTok, Shorts, and Reels, plus one or two compilations.
3. **After launch.** Keep posting regularly, paying attention to which clips do best and to whatever is trending in short video, and always keeping the name of the game and the world at the center.

8.3 Why This Fits the Audience

The game is made for people who want to try a new game, enjoy a meme game, relax with something low stress, or replay classic arcade styles inside Roblox. These players are reached through discovery, not persuasion. Short videos are a perfect match: they deliver the "that looks fun" moment in seconds, they do not ask the viewer to commit to anything, and they spread on their own when a clip is charming enough. The GitHub link on YouTube adds a second layer, turning the project into both a game that people play and a learning resource that other developers can study.

Part III

Technical Implementation

9 System Architecture Overview

Before going into the details of each part, the diagram below gives a single bird's eye view of the whole project. It shows how everything connects: the player interacts with the game world, the client reads the input and draws the screen, the client and the server talk through RemoteEvents and RemoteFunctions, the server uses the shared modules for the game logic, and the modules read and write to the Roblox DataStore. The same five layers are reused by all ten games and by the shared systems.

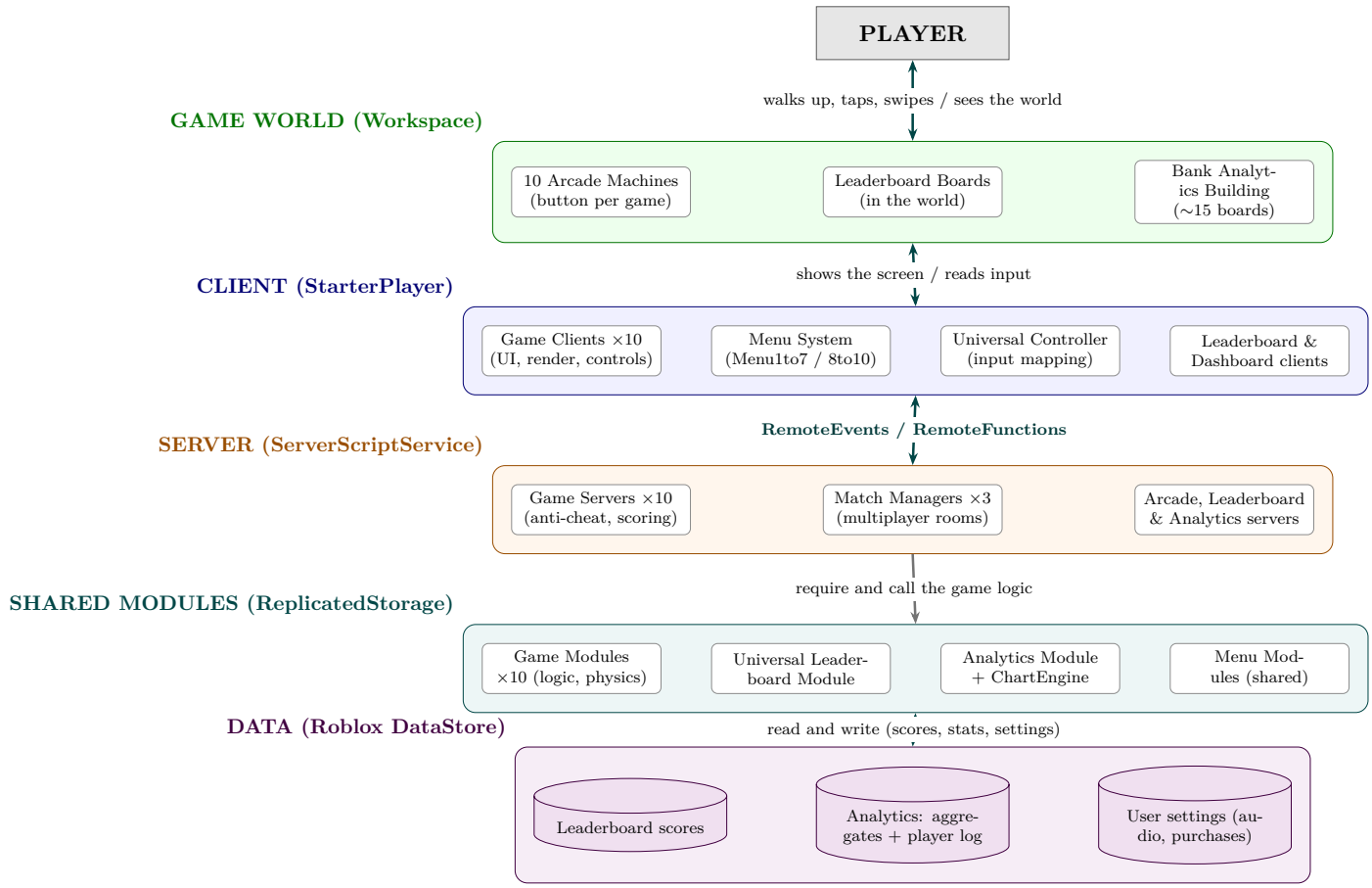


Figure 4: Bird's eye view of the system. Five layers, from the player at the top down to the DataStore, shared by the ten games, the leaderboards, the menus, the controller, and the analytics.

10 Roblox Structure

```
.
|-- Workspace
|   +-- UniversalLeaderboardPart (Part, Tag="LeadboradAvisToucan",
    Attribute="GAMEID=AvisToucan")
|   +-- UniversalLeaderboardPart (Part, Tag="LeadboradOxybelis",
    Attribute="GAMEID=Oxybelis")
|   +-- UniversalLeaderboardPart (Part, Tag="LeadboradResilio",
    Attribute="GAMEID=Resilio")
|   +-- UniversalLeaderboardPart (Part, Tag="LeadboradFoxyFruit",
    Attribute="GAMEID=FoxyFruit")
|   +-- UniversalLeaderboardPart (Part, Tag="LeadboradFirmamentDefender",
    Attribute="GAMEID=FirmamentDefender")
|   +-- UniversalLeaderboardPart (Part, Tag="LeadboradFirmamentSiege",
    Attribute="GAMEID=FirmamentSiege")
|   +-- UniversalLeaderboardPart (Part, Tag="LeadboradJuiceCutter",
    Attribute="GAMEID=JuiceCutter")
|   +-- TurrisCaseiLeaderboardPart (Score Table)
|   +-- TapAndHoldLeaderboardPart (Score Table)
|   +-- ImNotASphereLeaderboardPart (Score Table)
|   +-- Oxybelis (Folder)
|   |   +-- ButtonOxybelis (Part, Tag="ButtonOxybelis")
|   +-- AvisToucan (Folder)
|   |   +-- ButtonAvisToucan (Part, Tag="ButtonAvisToucan")
|   +-- Resilio (Folder)
|   |   +-- ButtonResilio (Part, Tag="ButtonResilio")
|   +-- FoxyFruit (Folder)
|   |   +-- ButtonFoxyFruit (Part, Tag="ButtonFoxyFruit")
|   +-- FirmamentDefender (Folder)
|   |   +-- ButtonFirmamentDefender (Part, Tag="ButtonFirmamentDefender")
|   +-- FirmamentSiege (Folder)
|   |   +-- ButtonFirmamentSiege (Part, Tag="ButtonFirmamentSiege")
|   +-- JuiceCutter (Folder)
|   |   +-- ButtonJuiceCutter (Part, Tag="ButtonJuiceCutter")
|   +-- TurrisCasei (Folder)
|   |   +-- ButtonTurrisCasei (Part, Tag="TurrisTerminal")
|   +-- TapAndHold (Folder)
|   |   +-- ButtonTapAndHold (Part, Tag="TapAndHoldTerminal")
|   +-- ImNotASphere (Folder)
|   |   +-- ButtonImNotASphere (Part, Tag="ImNotASphereTerminal")
|   +-- BankBuilding (Model, themed data analytics area)
|       +-- AnalyticsBoard x10 (Part, Tag="AnalyticsBoard",
    Attribute="GameId=<game>")
|       +-- AnalyticsBoard (Part, Tag="AnalyticsBoard",
    Attribute="GameId=General")
|       +-- AnalyticsBoard (Part, Tag="AnalyticsBoard",
    Attribute="GameId=Money")
|       +-- AnalyticsBoard (Part, Tag="AnalyticsBoard",
    Attribute="GameId=MoneyG")
|       +-- AnalyticsBoard (Part, Tag="AnalyticsBoard",
    Attribute="GameId=RealMoney")
|       +-- AnalyticsBoard (Part, Tag="AnalyticsBoard",
    Attribute="GameId=RealPlayer")
```

```
|       +-- AnalyticsBoard (Part, Tag="AnalyticsBoard",
   Attribute="GameId=LastMessage")
|-- ReplicatedStorage
|   +-- ArcadeRemotes
|   |   +-- OpenArcadeMenu (RemoteEvent)
|   |   +-- GlobalBoard (Folder)
|   |   |   +-- UpdateUniversalLeaderboard (RemoteEvent)
|   |   +-- AvisToucan (Folder)
|   |   |   +-- StartAvisToucanRequest (RemoteEvent)
|   |   |   +-- SubmitFlap (RemoteEvent)
|   |   |   +-- GameState (RemoteEvent)
|   |   |   +-- ToggleAudio (RemoteEvent)
|   |   +-- Oxybelis (Folder)
|   |   |   +-- StartOxybelisRequest (RemoteEvent)
|   |   |   +-- SubmitMove (RemoteEvent)
|   |   |   +-- GameState (RemoteEvent)
|   |   |   +-- ToggleAudio (RemoteEvent)
|   |   +-- Resilio
|   |   |   +-- StartResilioRequest (RemoteEvent)
|   |   |   +-- SubmitMoveInput (RemoteEvent)
|   |   |   +-- SubmitLaunch (RemoteEvent)
|   |   |   +-- GameState (RemoteEvent)
|   |   |   +-- ToggleAudio (RemoteEvent)
|   |   +-- FoxyFruit
|   |   |   +-- StartFoxyFruitRequest (RemoteEvent)
|   |   |   +-- SubmitMove (RemoteEvent)
|   |   |   +-- GameState (RemoteEvent)
|   |   |   +-- ToggleAudio (RemoteEvent)
|   |   +-- FirmamentDefender
|   |   |   +-- StartFirmamentDefenderRequest (RemoteEvent)
|   |   |   +-- SubmitMove (RemoteEvent)
|   |   |   +-- GameState (RemoteEvent)
|   |   |   +-- ToggleAudio (RemoteEvent)
|   |   +-- FirmamentSiege
|   |   |   +-- StartFirmamentSiegeRequest (RemoteEvent)
|   |   |   +-- SubmitMove (RemoteEvent)
|   |   |   +-- GameState (RemoteEvent)
|   |   |   +-- ToggleAudio (RemoteEvent)
|   |   +-- JuiceCutter
|   |   |   +-- StartJuiceCutterRequest (RemoteEvent)
|   |   |   +-- SubmitMove (RemoteEvent)
|   |   |   +-- GameState (RemoteEvent)
|   |   |   +-- ToggleAudio (RemoteEvent)
|   |   +-- TurrisCasei
|   |   |   +-- StartTurrisCaseiRequest (RemoteEvent)
|   |   |   +-- MatchStatus (RemoteEvent)
|   |   |   +-- ToggleAudio (RemoteEvent)
|   |   |   +-- UpdateTurrisCaseiLeaderboard (RemoteEvent)
|   |   |   +-- GetAudioState (RemoteFunction)
|   |   |   +-- LobbySystem (Folder)
|   |   |       +-- CreateRoom (RemoteFunction)
|   |   |       +-- JoinRoom (RemoteFunction)
|   |   |       +-- StartMultiplayerMatch (RemoteEvent)
```

```
|   |   |       +-- RoomUpdate (RemoteEvent)
|   |   |       +-- KickPlayer (RemoteEvent)
|   |   |       +-- LeaveRoom (RemoteEvent)
|   |   |       +-- UpdateLiveScore (RemoteEvent)
|   |   |       +-- SpectateTarget (RemoteEvent)
|   |   |       +-- PlayerDied (RemoteEvent)
|   |   +-- TapAndHold
|   |   |   +-- StartTapAndHoldRequest (RemoteEvent)
|   |   |   +-- SubmitJumpCharge (RemoteEvent)
|   |   |   +-- MatchStatus (RemoteEvent)
|   |   |   +-- ToggleAudio (RemoteEvent)
|   |   |   +-- UpdateTapAndHoldLeaderboard (RemoteEvent)
|   |   |   +-- GetAudioState (RemoteFunction)
|   |   |   +-- LobbySystem (Folder)
|   |   |       +-- CreateRoom (RemoteFunction)
|   |   |       +-- JoinRoom (RemoteFunction)
|   |   |       +-- StartMultiplayerMatch (RemoteEvent)
|   |   |       +-- RoomUpdate (RemoteEvent)
|   |   |       +-- KickPlayer (RemoteEvent)
|   |   |       +-- LeaveRoom (RemoteEvent)
|   |   |       +-- UpdateLiveScore (RemoteEvent)
|   |   |       +-- SpectateTarget (RemoteEvent)
|   |   |       +-- PlayerDied (RemoteEvent)
|   |   +-- ImNotASphere
|   |       +-- StartImNotASphereRequest (RemoteEvent)
|   |       +-- MatchStatus (RemoteEvent)
|   |       +-- ToggleAudio (RemoteEvent)
|   |       +-- UpdateImNotASphereLeaderboard (RemoteEvent)
|   |       +-- GetAudioState (RemoteFunction)
|   |       +-- LobbySystem (Folder)
|   |           +-- CreateRoom (RemoteFunction)
|   |           +-- JoinRoom (RemoteFunction)
|   |           +-- StartMultiplayerMatch (RemoteEvent)
|   |           +-- RoomUpdate (RemoteEvent)
|   |           +-- KickPlayer (RemoteEvent)
|   |           +-- LeaveRoom (RemoteEvent)
|   |           +-- UpdateLiveScore (RemoteEvent)
|   |           +-- SpectateTarget (RemoteEvent)
|   |           +-- PlayerDied (RemoteEvent)
|   +-- AnalyticsRemotes (Folder)
|   |   +-- GetGameDashboard (RemoteFunction)
|   |   +-- GetGeneralDashboard (RemoteFunction)
|   |   +-- GetMoneyDashboard (RemoteFunction)
|   |   +-- GetRecentDonations (RemoteFunction)
|   |   +-- GetRecentScores (RemoteFunction)
|   |   +-- DashboardUpdated (RemoteEvent)
|   +-- Menu (Folder)
|   |   +-- Menu1to7 (ModuleScript)
|   |   +-- Menu8to10 (ModuleScript)
|   +-- TurrisCasei_Assets (Folder)
|   |   +-- BaseP1 (Part/Model)
|   |   +-- QuesoBlock (Part)
|   |   +-- RoomModel (Model)
```

```
|   +-- TapAndHold_Assets (Folder)
|   |   +-- tapsuelo1..tapsuelo4 (Platform)
|   |   +-- Model (Model floor)
|   +-- ImNotASphere_Assets (Folder)
|       +-- PlayerSphere (Part player)
|       +-- Level1 (Model)
|       |   +-- Geometry (Folder - Optional)
|       |   |   +-- Suelo1, Rampa2, Pared3 (Parts)
|       |   +-- MovingPlatforms (Folder)
|       |   |   +-- Elevator1 (Model)
|       |   |       +-- Platform (Moving part)
|       |   |       +-- StartPart (Transparent, CanCollide=false)
|       |   |       +-- EndPart (Transparent, CanCollide=false)
|       |   +-- Collectibles (Folder)
|       |   |   +-- Coin (Part)
|       |   |   +-- ExtraPoint (Part)
|       |   +-- KillZones (Folder)
|       |   |   +-- DeathPit (Giant transparent part below level)
|       |   +-- SpawnPoint (Transparent part)
|       |   +-- FinishLine (Part - Goal to proceed to Level2)
|       +-- Level2, Level3 (Models)
|   +-- Modules
|       +-- UniversalLeaderboardModule (ModuleScript: Global Score logic)
|       +-- TurrisCaseiLeaderboard (score turris casei)
|       +-- TapAndHoldLeaderboard (score Tap-and-Hold)
|       +-- ImNotASphereLeaderboard (ModuleScript: Score table logic)
|       +-- AvisToucanModule (ModuleScript logic,physics,collisions,etc)
|       +-- OxybelisModule (ModuleScript logic,physics,collisions,etc)
|       +-- ResilioModule (ModuleScript logic,physics,collisions,etc)
|       +-- FoxyFruitModule (ModuleScript logic,physics,collisions,etc)
|       +-- FirmamentDefenderModule (ModuleScript logic,physics,collisions,etc)
|       +-- FirmamentSiegeModule (ModuleScript logic,physics,collisions,etc)
|       +-- JuiceCutterModule (ModuleScript logic,physics,collisions,etc)
|       +-- TurrisCaseiModule (ModuleScript math,speed,physics)
|       +-- TapAndHoldModule (ModuleScript logic,physics,parabolic math)
|       +-- ImNotASphereModule (ModuleScript: Physics, speed, powers, elevator
   math)
|       +-- AnalyticsModule (ModuleScript: session/donation tracking,
   aggregates, dashboards)
|       +-- ChartEngine (ModuleScript: draws charts on EditableImage canvases)
|-- ServerScriptService
|   +-- UniversalLeaderboardServer.lua (Script: Global Data Broadcast)
|   +-- ArcadeServer.lua (Script:binds buttons by tag,sends OpenArcadeMenu)
|   +-- AvisToucanServer.lua (Script:button and input connections)
|   +-- OxybelisServer.lua (Script:button and input connections)
|   +-- ResilioServer.lua (Script:button and input connections)
|   +-- FoxyFruitServer.lua (Script:button and input connections)
|   +-- FirmamentDefenderServer.lua (Script:button and input connections)
|   +-- FirmamentSiegeServer.lua (Script:button and input connections)
|   +-- JuiceCutterServer.lua (Script:button and input connections)
|   +-- TurrisCaseiServer.lua (Script: Anti-Cheat & Global Data)
|   +-- TurrisCaseiMatchManager.lua (ModuleScript: versus logic, rooms, winner
   selection)
```

```
|   +-- TapAndHoldServer.lua (Script: Anti-Cheat & Global Data)
|   +-- TapAndHoldMatchManager.lua (ModuleScript: versus logic, rooms, winner
    selection)
|   +-- ImNotASphereServer.lua (Script: Anti-Cheat, Power up validation &
    Global Data)
|   +-- ImNotASphereMatchManager.lua (ModuleScript: versus logic, rooms, winner
    selection)
|   +-- AnalyticsServer.lua (Script: analytics remotes, rate limiting, refresh
    broadcast)
|   +-- SunsetAtmosphere.lua (Script: world lighting and ambiance)
|   +-- GodRays.lua (Script: sun ray visual effect)
|-- StarterPlayer
|   +-- StarterPlayerScripts
|       +-- UniversalController.lua (ModuleScript: Mapping for mobile, pc,
    xbox, playstation)
|       +-- UniversalLeaderboardClient.lua (LocalScript: Multi-game dynamic UI
    renderer)
|       +-- AvisToucanClient.lua (LocalScript: arcade UI/menu, render, controls)
|       +-- OxybelisClient.lua (LocalScript: arcade UI/menu, render, controls)
|       +-- ResilioClient.lua (LocalScript: arcade UI/menu, render, controls)
|       +-- FoxyFruitClient.lua (LocalScript: arcade UI/menu, render, controls)
|       +-- FirmamentDefenderClient.lua (LocalScript: arcade UI/menu, render,
    controls)
|       +-- FirmamentSiegeClient.lua (LocalScript: arcade UI/menu, render,
    controls)
|       +-- JuiceCutterClient.lua (LocalScript: arcade UI/menu, render,
    controls)
|       +-- TurrisCaseiClient.lua (module: Visual Game, HUD, Spectator)
|       +-- TurrisCaseiLeaderboardClient (localscript: ux score table)
|       +-- TapAndHoldClient.lua (LocalScript: Visual Game, Input Holding, HUD,
    Spectator)
|       +-- TapAndHoldLeaderboardClient (LocalScript: ux score table)
|       +-- ImNotASphereClient.lua (LocalScript: Visual Game, Controls, Camera,
    HUD)
|       +-- ImNotASphereLeaderboardClient.lua (LocalScript: UX score table)
|       +-- AnalyticsDashboardClient.lua (LocalScript: renders the analytics
    boards)
|       +-- BlobShadows.lua (LocalScript: soft ground shadows)
```

Listing 1: Current project structure in Roblox (Updated with Avis Toucan, Oxybelis, Resilio, Foxy Fruit, Firmament Defender, Firmament Siege, Juice Cutter, Universal Leaderboard, and the Data Analytics System).

11 Coding in Roblox by Game Type

All the games follow the same pattern of communication between the client and the server, even though each one shows something different on the screen. The client never decides the result of the game; it only draws what is happening and sends the player's input to the server. The server runs the real game logic in a module, and when the run ends it reports the score to the leaderboard, which saves it to the DataStore. The sequence diagram below follows a normal single player session from start to game over, so the per game flowcharts that follow can focus on the rules instead of the messages.

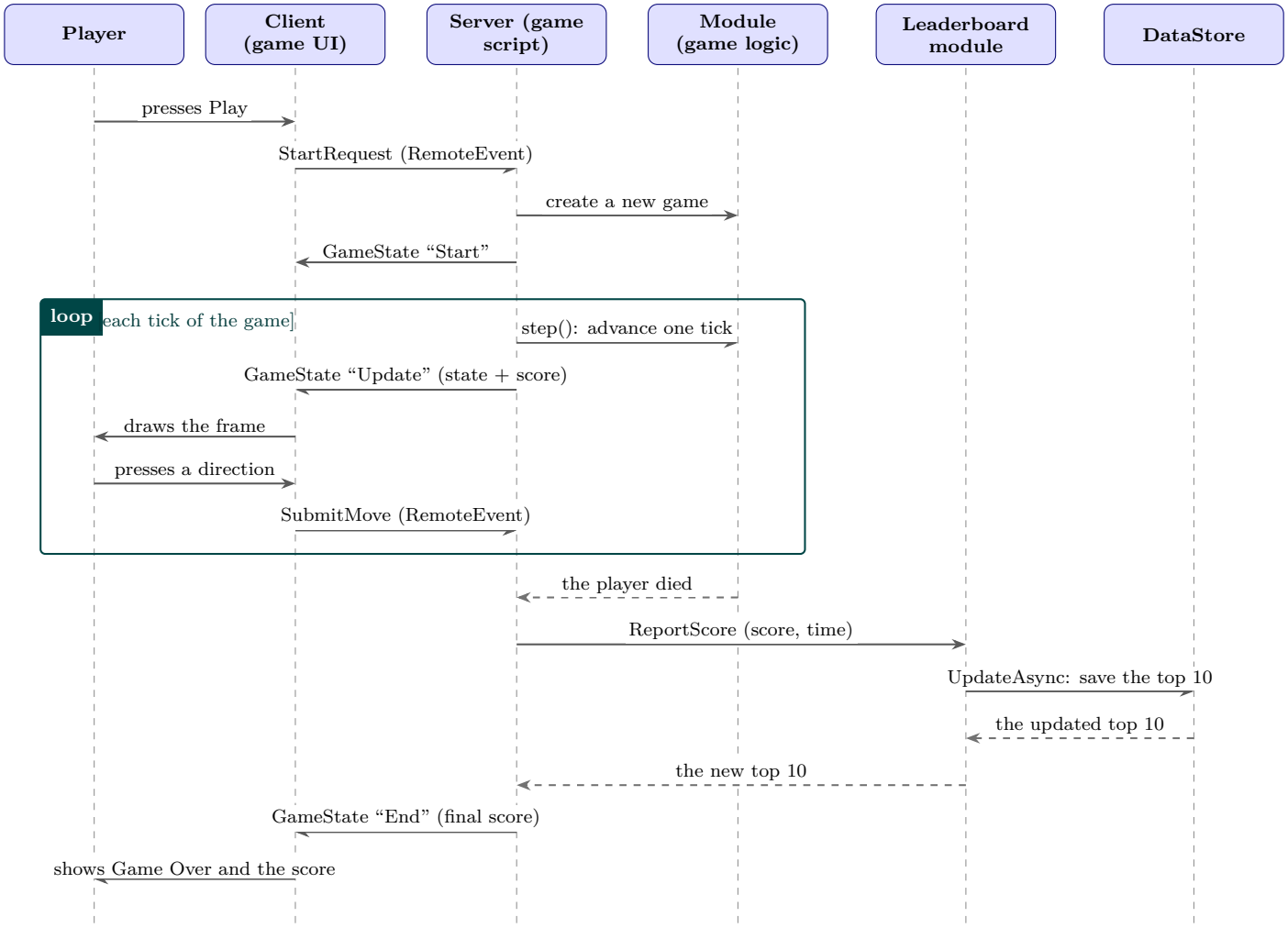


Figure 5: Sequence diagram of a normal game session, showing the back and forth between the player, the client, the server, the game module, the leaderboard, and the DataStore.

Inside the "Playing" part of that flow, each round also moves through a small set of internal states. The flowcharts in each game focus on the rules, so the state machine below shows the technical side: the real states the code uses to manage a round, from the first countdown to game over, including losing a life and moving to the next level.

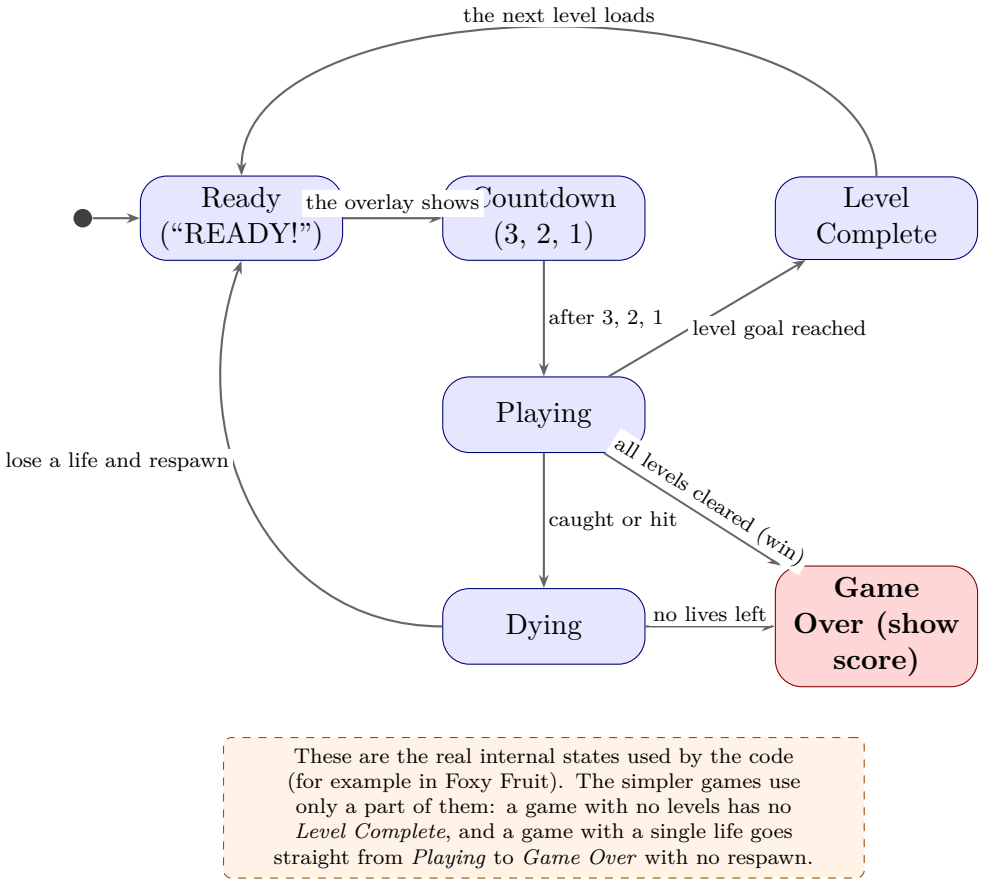


Figure 6: Internal state machine of a single round. These are the real states from the code; simpler games use only a part of them.

11.1 Oxybelis

Oxybelis is a grid based arcade game built from scratch inside Roblox Studio. The idea is simple: guide a growing creature around a closed grid without crashing into the walls or into its own body. The project took five days. The first two days went into learning Roblox Studio and getting the movement to feel right, and the last three went into the global leaderboard, which turned out to be the hardest part. It was hard not because of the sorting, but because of the DataStore rate limits and making sure no score was ever lost if the server crashed.

11.1.1 Rules of Oxybelis

- **Main Goal:** Eat as much food as possible without crashing.
- **Controls (PC):** Arrow keys or WASD for direction. The game is also adapted for mobile touchscreen, Xbox, and PlayStation controllers through `UniversalController`.
- **Core Mechanics:**
  - The creature moves on its own at a fixed speed; the player only chooses the direction.
  - Food appears on a random free cell. Eating it makes the body one cell longer and adds one to the score.
- **End Conditions:** The game ends when the creature hits a wall or its own body. A perfect game happens when the body fills the whole 16×12 grid.

Flowchart

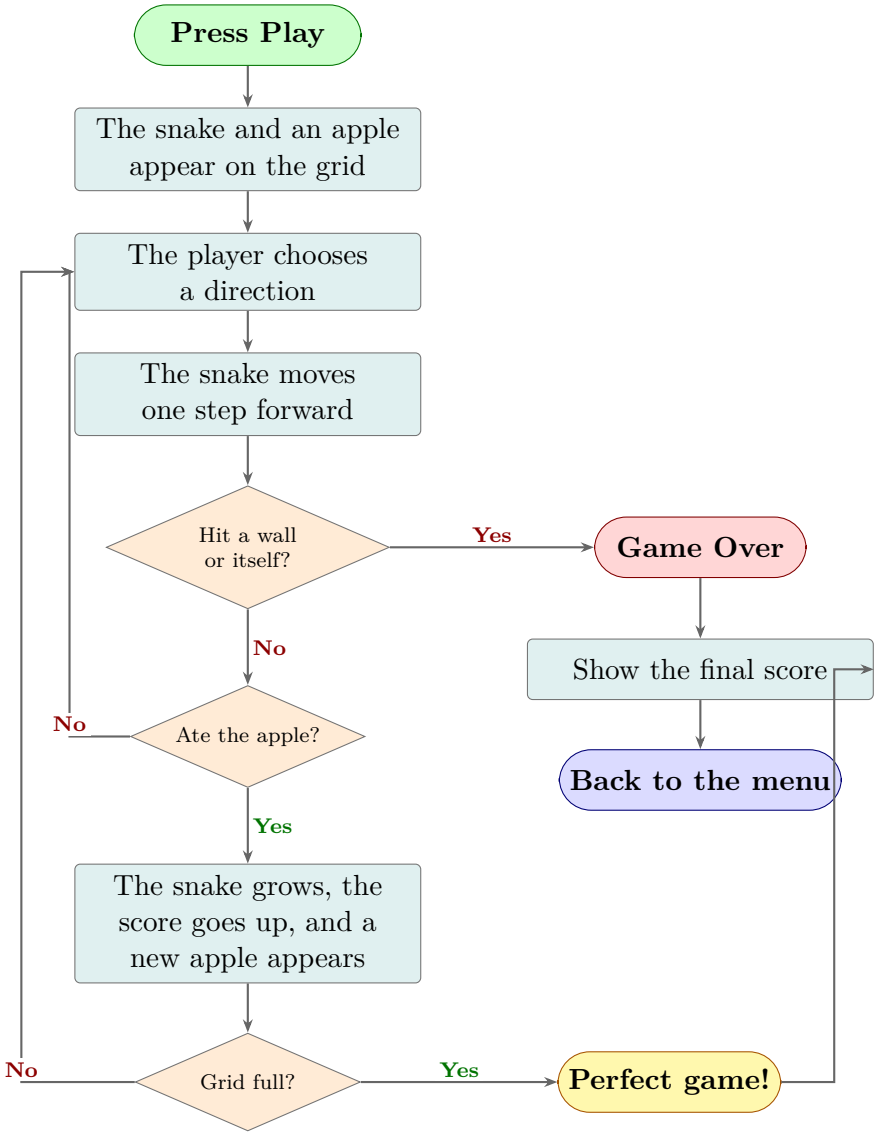


Figure 7: Flowchart for Oxybelis.

11.1.2 Project Structure

Oxybelis is split across four files that follow the way Roblox separates the client and the server. Each file has one clear job.

```
Oxybelis/
|-- Workspace/
|   +-- Oxybelis/ (Folder)
|       +-- ButtonOxybelis      (Part, Tag="ButtonOxybelis")
|-- ReplicatedStorage/
|   |-- ArcadeRemotes/
|   |   +-- Oxybelis/ (Folder)
|   |       |-- StartRequest    (RemoteEvent)
|   |       |-- SubmitMove      (RemoteEvent)
|   |       |-- GameState       (RemoteEvent)
|   |       +-- ToggleAudio     (RemoteEvent)
|   +-- Modules/
|       +-- OxybelisModule      (ModuleScript)
|-- ServerScriptService/
```

```
|   +-- OxybelisServer          (Script)
+-- StarterPlayer/
    +-- StarterPlayerScripts/
        +-- OxybelisClient      (LocalScript)
```

11.1.3 Time Complexity

Each game tick is very cheap to run, with a cost of $O(1)$. The `step()` function moves the creature by shifting two pointers in a circular buffer, checks the walls with simple number comparisons, and checks whether the head touched the body by looking it up in a table. All of these take the same small amount of time, no matter how long the creature is. Placing new food is also $O(1)$, because the game keeps a list of free cells and picks one at random instead of scanning the whole grid. The only part that grows with the creature is `getOxybelis()`, which copies the body into a list to send to the screen; this costs $O(L)$, where L is the current length. Since the body can never be longer than the grid ($16 \times 12 = 192$ cells), even this stays small. In short, the game stays fast no matter how big the creature gets.

11.1.4 Space Complexity

The memory the game uses stays fixed at $O(N)$, where $N = 192$ is the number of cells. Four structures are created when the game starts and never grow: the circular buffer that holds the body, a table that marks which cells are taken, and two lists that keep track of the free cells. Nothing new is created while playing. A cell simply goes back to the free list when the tail moves, and is taken again when the body grows. Because of this, the memory stays the same from the first second to the last, no matter the score or the length of the creature.

11.1.5 Game Features and Implementation Status

Feature	Description	Status
Main Menu	Interface with Play, VIP, and Exit buttons. Supports navigation via A, S, arrow keys, and selection with Space/Enter.	Working
Menu Sounds	UI elements include audio feedback for navigation and selection.	Working
Gameplay Movement	Responsive control system using WASD or arrow keys.	Working
Mute Button	In game toggle to turn all audio on or off.	Working
Exit Button (Game)	Allows the player to abort the run and return to the score menu.	Working
Score Menu	Post game screen displaying the final score and options to replay or exit.	Working
Top 10 Score Table	Displays the top 10 players globally. Uses Roblox DataStore for persistent online storage.	Working
VIP Donation Menu	Opens a menu with various donation tiers using in game currency.	Working
Optimization	Circular buffer implementation ensures $O(1)$ logic updates.	Working
Mobile Adaptation	Touchscreen controls and UI scaling via UniversalController.	Working
Xbox Adaptation	Gamepad support and console UI via UniversalController.	Working
PlayStation Adaptation	DualSense and DualShock input support via UniversalController.	Working

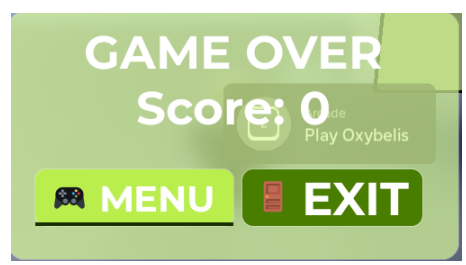
Table 9: Functionalities and current status of Oxybelis.

11.1.6 Game Results

The following images illustrate the final implementation, including the user interface, gameplay view, and score summary.

Main Menu: Features animated buttons and sound effects.

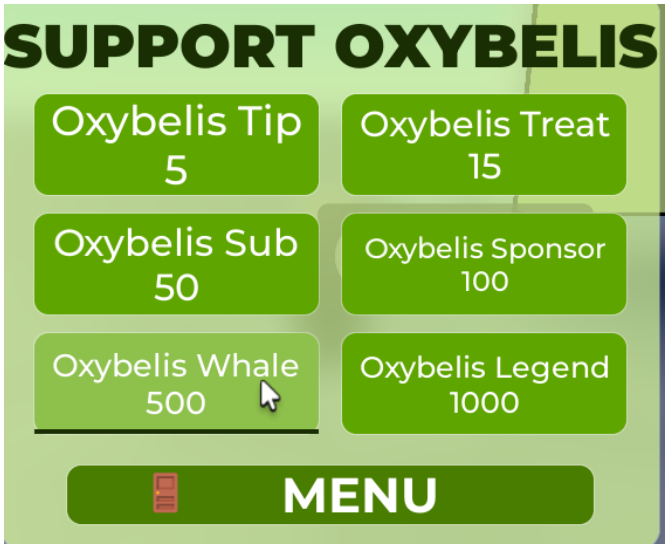
Gameplay: The active game state with HUD elements.



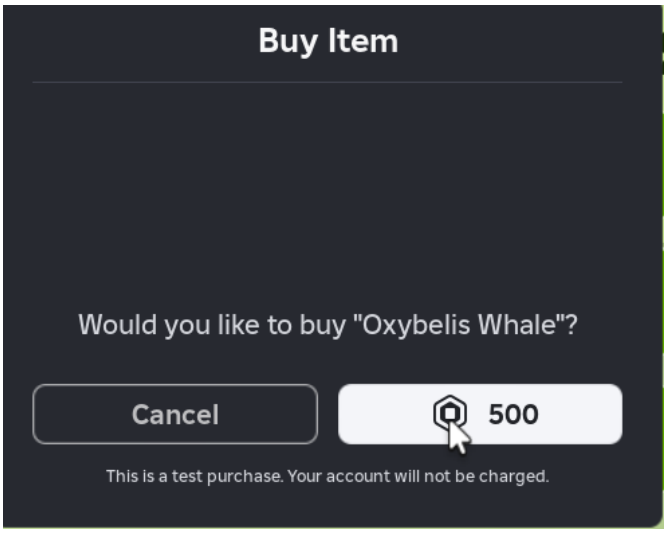
Score Menu: Post game summary and navigation.

11.1.7 VIP and Global Scores

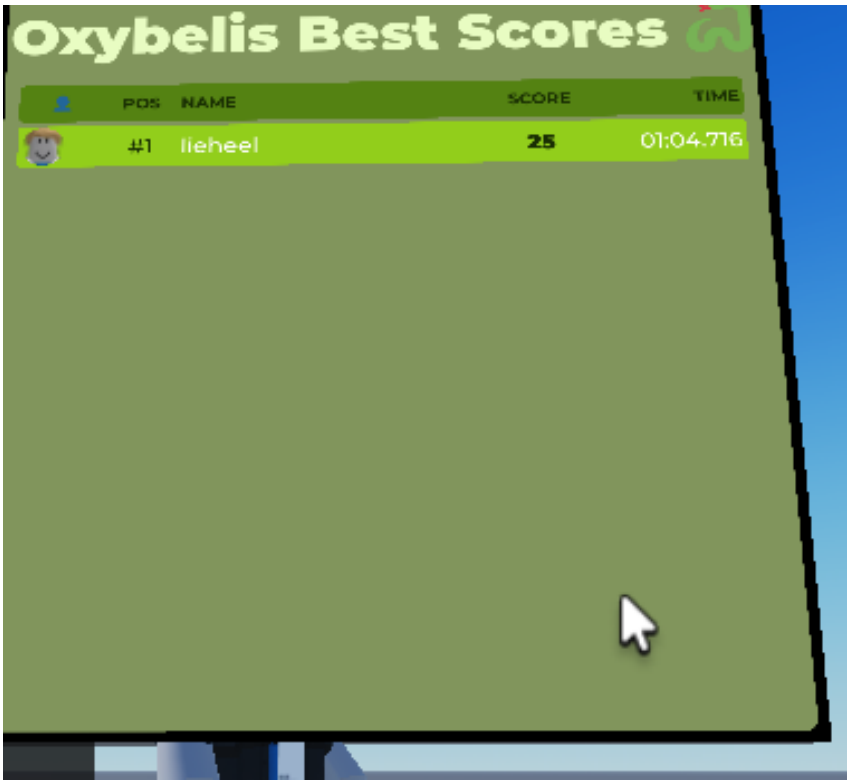
The project includes a monetization test via a VIP donation system and a competitive leaderboard managed via the cloud.



VIP Section: Donation menu interface.



Donation Process: Transaction confirmation flow.



Global Score Table: Displays the top 10 players. Ties are resolved by comparing completion time.

11.2 Avis Toucan

Avis Toucan is a side scrolling arcade game built inside Roblox Studio. The goal is simple: keep the bird in the air as long as possible by tapping at the right moments to pass through the gaps between the pipes. The project took four days. The logic itself was not the hard part, since gravity, collision, and spawning pipes are each only a few lines. The real challenge was the game feel: adjusting the gravity ($900\,\mathrm{px/s^2}$) and the flap force ($-320\,\mathrm{px/s}$) until the difficulty felt fair without being frustrating, especially for a younger audience.

11.2.1 Rules of Avis Toucan

- **Main Goal:** Navigate through pipe gaps to achieve the highest score.
- **Controls (PC):** Click or press Space to flap. Also adapted for mobile touchscreen, Xbox, and PlayStation controllers via `UniversalController`.
- **Core Mechanics:** The bird is subject to continuous gravity ($900\,\mathrm{px/s^2}$). Each flap applies an upward velocity impulse of $-320\,\mathrm{px/s}$. Pipes spawn every $1.8\,\mathrm{s}$ and move left at $117\,\mathrm{px/s}$.
- **Scoring:** One point for every pipe pair cleared.
- **End Condition:** Hitting a pipe or the ground ends the session immediately.

Flowchart

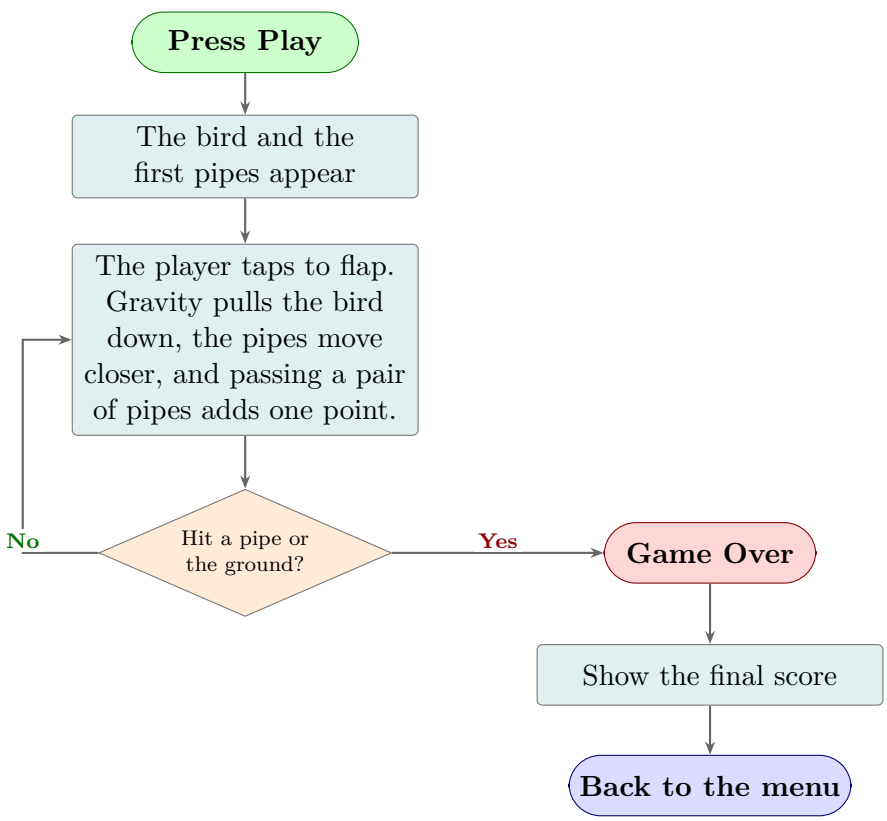


Figure 8: Flowchart for Avis Toucan.

11.2.2 Project Structure

Avis Toucan follows the same four file layout as the rest of the arcade. Each component has one clear responsibility and no cross game dependencies.

```
AvisToucan/
|-- Workspace/
|   +-- AvisToucan/ (Folder)
|       +-- ButtonAvisToucan   (Part, Tag="ButtonAvisToucan")
|-- ReplicatedStorage/
|   |-- ArcadeRemotes/
|   |   +-- AvisToucan/ (Folder)
|   |       |-- StartAvisToucanRequest  (RemoteEvent)
|   |       |-- SubmitFlap              (RemoteEvent)
|   |       |-- GameState               (RemoteEvent)
|   |       +-- ToggleAudio             (RemoteEvent)
|   +-- Modules/
|       +-- AvisToucanModule   (ModuleScript)
|-- ServerScriptService/
|   +-- AvisToucanServer       (Script)
+-- StarterPlayer/
    +-- StarterPlayerScripts/
        +-- AvisToucanClient   (LocalScript)
```

11.2.3 Time Complexity

Each call to `step(dt)` costs $O(P)$ time, where P is the number of pipes on screen. Moving the bird and applying gravity are $O(1)$. Moving the pipes, removing the ones that have left the screen, and checking for collisions all go through the list of pipes, which is what gives the $O(P)$. The key point is that there are never many pipes at once: since a new pipe appears every 1.8 seconds and they move at 117 pixels per second across a screen 480 pixels wide, at most 3 pipes can be visible at the same time. With P always 3 or fewer, the real cost per tick is basically $O(1)$. Sending the game state to the screen is also $O(P)$ and happens every 50 milliseconds.

11.2.4 Space Complexity

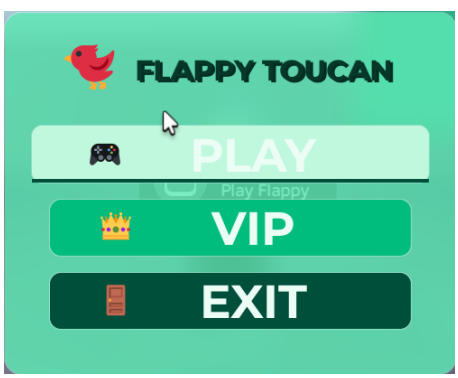
The memory cost is $O(P)$, where P is at most 3 pipes. The only thing that grows while playing is the list of pipes, and a pipe is removed as soon as it leaves the left edge, so the list never piles up. Everything else (the bird's position and speed, the score, the spawn timer) is a single value. In practice, the memory used in a session stays constant.

11.2.5 Game Features and Implementation Status

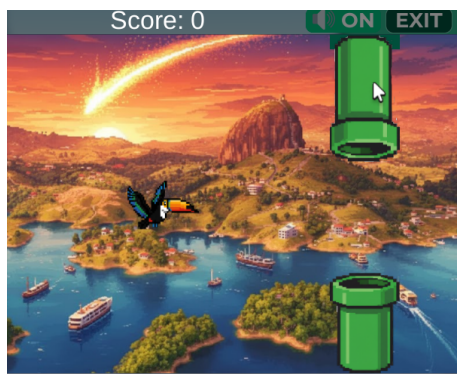
Feature	Description	Status
Main Menu	Interface with Play, VIP, and Exit buttons.	Working
Menu Sounds	Audio feedback for navigation and selection.	Working
Gameplay Control	One button control (click, tap, or Space).	Working
Mute Button	In game toggle to turn all audio on or off.	Working
Exit Button (Game)	Allows the player to abort the run and return to the score menu.	Working
Score Menu	A screen after the game showing the final score and a replay option.	Working
Top 10 Score Table	Global leaderboard via Roblox DataStore.	Working
VIP Donation Menu	Opens a menu with various donation tiers.	Working
Optimization	dt-based physics integration; pipe count bounded at $P \leq 3$, giving $O(1)$ practical cost per tick.	Working
Mobile Adaptation	Touchscreen tap input via `UniversalController`.	Working
Xbox Adaptation	Gamepad button input via `UniversalController`.	Working
PlayStation Adaptation	DualSense and DualShock input via `UniversalController`.	Working

Table 10: Functionalities and current status of Avis Toucan.

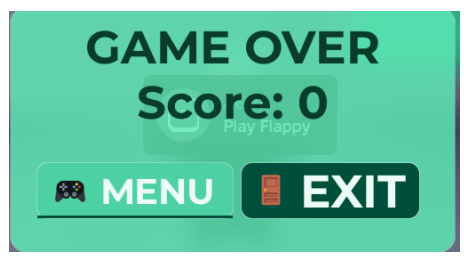
11.2.6 Game Results



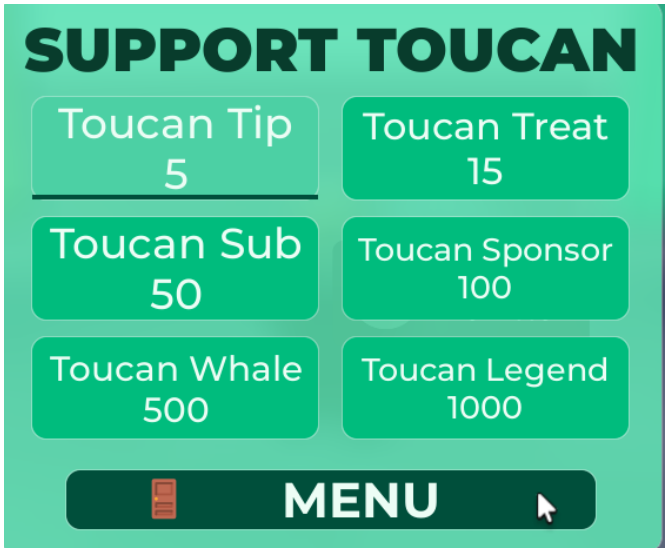
Main Menu



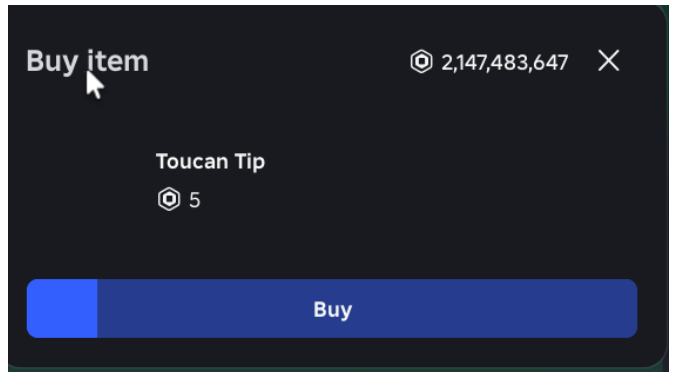
Gameplay



Score Menu



VIP Menu



Donation System

Global Score Table: Top 10 ranking.

11.3 Resilio

Resilio is a ball and paddle arcade game built inside Roblox Studio. It had two main challenges. The first was the physics: the ball bounces off the paddle at an angle that depends on where it lands, it collides with the bricks, and some bricks need more than one hit to break. The second was the system of levels, with five handmade layouts that the player clears one by one. The game also adds an auto fire power up, dropped by a special brick, that gives the paddle laser shots for four seconds and brings a bit of strategy to the usual gameplay.

11.3.1 Rules of Resilio

- **Main Goal:** Destroy all bricks across five levels without losing all three lives.
- **Controls (PC):** Arrow keys or A/D to move the paddle, Space to launch the ball. Also adapted for mobile touchscreen, Xbox, and PlayStation controllers via `UniversalController`.
- **Mechanics:** The ball reflects off walls and the paddle. The deflection angle is calculated from the impact position on the paddle, clamped to ±60ř. Ball speed increases by 10 px/s on each paddle hit, up to a maximum of 250 px/s.
- **Bricks:** Three brick types (normal, medium, hard). At level load, 5 to 10 random bricks are assigned 2 to 4 HP. One of them is the auto fire brick worth 50 points.
- **Auto fire:** Destroying the special brick activates 4 seconds of laser fire from the paddle, targeting the nearest brick every 125 ms.
- **Lives:** The session starts with 3 lives. Clearing a level grants 1 extra life.
- **End Condition:** Losing all lives ends the session. Clearing all five levels triggers a perfect game.

Flowchart

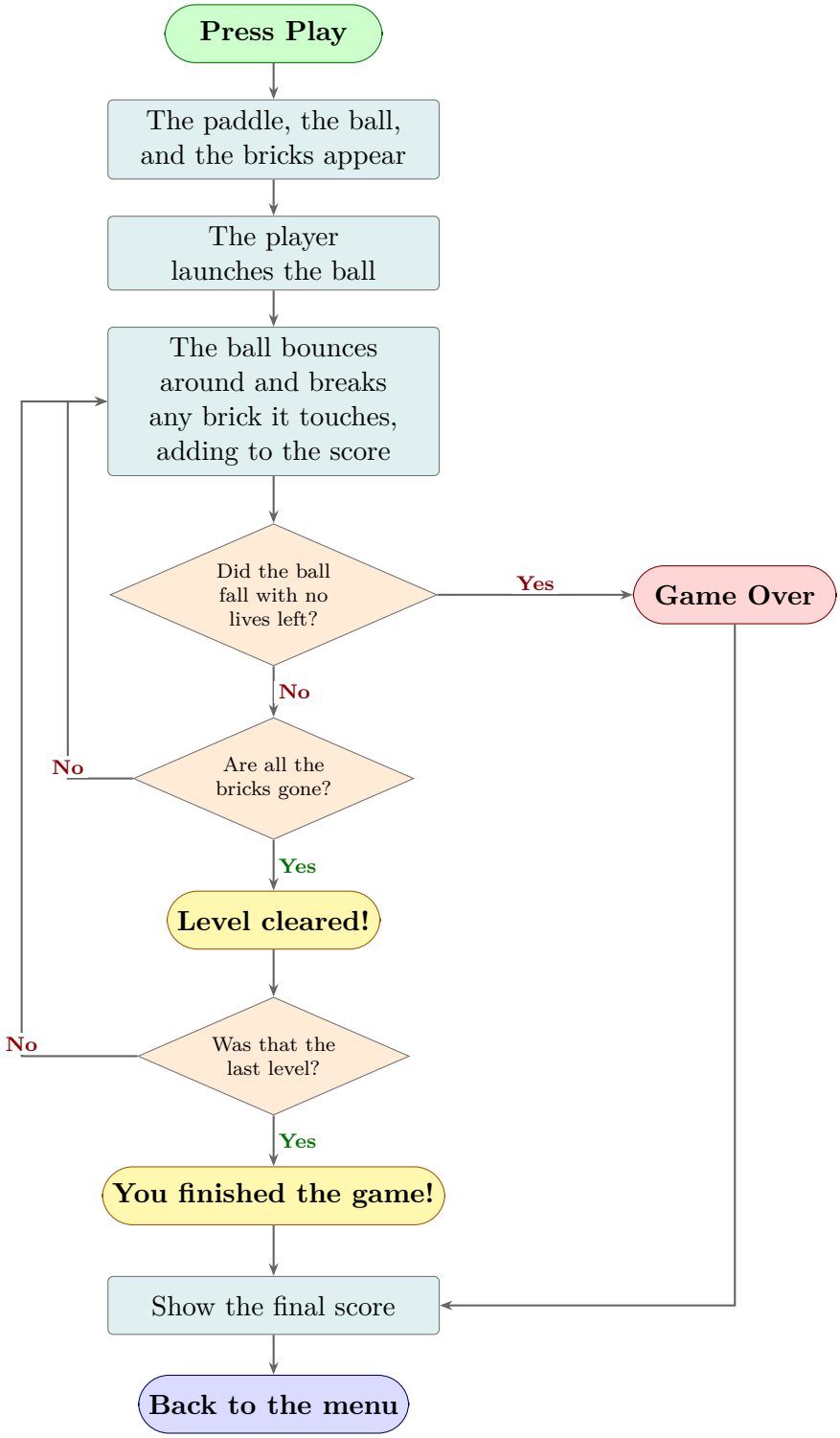


Figure 9: Flowchart for Resilio.

11.3.2 Project Structure

Resilio uses the same four file layout as the rest of the arcade.

```
Resilio/
|-- Workspace/
|   +-- Resilio/ (Folder)
|       +-- ButtonResilio          (Part, Tag="ButtonResilio")
```

```
|-- ReplicatedStorage/
|   |-- ArcadeRemotes/
|   |   +-- Resilio/ (Folder)
|   |       |-- StartResilioRequest  (RemoteEvent)
|   |       |-- SubmitMoveInput      (RemoteEvent)
|   |       |-- SubmitLaunch         (RemoteEvent)
|   |       |-- GameState            (RemoteEvent)
|   |       +-- ToggleAudio          (RemoteEvent)
|   +-- Modules/
|       +-- ResilioModule          (ModuleScript)
|-- ServerScriptService/
|   +-- ResilioServer              (Script)
+-- StarterPlayer/
    +-- StarterPlayerScripts/
        +-- ResilioClient          (LocalScript)
```

11.3.3 Time Complexity

Each call to `step(dt)` costs $O(B)$ time, where B is the number of bricks still standing. Moving the paddle is $O(1)$, and bouncing the ball off the walls and the paddle is $O(1)$ as well. Checking the ball against the bricks goes through the brick list once, which gives the $O(B)$, and the auto fire also looks through the bricks to find the nearest one, again $O(B)$. At the start of a level there are at most 120 bricks (12 columns and up to 10 rows), and that number only goes down as the player breaks them, so the game gets cheaper as the level goes on. Sending the state to the screen copies the live bricks in $O(B)$ and the active laser shots, which are never more than 2 at a time.

11.3.4 Space Complexity

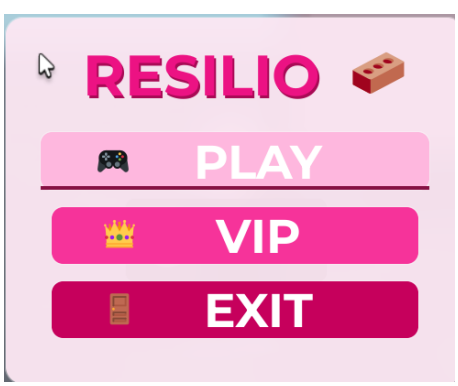
The memory cost is $O(B)$, where B is at most 120 bricks per level. Each brick keeps its position, size, hit points, type, and a flag for the special brick. When a brick is broken it is simply marked as not alive and left out of the state sent to the screen; the game keeps a separate counter that reaches zero when the level is cleared. The laser shots are kept in a small list that never holds more than 2 at once. Everything else (the paddle, the ball, the lives, the score, the current level) is a single value.

11.3.5 Game Features and Implementation Status

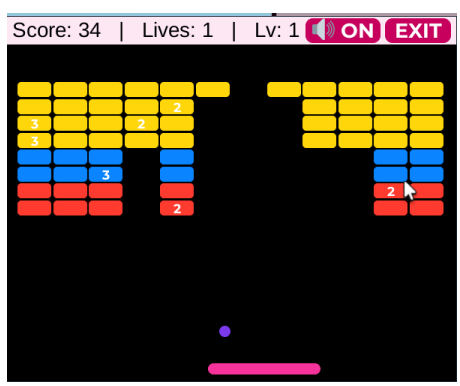
Feature	Description	Status
Main Menu	Interface with Play, VIP, and Exit buttons.	Working
Menu Sounds	Audio feedback for navigation and selection.	Working
Gameplay Controls	Paddle movement (left/right) and ball launch.	Working
Mute Button	In game toggle to turn all audio on or off.	Working
Exit Button (Game)	Allows the player to abort the run and return to the score menu.	Working
Score Menu	A screen after the game showing the final score and a replay option.	Working
Level System	Five built in layouts with increasing complexity. Clearing a level grants 1 extra life.	Working
Multi HP Brick System	Bricks have 1 to 4 HP. Tough bricks are assigned randomly at level load.	Working
Auto fire Power up	Destroying the special brick activates 4 s of laser shots from the paddle, targeting the nearest brick every 125 ms.	Working
Top 10 Score Table	Global leaderboard via Roblox DataStore.	Working
VIP Donation Menu	Opens a menu with various donation tiers.	Working
Optimization	$O(B)$ collision scan with B decreasing per level; $S \leq 2$ active shots at any instant.	Working
Mobile Adaptation	Touchscreen input via `UniversalController`.	Working
Xbox Adaptation	Gamepad input via `UniversalController`.	Working
PlayStation Adaptation	DualSense and DualShock input via `UniversalController`.	Working

Table 11: Functionalities and current status of Resilio.

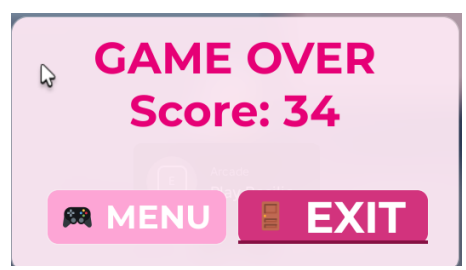
11.3.6 Game Results



Main Menu

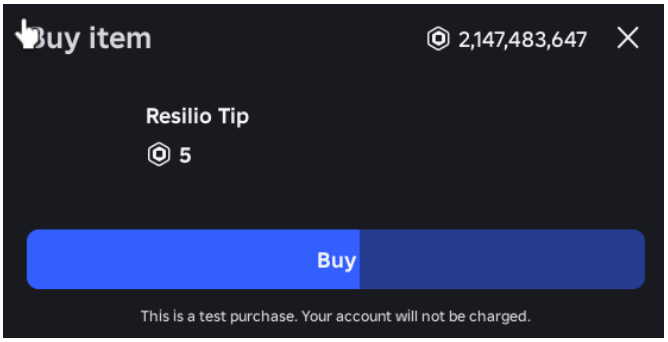


Gameplay



Score Menu

VIP Menu



Donation Interface

Global Score Table

11.4 Foxy Fruit

Foxy Fruit is a maze based action game built inside Roblox Studio on a grid of 16×12 tiles. The hardest part to build was the bear AI. The four bears all use the same pathfinding (a breadth first search, or BFS), but each one aims at a different cell, so they end up behaving in different and hard to predict ways without having to script every single situation by hand. The game has three handmade levels and a power mode: when the player eats a blueberry, all the bears turn scared for a few seconds.

11.4.1 Rules of Foxy Fruit

- **Main Goal:** Eat all raspberries across three levels without losing all three lives.
- **Controls (PC):** Arrow keys or WASD for grid based movement. Also adapted for mobile touchscreen, Xbox, and PlayStation controllers via `UniversalController`.
- **Collectibles:** Raspberries score 10 points each. Blueberries score 50 points and activate power mode for 4 seconds.
- **Power Mode:** All bears switch to frightened state and flee. Eating a frightened bear scores 200, 400, 800, or 1600 points depending on how many are eaten in one power mode activation.
- **Bear Behaviors:**
  - **Red Bear:** Uses BFS to move directly toward the player's current cell.
  - **Pink Bear:** Uses BFS to target a cell 4 steps ahead of the player's current direction.
  - **Blue Bear:** Calculates a point 2 cells ahead of the player, then mirrors the red bear's position relative to that point and targets the reflected cell.
  - **Orange Bear:** Chases the player directly (BFS) if more than 8 cells away; retreats to the bottom left corner when closer.
- **Lives:** The session starts with 3 lives. Clearing a level grants 1 extra life.
- **End Condition:** Losing all lives ends the session. Clearing all three levels triggers a perfect game.

Flowchart

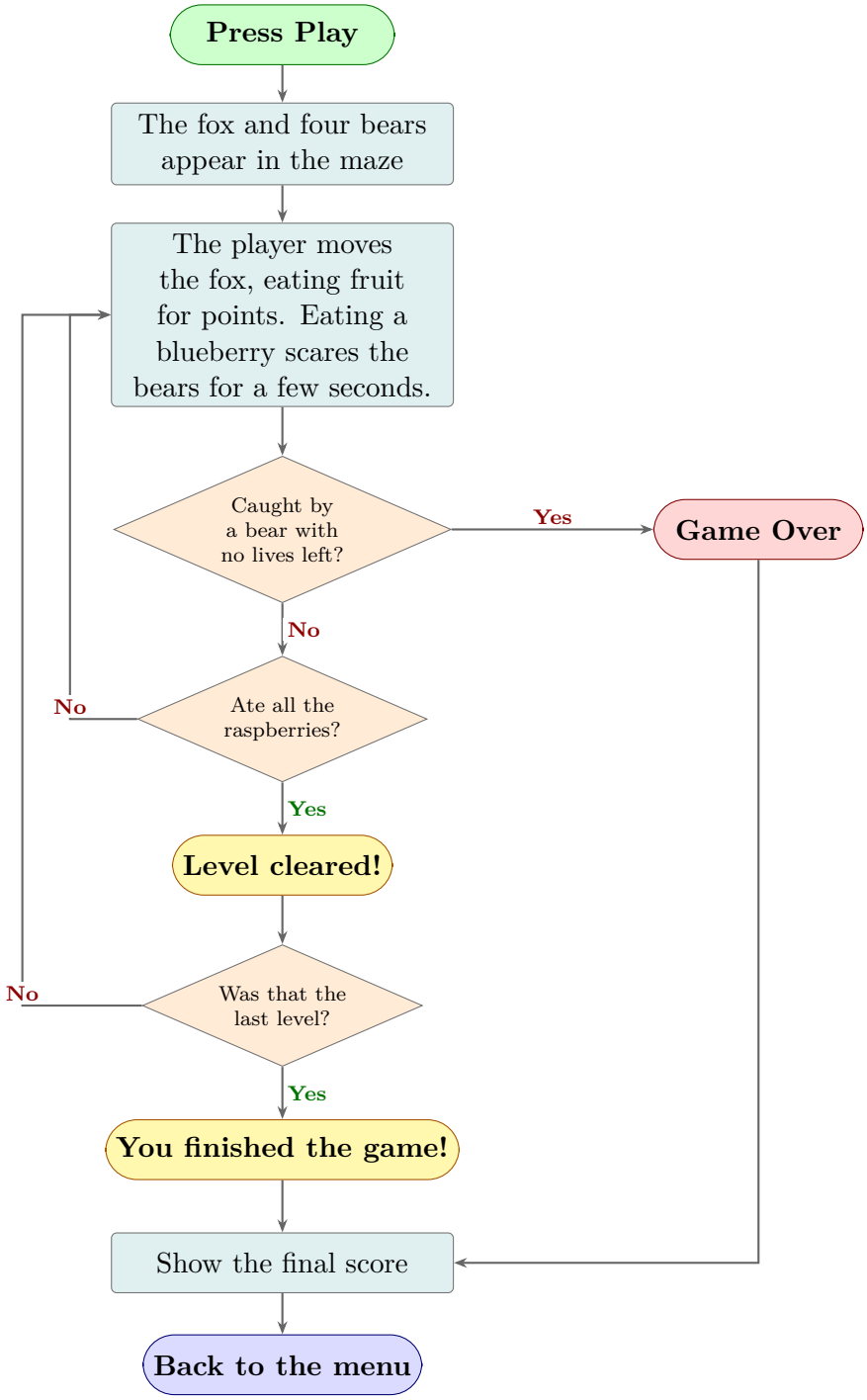


Figure 10: Flowchart for Foxy Fruit.

11.4.2 Project Structure

Foxy Fruit follows the same four file layout as the rest of the arcade.

```
FoxyFruit/
|-- Workspace/
|   +-- FoxyFruit/ (Folder)
|       +-- ButtonFoxyFruit        (Part, Tag="ButtonFoxyFruit")
|-- ReplicatedStorage/
|   |-- ArcadeRemotes/
|   |   +-- FoxyFruit/ (Folder)
```

```
|   |       |-- StartFoxyFruitRequest  (RemoteEvent)
|   |       |-- SubmitMove             (RemoteEvent)
|   |       |-- GameState              (RemoteEvent)
|   |       +-- ToggleAudio            (RemoteEvent)
|   +-- Modules/
|       +-- FoxyFruitModule        (ModuleScript)
|-- ServerScriptService/
|   +-- FoxyFruitServer            (Script)
+-- StarterPlayer/
    +-- StarterPlayerScripts/
        +-- FoxyFruitClient        (LocalScript)
```

11.4.3 Time Complexity

Each call to `update(dt)` costs $O(E \times N)$ time, where $E = 4$ is the number of bears and $N = 12 \times 16 = 192$ is the number of cells in the grid. Moving the player and eating a raspberry are $O(1)$, because the game looks the cell up directly. When a bear takes a step, it runs its BFS search from its cell to its target cell, and that search visits at most 192 cells. With 4 bears, the worst case per tick is about 4×192 steps. Sending the state to the screen goes through the grid once. Since the number of bears and the size of the grid are both fixed, all of this is really just a constant amount of work per tick, even if it looks large written out.

11.4.4 Space Complexity

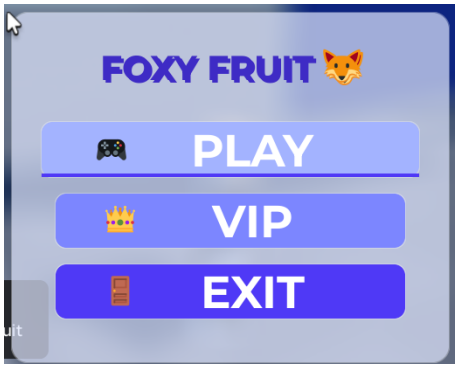
The memory cost is $O(N)$, where $N = 192$ cells. The grid is a 12×16 table of numbers that is copied fresh each time a level loads. Every BFS search creates two small helper tables the size of the grid and throws them away as soon as it finishes. The list of bears is always 4 entries. Everything else (score, lives, the power timer, the game state) is a single value, so the memory used in a session stays within a fixed limit.

11.4.5 Game Features and Implementation Status

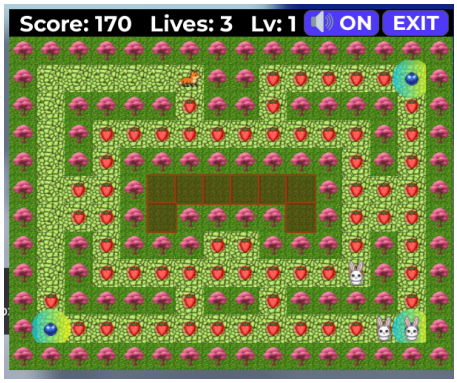
Feature	Description	Status
Main Menu	Interface with Play, VIP, and Exit buttons.	Working
Menu Sounds	Audio feedback for navigation and selection.	Working
Gameplay Controls	Grid based movement via arrow keys or WASD.	Working
Mute Button	In game toggle to turn all audio on or off.	Working
Exit Button (Game)	Allows the player to abort the run and return to the score menu.	Working
Score Menu	A screen after the game showing the final score and a replay option.	Working
Level System	Three handmade tile maps with increasing layout complexity. Clearing a level grants 1 extra life.	Working
Bear AI	Four bears with distinct BFS based targeting behaviors (direct chase, ambush, flank, territorial).	Working
Power Mode	Eating a blueberry activates a four second frightened state on all bears with exponential score multiplier.	Working
Top 10 Score Table	Global leaderboard via Roblox DataStore.	Working
VIP Donation Menu	Opens a menu with various donation tiers.	Working
Optimization	$O(E \times N)$ with $E = 4$ bears and $N = 192$ grid cells fixed; effectively $O(1)$ per tick.	Working
Mobile Adaptation	Touchscreen input via `UniversalController`.	Working
Xbox Adaptation	Gamepad input via `UniversalController`.	Working
PlayStation Adaptation	DualSense and DualShock input via `UniversalController`.	Working

Table 12: Functionalities and current status of Foxy Fruit.

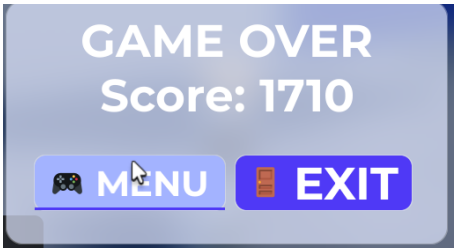
11.4.6 Game Results



Main Menu

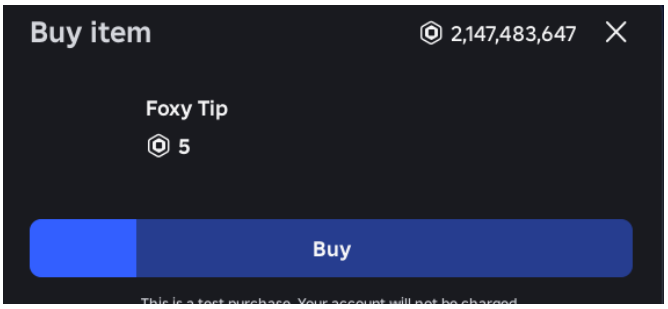


Gameplay



Score Menu

VIP Access



Donation Interface

Global Score Table

11.5 Firmament Defender

Firmament Defender is a wave based arcade shooter built inside Roblox Studio on a 16×12 grid. The tricky part to build was the group of planes. The 30 planes move together as one block, turn around and drop a row when they reach a wall, get faster as fewer of them are left, and fire from random planes along the bottom of the group. Three waves, each one faster than the last, make the game harder over time, and once in a while a UFO crosses the top of the screen as a risky but rewarding target.

11.5.1 Rules of Firmament Defender

- **Main Goal:** Destroy all planes across three waves before any plane descends to the ship's row.
- **Controls (PC):** Arrow keys or A/D to move the ship, Space to fire. Also adapted for mobile touchscreen, Xbox, and PlayStation controllers via `UniversalController`.
- **Plane Types:** Type C (top row, 300 pts), Type B (middle two rows, 200 pts), Type A (bottom two rows, 100 pts). Destroying one of the last 5 surviving planes adds a bonus of 500 points.
- **Swarm Movement:** Planes march horizontally as a single unit. When the leftmost or rightmost plane reaches a wall, the entire group drops one row and reverses direction. The base step interval is 1.20 s.
- **Low Count Boost:** When 5 or fewer planes remain, their step interval drops to 65% of its current value, down to a minimum of 0.30 s.
- **Enemy Fire:** Every 0.40 to 1.40 s (scaled by plane count), up to 3 random bottom row planes fire downward simultaneously.
- **Shields:** Four shield positions, each 2 cells wide and 3 HP. Both player and enemy bullets damage shields.
- **UFO:** Spawns every 8 to 16 s and crosses row 1 horizontally. Shooting it scores 1000 points and grants 1 extra life.
- **Waves:** Three waves total. Clearing a wave grants 1 extra life and reduces the plane step interval by 10%.
- **End Condition:** Losing all 4 lives or any plane reaching the ship's row ends the session. Clearing all three waves is a perfect game.

Flowchart

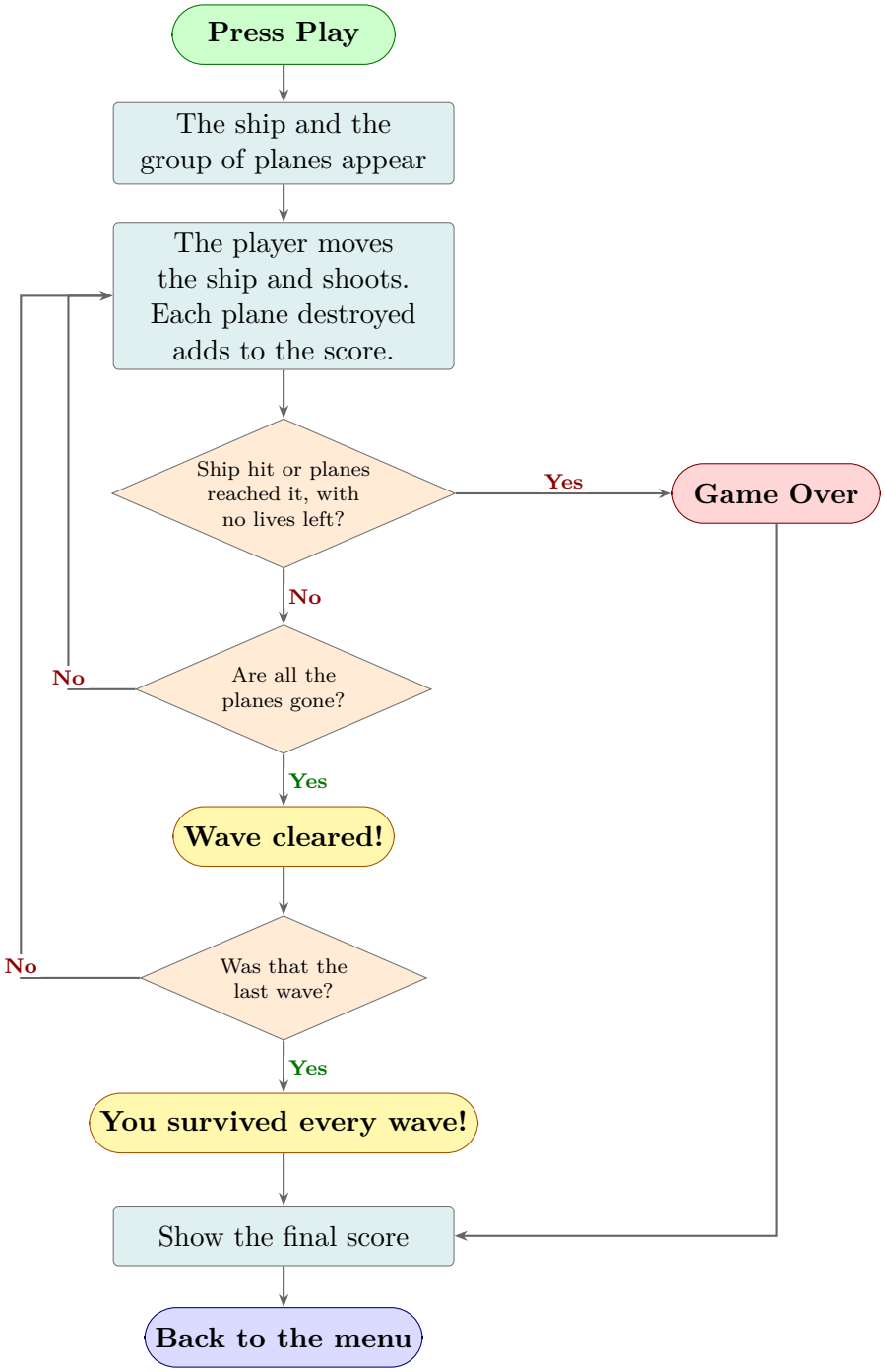


Figure 11: Flowchart for Firmament Defender.

11.5.2 Project Structure

Firmament Defender follows the same four file layout as the rest of the arcade.

```
FirmamentDefender/
|-- Workspace/
|   +-- FirmamentDefender/ (Folder)
|       +-- ButtonFirmamentDefender  (Part, Tag="ButtonFirmamentDefender")
|-- ReplicatedStorage/
|   |-- ArcadeRemotes/
|   |   +-- FirmamentDefender/ (Folder)
```

```
|   |       |-- StartFirmamentDefenderRequest  (RemoteEvent)
|   |       |-- SubmitMove                     (RemoteEvent)
|   |       |-- GameState                      (RemoteEvent)
|   |       +-- ToggleAudio                    (RemoteEvent)
|   +-- Modules/
|       +-- FirmamentDefenderModule  (ModuleScript)
|-- ServerScriptService/
|   +-- FirmamentDefenderServer      (Script)
+-- StarterPlayer/
    +-- StarterPlayerScripts/
        +-- FirmamentDefenderClient  (LocalScript)
```

11.5.3 Time Complexity

Each call to `update(dt)` costs $O(N)$ time, where N is the number of planes, which is never more than 30. Moving the ship is $O(1)$. Moving the whole group of planes and checking them against the shields goes through all the planes, which gives the $O(N)$. Choosing which planes shoot also goes through the list and picks up to 3, again $O(N)$. Checking the player's bullet against the planes is $O(N)$ too, and there is never more than one player bullet at a time. The UFO is $O(1)$. Because the number of planes only goes down as the player destroys them, the game actually gets lighter as each wave goes on. Sending the state to the screen copies the planes and shields once.

11.5.4 Space Complexity

The memory cost is $O(N)$, where N is at most 30 planes at the start of a wave. The game keeps the position and type of each plane, the 8 shield pieces (4 shields, 2 cells each), and the bullets. There is only ever one player bullet at a time, and the enemy bullets can never be more than the 12 rows of the grid. Everything else (the ship position, the wave number, the timers) is a single value, so the memory used in a session stays within a fixed limit.

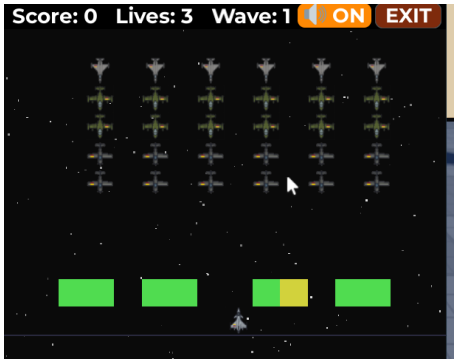
11.5.5 Game Features and Implementation Status

Feature	**Description**	**Status**
Main Menu	Interface with Play, VIP, and Exit buttons.	Working
Menu Sounds	Audio feedback for navigation and selection.	Working
Gameplay Controls	Ship movement (left/right) and continuous fire.	Working
Mute Button	In game toggle to turn all audio on or off.	Working
Exit Button (Game)	Allows the player to abort the run and return to the score menu.	Working
Score Menu	A screen after the game showing the final score and a replay option.	Working
Wave System	Three waves of increasing speed (step interval $\times 0.9$ per wave). Clearing a wave grants 1 extra life.	Working
Plane Swarm Logic	30 planes march as a group, drop and reverse at walls, and accelerate when 5 or fewer remain.	Working
Plane Type Scoring	Type A = 100 pts, Type B = 200 pts, Type C = 300 pts. Last 5 alive grant +500 bonus each.	Working
Destructible Shields	Four shield positions, 2 cells wide, 3 HP each. Damaged by both player and enemy bullets.	Working
UFO	Spawns every 8 to 16 s, crosses row 1, worth 1000 points and 1 extra life on hit.	Working
Top 10 Score Table	Global leaderboard via Roblox DataStore.	Working
VIP Donation Menu	Opens a menu with various donation tiers.	Working
Optimization	$O(N)$ per tick with $N \leq 30$ decreasing monotonically; player bullet capped at 1.	Working
Mobile Adaptation	Touchscreen input via `UniversalController`.	Working
Xbox Adaptation	Gamepad input via `UniversalController`.	Working
PlayStation Adaptation	DualSense and DualShock input via `UniversalController`.	Working

Table 13: Functionalities and current status of Firmament Defender.

11.5.6 Game Results

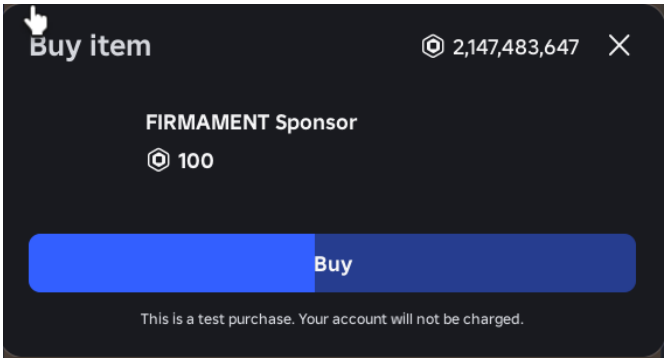
Main Menu



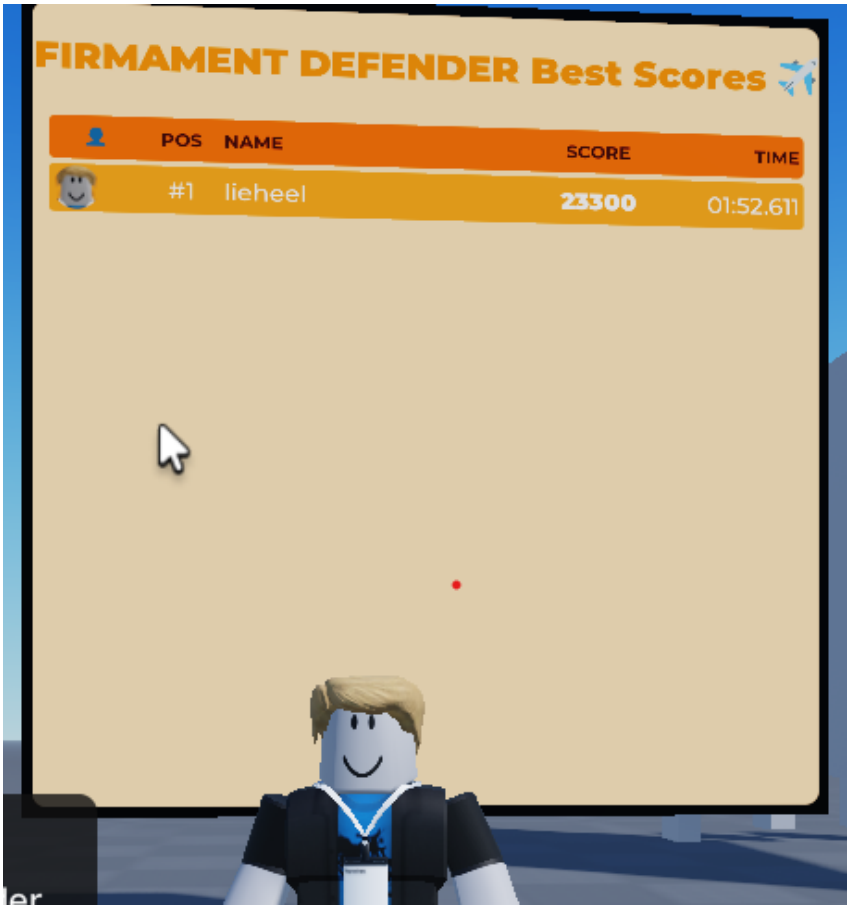
Gameplay

Score Menu

VIP Menu



Donation Interface



Global Score Table

11.6 Firmament Siege

Firmament Siege is a survival shooter with free movement, built inside Roblox Studio on a 16×12 open space. It had three main challenges. The first was the ship physics: the ship keeps drifting after the player stops pushing, and slows down little by little instead of stopping at once. The second was the screen wrapping: when the ship, a drone, or a bullet leaves one edge, it comes back from the opposite edge. The third was the aiming: the player can shoot in any direction, no matter which way the ship is moving. The game never really ends. Drones are destroyed and new ones appear, so there are always between 4 and 7 of them on screen.

11.6.1 Rules of Firmament Siege

- **Main Goal:** Survive as long as possible and accumulate the highest score by destroying drones.
- **Controls (PC):** WASD or arrow keys to thrust the ship in eight directions, mouse to aim, click or Space to fire, H for hyperspace. Also adapted for mobile touchscreen, Xbox, and PlayStation controllers via `UniversalController`.
- **Ship Physics:** The ship accelerates at 35 cells/s^2 up to a maximum of 10 cells/s. Releasing the thrust keys applies exponential damping ($0.82^{60\Delta t}$ per frame) rather than an instant stop.
- **Screen Wrapping:** The ship, all drones, and all bullets wrap toroidally: exiting one edge of the grid immediately reappears on the opposite edge.
- **Bullets:** Fire rate is capped at one bullet every 80 ms. Each bullet travels at 23 cells/s and disappears after 0.8 s (maximum range 18.4 cells).
- **Drone Types:** Type A (smallest, 1.4 to 2.2 cells/s, 150 pts), Type B (medium, 1.6 to 2.4 cells/s, 100 pts), Type C (largest, 1.8 to 2.6 cells/s, 50 pts). All drones steer directly toward the ship using toroidal vector math.
- **Dynamic Composition:** Every 3.2 s the game picks a new target total (4 to 7 drones) and spawns the missing drones outside a safety radius of 7 cells around the ship. The mix of drone types follows a fixed ratio for each total count.
- **Hyperspace:** Teleports the ship to a random position and grants a brief 0.6 s invulnerability extension.
- **Lives:** Three lives. After each death the ship respawns at the grid center with 1.1 s of invulnerability and nearby drones cleared.
- **End Condition:** Losing all three lives ends the session. The game has no win state: it runs until the player dies.

Flowchart

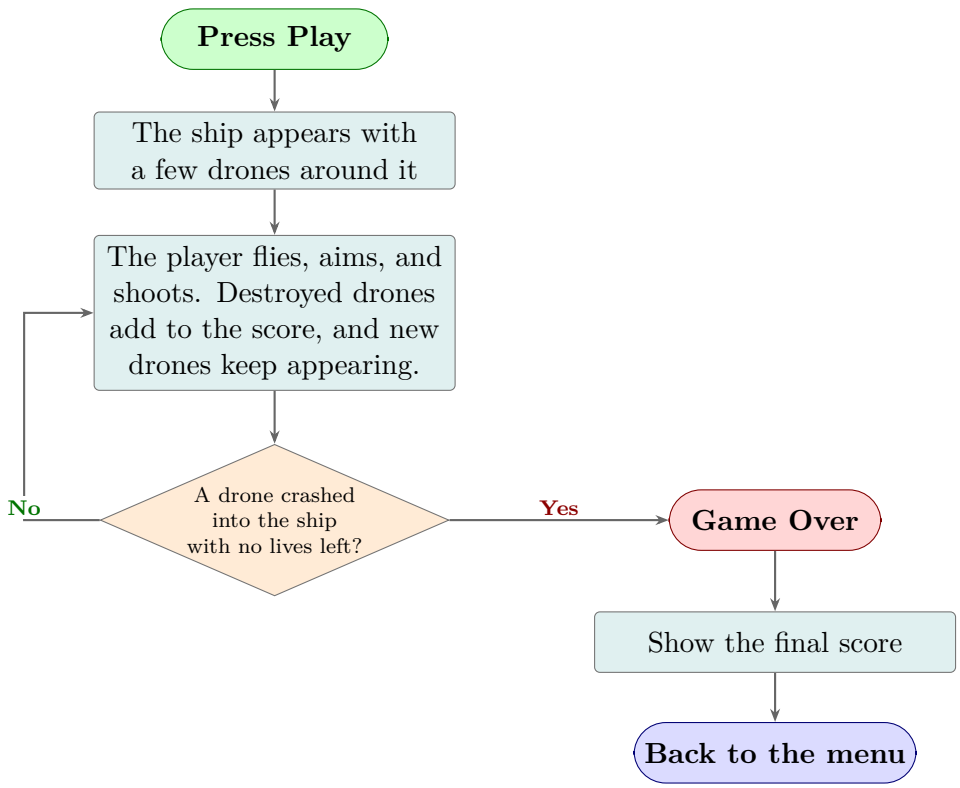


Figure 12: Flowchart for Firmament Siege.

11.6.2 Project Structure

Firmament Siege follows the same four file layout as the rest of the arcade.

```
FirmamentSiege/
|-- Workspace/
|   +-- FirmamentSiege/ (Folder)
|       +-- ButtonFirmamentSiege       (Part, Tag="ButtonFirmamentSiege")
|-- ReplicatedStorage/
|   |-- ArcadeRemotes/
|   |   +-- FirmamentSiege/ (Folder)
|   |       |-- StartFirmamentSiegeRequest  (RemoteEvent)
|   |       |-- SubmitMove                  (RemoteEvent)
|   |       |-- GameState                   (RemoteEvent)
|   |       +-- ToggleAudio                 (RemoteEvent)
|   +-- Modules/
|       +-- FirmamentSiegeModule       (ModuleScript)
|-- ServerScriptService/
|   +-- FirmamentSiegeServer           (Script)
+-- StarterPlayer/
    +-- StarterPlayerScripts/
        +-- FirmamentSiegeClient       (LocalScript)
```

11.6.3 Time Complexity

Each call to `update(dt)` costs $O(N \times M)$ time, where N is the number of drones (never more than 7) and M is the number of bullets on screen. Moving the ship is $O(1)$. Moving the bullets is $O(M)$, and steering the drones toward the ship is $O(N)$. The most expensive step is checking every bullet against

every drone, which is $O(N \times M)$. The number of bullets is limited too: the ship can only fire one bullet every 80 milliseconds and each bullet lasts 0.8 seconds, so there are never more than about 10 bullets at once. With 7 drones and 10 bullets, the worst case is around 70 checks, which is a small fixed number. Sending the state to the screen copies the drones and bullets once.

11.6.4 Space Complexity

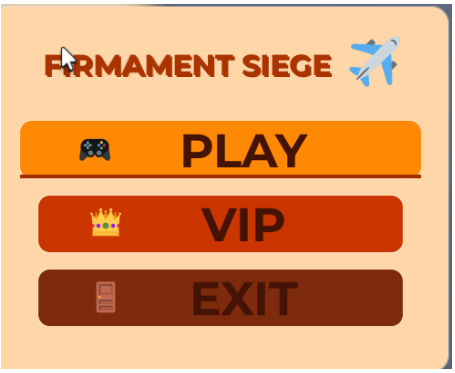
The memory cost is $O(N + M)$, with at most 7 drones and 10 bullets, both fixed numbers. Each drone keeps its position, speed, type, and rotation, and each bullet keeps its position, speed, and how much time it has left. Everything else (the ship's position and speed, the aim angle, the timers, the score, the lives) is a single value. Because of this, the memory used stays within a fixed limit no matter how long the game runs.

11.6.5 Game Features and Implementation Status

Feature	**Description**	**Status**
Main Menu	Interface with Play, VIP, and Exit buttons.	Working
Menu Sounds	Audio feedback for navigation and selection.	Working
Gameplay Controls	eight direction thrust, free aim shooting, and hyperspace teleport.	Working
Mute Button	In game toggle to turn all audio on or off.	Working
Exit Button (Game)	Allows the player to abort the run and return to the score menu.	Working
Score Menu	A screen after the game showing the final score and a replay option.	Working
Physics Engine	Inertia based movement with exponential damping and toroidal screen wrapping for all entities.	Working
Drone AI	Three drone types steer toward the ship via toroidal vector math at different speeds (1.4 to 2.6 cells/s).	Working
Dynamic Composition	Target drone count (4 to 7) chosen again every 3.2 s; missing drones spawn outside a safety radius of 7 cells.	Working
Hyperspace	Teleports the ship to a random safe position with a brief invulnerability extension.	Working
Top 10 Score Table	Global leaderboard via Roblox DataStore.	Working
VIP Donation Menu	Opens a menu with various donation tiers.	Working
Optimization	$O(N \times M)$ per tick with $N \leq 7$ drones and $M \leq 10$ bullets; effectively $O(1)$.	Working
Mobile Adaptation	Touchscreen input via `UniversalController`.	Working
Xbox Adaptation	Gamepad input via `UniversalController`.	Working
PlayStation Adaptation	DualSense and DualShock input via `UniversalController`.	Working

Table 14: Functionalities and current status of Firmament Siege.

11.6.6 Game Results

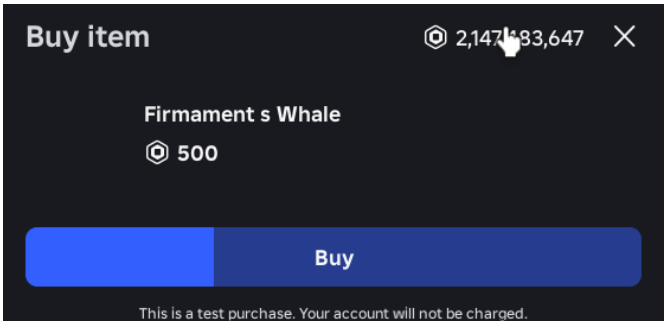


Main Menu

Gameplay

Score Menu

VIP Menu



Donation Interface

Global Score Table

11.7 Juice Cutter

Juice Cutter is a swipe based arcade game built inside Roblox Studio on a canvas of 380×360 pixels. Fruits fly up into the air and fall back down, and the player cuts them by drawing lines across the screen with a swipe. What makes it different is the recipe system: each of the five levels asks for certain fruits (for example, 15 lemons, or 12 cucumbers, 12 kiwis, and 12 green apples). Cutting the wrong fruit costs a life and takes away points, so paying attention to what is in the air matters as much as fast reactions.

11.7.1 Rules of Juice Cutter

- **Main Goal:** Complete the recipe for each of the five levels by cutting the required fruits before running out of lives.
- **Controls (PC):** Click and drag the mouse to draw a cut line. Also adapted for mobile touchscreen, Xbox, and PlayStation controllers via `UniversalController`.
- **Cut Detection:** A cut is registered when the swipe line segment passes within 40 pixels of a fruit center (point to segment distance test).
- **Good Fruits:** Cutting a target fruit scores points and counts toward the recipe goal.
- **Bad Fruits:** Cutting a wrong fruit deducts the same number of points and removes one life.
- **Physics:** Fruits launch from the bottom of the canvas at 520 to 850 px/s (varies by level) with a random horizontal velocity of ± 220 px/s. Gravity pulls them down at 950 px/s^2. They bounce off side walls and disappear when they fall below the canvas.
- **Levels:**
  - Level 1 (Lemonade): 20 lemons, 50 pts each, 65% chance of a good fruit.
  - Level 2 (StrawBanana): 15 strawberries and 15 bananas, 100 pts each, 60% chance of a good fruit.
  - Level 3 (Tropical): 12 pineapples, 12 coconuts, and 12 cherries, 150 pts each, 55%.
  - Level 4 (Green Detox): 12 cucumbers, 12 green apples, and 12 kiwis, 200 pts each, 48%.
  - Level 5 (Pure Red Juice): 15 grapes, 15 red apples, and 15 oranges, 250 pts each, 42%.
- **Lives:** Level 1 starts with 5 lives. Completing a level carries lives over with a +1 bonus.
- **Weighted Spawning:** In split goal levels, the spawner biases toward fruit kinds that still need to be cut, reducing blind luck situations near the end of a level.

Flowchart

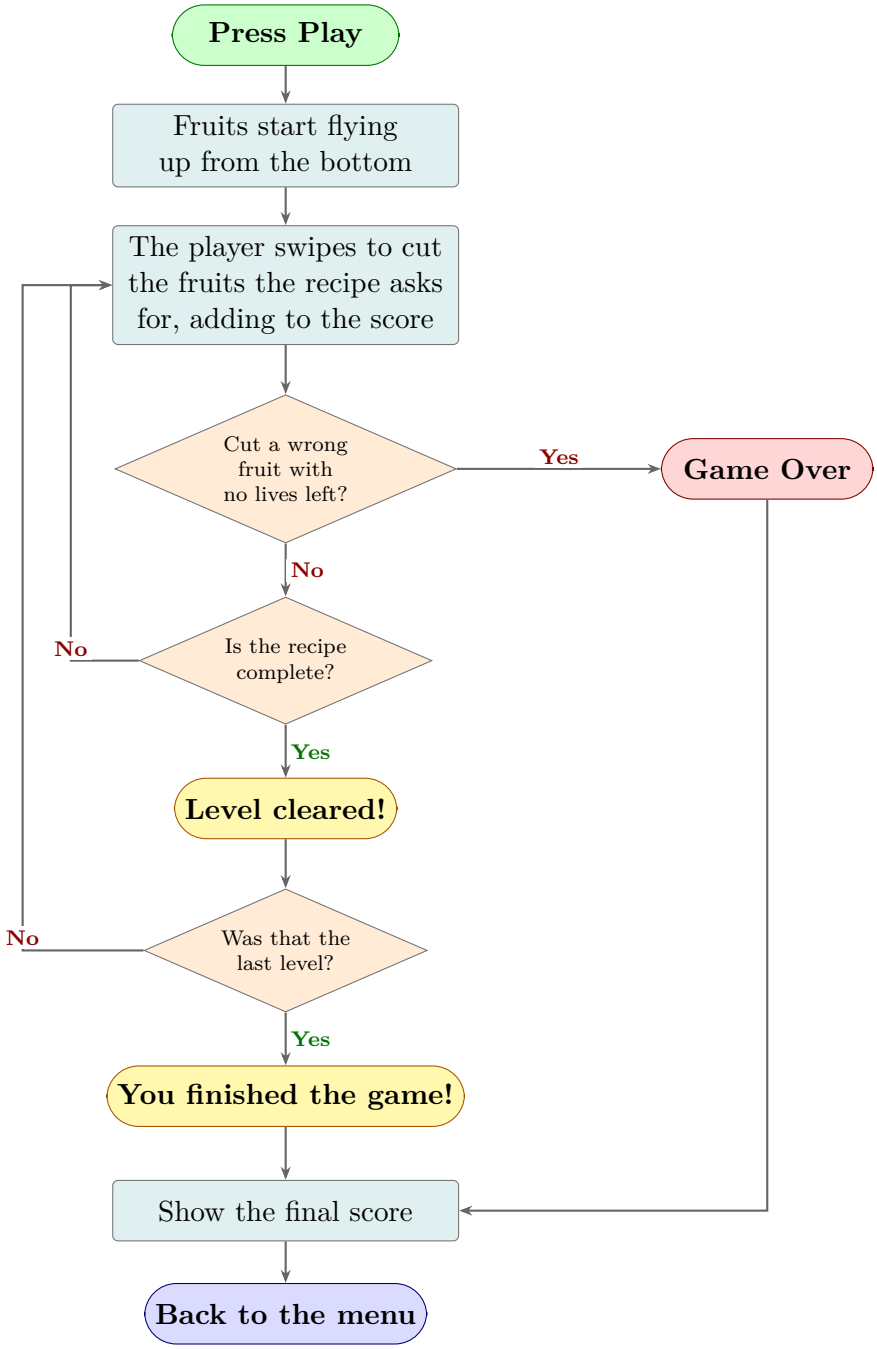


Figure 13: Flowchart for Juice Cutter.

11.7.2 Project Structure

Juice Cutter follows the same four file layout as the rest of the arcade.

```
JuiceCutter/
|-- Workspace/
|   +-- JuiceCutter/ (Folder)
|       +-- ButtonJuiceCutter        (Part, Tag="ButtonJuiceCutter")
|-- ReplicatedStorage/
|   |-- ArcadeRemotes/
|   |   +-- JuiceCutter/ (Folder)
|   |       |-- StartJuiceCutterRequest  (RemoteEvent)
|   |       |-- SubmitMove               (RemoteEvent)
```

```
|   |       |-- GameState                (RemoteEvent)
|   |       +-- ToggleAudio              (RemoteEvent)
|   +-- Modules/
|       +-- JuiceCutterModule        (ModuleScript)
|-- ServerScriptService/
|   +-- JuiceCutterServer            (Script)
+-- StarterPlayer/
    +-- StarterPlayerScripts/
        +-- JuiceCutterClient        (LocalScript)
```

11.7.3 Time Complexity

Each call to `step(dt)` costs $O(F)$ time, where F is the number of fruits in the air. Deciding whether to spawn a new fruit is $O(1)$. Updating the fruits goes through all of them to apply gravity, move them, bounce them off the walls, and remove the ones that have fallen off the screen, which gives the $O(F)$. Checking a swipe also goes through the fruits once to see which ones it cut. Sending the state to the screen copies the fruit list once. There are never many fruits at the same time: even at the fastest level, a fruit is only on screen for about half a second, so only a few are in the air at any moment. In practice the count stays well under 10, so the real cost is basically constant.

11.7.4 Space Complexity

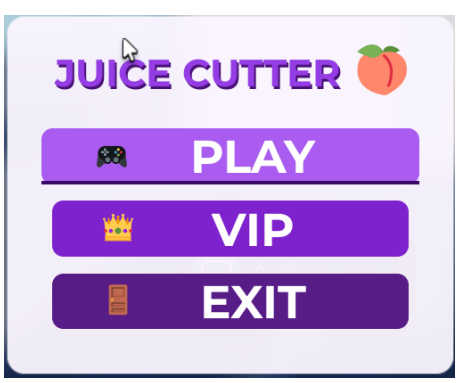
The memory cost is $O(F)$, where F is the small number of fruits in the air. Each fruit keeps its position, speed, rotation, kind, images, and whether it has been cut. The game also keeps a small counter for each kind of fruit the level needs, and a level never asks for more than 3 kinds. Everything else (score, lives, the current level, the spawn timer, the goal counters) is a single value, so the memory used in a session stays within a fixed limit.

11.7.5 Game Features and Implementation Status

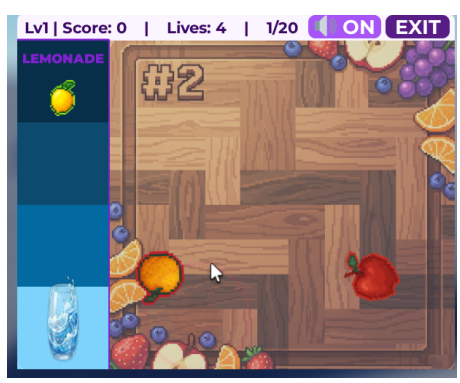
Feature	Description	Status
Main Menu	Interface with Play, VIP, and Exit buttons.	Working
Menu Sounds	Audio feedback for navigation and selection.	Working
Gameplay Controls	Click and drag swipe to cut fruits flying across the canvas.	Working
Mute Button	In game toggle to turn all audio on or off.	Working
Exit Button (Game)	Allows the player to abort the run and return to the score menu.	Working
Score Menu	A screen after the game showing the final score and a replay option.	Working
Level System	Five levels with unique recipes and increasing difficulty. Lives carry over with a +1 bonus per level cleared.	Working
Recipe System	Per level goals tracked per fruit kind. Weighted spawner biases toward still needed kinds in split goal levels.	Working
Physics Engine	Gravity ($950\,\mathrm{px/s^2}$), side wall bounce, random spin, and cut linger animation ($1.2\,\mathrm{s}$) per fruit.	Working
Cut Detection	Point to segment distance test against a $40\,\mathrm{px}$ radius per fruit; fires once per swipe event.	Working
Top 10 Score Table	Global leaderboard via Roblox DataStore.	Working
VIP Donation Menu	Opens a menu with various donation tiers.	Working
Optimization	$O(F)$ per tick with $F \leq 10$ active fruits; effectively $O(1)$.	Working
Mobile Adaptation	Touchscreen swipe input via `UniversalController`.	Working
Xbox Adaptation	Gamepad input via `UniversalController`.	Working
PlayStation Adaptation	DualSense and DualShock input via `UniversalController`.	Working

Table 15: Functionalities and current status of Juice Cutter.

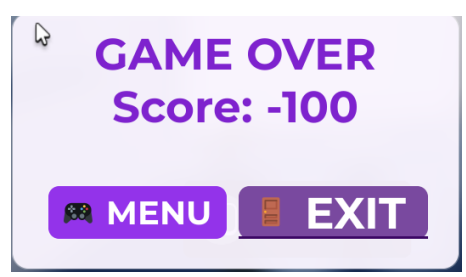
11.7.6 Game Results



Main Menu

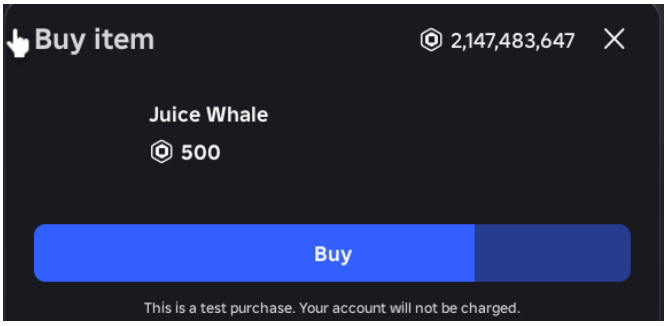


Gameplay



Score Menu

VIP Menu



Donation Interface

Global Score Table

11.8 Turris Casei

Turris Casei is a 3D stacking game built inside Roblox Studio. A block swings back and forth above the tower, and the player has to drop it at the right moment so it lands as well as possible on the block below. If it does not line up, the part that sticks out is cut off, and the landing area gets a little smaller for the next block. There were two main challenges. The first was the slicing math, which figures out the overlap between two blocks instantly using only simple arithmetic. The second was the multiplayer lobby, with private rooms protected by a PIN, live scores shared with everyone in the room, a spectator mode, and the option for the host to kick players.

11.8.1 Rules of Turris Casei

- **Main Goal:** Stack blocks as high as possible without losing the platform.
- **Controls (PC):** Space or click to drop the block. Also adapted for mobile touchscreen, Xbox, and PlayStation controllers via `UniversalController`.
- **Block Oscillation:** The active block swings sinusoidally at a fixed speed of 25 units/s. Its horizontal position is $\sin(t \times 2.5) \times \text{limit}$.
- **Perfect Landing:** If the misalignment on the active axis is ≤ 1.8 studs, the block snaps to a perfect placement and no trimming occurs.
- **Slicing:** Any overhang beyond the tolerance is trimmed. The remaining block inherits the overlap area. If the block misses entirely, the session ends.
- **Block Types (weighted random):**
  - Normal (74%): no size change, no score bonus.
  - Good (12.5%): +1 to size and score.
  - Good Bonus (3.5%): +2 to +5 size and score (rarer values are rarer).
  - Bad (10%): −0.5 size, −1 score penalty.
- **Max Block Size:** Capped at 7 studs regardless of bonuses.
- **Multiplayer:** A room host creates a PIN protected lobby (four digit, up to 50 players, 300 s timeout). Guests join by PIN. The host can kick players, start the match, or spectate others. Live scores are broadcast to all room members during the match.
- **End Condition:** A block that misses the platform entirely ends the player's session. In multiplayer, the last survivor wins.

Flowchart

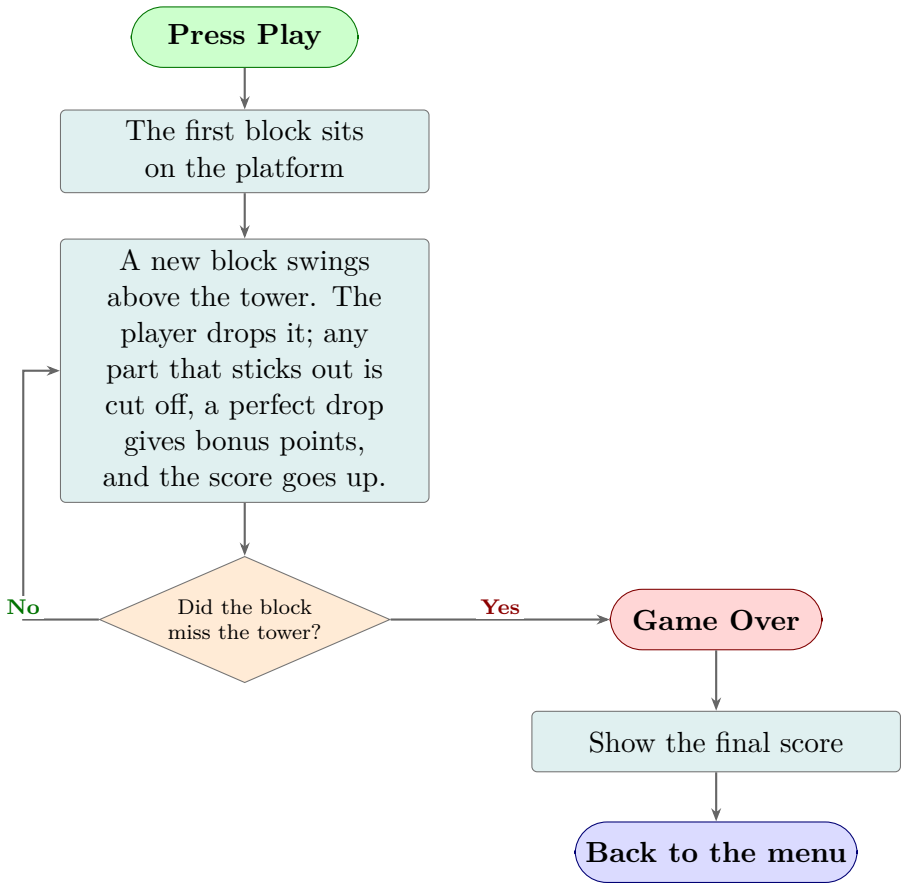


Figure 14: Flowchart for Turris Casei.

11.8.2 Project Structure

Turris Casei has a more complex server layout than the other games because it needs a dedicated match manager module for the multiplayer lobby.

```
TurrisCasei/
|-- Workspace/
|   +-- TurrisCasei/ (Folder)
|       +-- ButtonTurrisCasei          (Part, Tag="TurrisTerminal")
|-- ReplicatedStorage/
|   |-- ArcadeRemotes/
|   |   +-- TurrisCasei/ (Folder)
|   |       |-- StartTurrisCaseiRequest    (RemoteEvent)
|   |       |-- MatchStatus               (RemoteEvent)
|   |       |-- ToggleAudio               (RemoteEvent)
|   |       |-- UpdateTurrisCaseiLeaderboard (RemoteEvent)
|   |       |-- GetAudioState             (RemoteFunction)
|   |       +-- LobbySystem/ (Folder)
|   |           |-- CreateRoom            (RemoteFunction)
|   |           |-- JoinRoom              (RemoteFunction)
|   |           |-- StartMultiplayerMatch (RemoteEvent)
|   |           |-- RoomUpdate            (RemoteEvent)
|   |           |-- KickPlayer            (RemoteEvent)
|   |           |-- LeaveRoom             (RemoteEvent)
|   |           |-- UpdateLiveScore       (RemoteEvent)
|   |           |-- SpectateTarget        (RemoteEvent)
```

```
|   |           +-- PlayerDied            (RemoteEvent)
|   +-- Modules/
|       +-- TurrisCaseiModule          (ModuleScript)
|-- ServerScriptService/
|   |-- TurrisCaseiServer              (Script)
|   +-- TurrisCaseiMatchManager        (ModuleScript)
+-- StarterPlayer/
    +-- StarterPlayerScripts/
        +-- TurrisCaseiClient          (LocalScript)
```

11.8.3 Time Complexity

Every important step in the game runs in $O(1)$ time, meaning it always costs the same small amount no matter how tall the tower gets. Working out where the block is in its swing is a single calculation. Choosing the next block is one random draw plus a few checks. Working out the overlap between two blocks uses only subtractions and comparisons on a handful of numbers. There are no loops and nothing that grows while playing. The multiplayer match manager does keep a table for the room with up to 50 players, but the work done each time a block drops stays $O(1)$.

11.8.4 Space Complexity

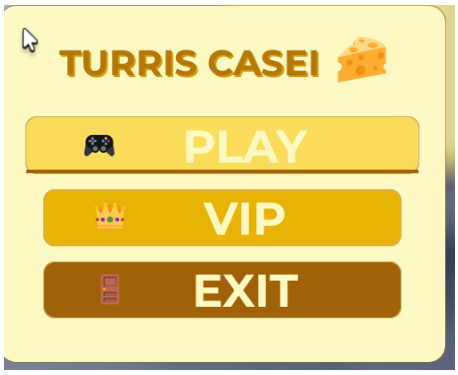
The memory the game logic uses is $O(1)$. To place a block, the game only needs the position and size of the previous block and the position of the current one, which are just a few numbers. It does not keep a history of the blocks already placed. The match manager keeps some data for each room (the player list, the score table, who is spectating) for up to 50 players, but that does not depend on how many blocks have been stacked.

11.8.5 Game Features and Implementation Status

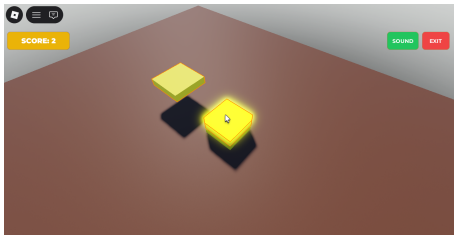
Feature	**Description**	**Status**
Main Menu	Interface with Play, VIP, and Exit buttons.	Working
Menu Sounds	Audio feedback for navigation and selection.	Working
Gameplay Controls	One button drop (Space or click/tap).	Working
Mute Button	In game toggle to turn all audio on or off.	Working
Exit Button (Game)	Allows the player to abort the run and return to the score menu.	Working
Score Menu	A screen after the game showing the final score and a replay option.	Working
Block Slicing Logic	$O(1)$ overlap calculation using axis aligned coordinate arithmetic. Trims overhang to the shared area.	Working
Block Types	Weighted random draw each drop: Normal 74%, Good 12.5%, Good Bonus 3.5%, Bad 10%.	Working
Perfect Landing	Misalignment $\leq$ 1.8 studs snaps to center with no size loss.	Working
Multiplayer Lobby	PIN protected rooms (four digit), up to 50 players, 300 s timeout, host kick controls.	Working
Live Score Broadcast	Real time score table sent to all room members after every block drop.	Working
Spectator Mode	Players eliminated from the match can spectate any surviving participant.	Working
Anti AFK	Inactivity detection to prevent stalling in multiplayer matches.	Working
Top 10 Score Table	Global leaderboard via Roblox DataStore.	Working
VIP Donation Menu	Opens a menu with various donation tiers.	Working
Optimization	$O(1)$ per block drop; $O(P)$ lobby data with $P \leq 50$ players.	Working
Mobile Adaptation	Touchscreen input via `UniversalController`.	Working
Xbox Adaptation	Gamepad input via `UniversalController`.	Working
PlayStation Adaptation	DualSense and DualShock input via `UniversalController`.	Working

Table 16: Functionalities and current status of Turris Casei.

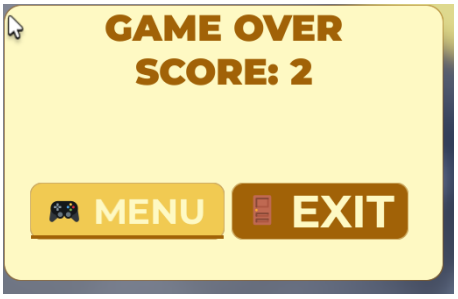
11.8.6 Game Results



Main Menu

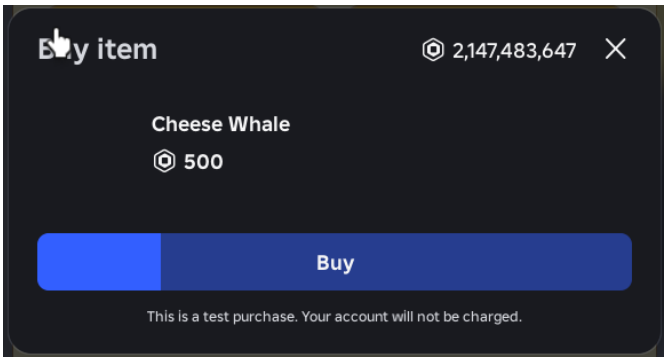


Gameplay



Score Menu

VIP Menu



Donation Interface

Global Score Table

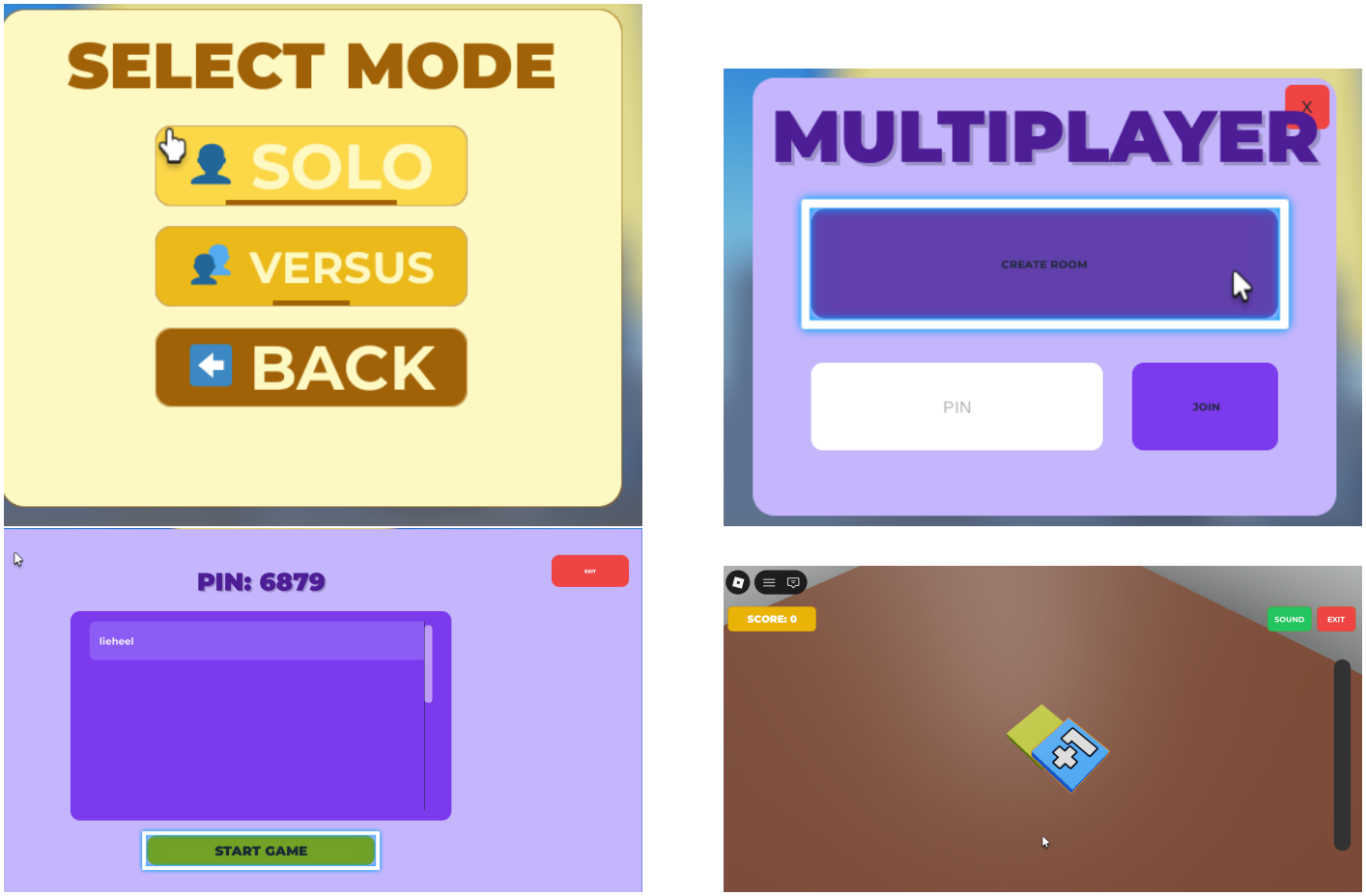


Figure 15: Multiplayer Lobby System: PIN entry, room creation, and versus mode.

11.9 Tap And Hold

Tap And Hold is a 3D jumping game built inside Roblox Studio. The player holds a button to charge a jump and lets go to send the character flying in a curve toward the next platform. The longer the hold, the farther the jump, up to a maximum charge of about 1.1 seconds. The platforms are created on the fly in a zigzag, with at most 4 visible at a time, and the old ones are removed as the player moves forward. The multiplayer works the same way as in Turris Casei: private rooms protected by a PIN, live scores after every jump, a spectator mode for players who fall, and the option for the host to kick players.

11.9.1 Rules of Tap And Hold

- **Main Goal:** Reach as many platforms as possible without falling.
- **Controls (PC):** Hold Space or click to charge the jump; release to launch. Also adapted for mobile touchscreen, Xbox, and PlayStation controllers via `UniversalController`.
- **Jump Physics:** Hold time is converted to power: $\text{power} = \min(\text{holdTime} \times 45,\ 50)$. Forward velocity is $\text{power} \times 2.2$ and upward velocity is $35 + \text{power} \times 0.2$. Gravity is set to 196.2 studs/s^2.
- **Perfect Landing:** Landing within 1.2 studs of the platform center awards a perfect bonus and triggers a visual highlight.
- **Platform Events (per platform, random):**
  - Normal (80%): no score modifier.
  - Bonus (10%): +5 score on landing.
  - Penalty (10%): −1 score on landing.
- **Platform Generation:** Platforms are chosen from four weighted models (tapsuelo1 and tapsuelo2 at 32.5% each, tapsuelo3 and tapsuelo4 at 17.5% each) and placed in a zigzag pattern between 8 and 22 studs apart. At most 4 platforms exist in the world at any instant; the oldest is destroyed when a new one spawns.
- **Multiplayer:** PIN protected rooms (four digit), up to 50 players, 300 s lobby timeout. The host can kick players, start the match, or spectate others. Live scores are broadcast to all room members after every jump.
- **End Condition:** Falling off a platform ends the player's session. In multiplayer, the last survivor wins.

Flowchart

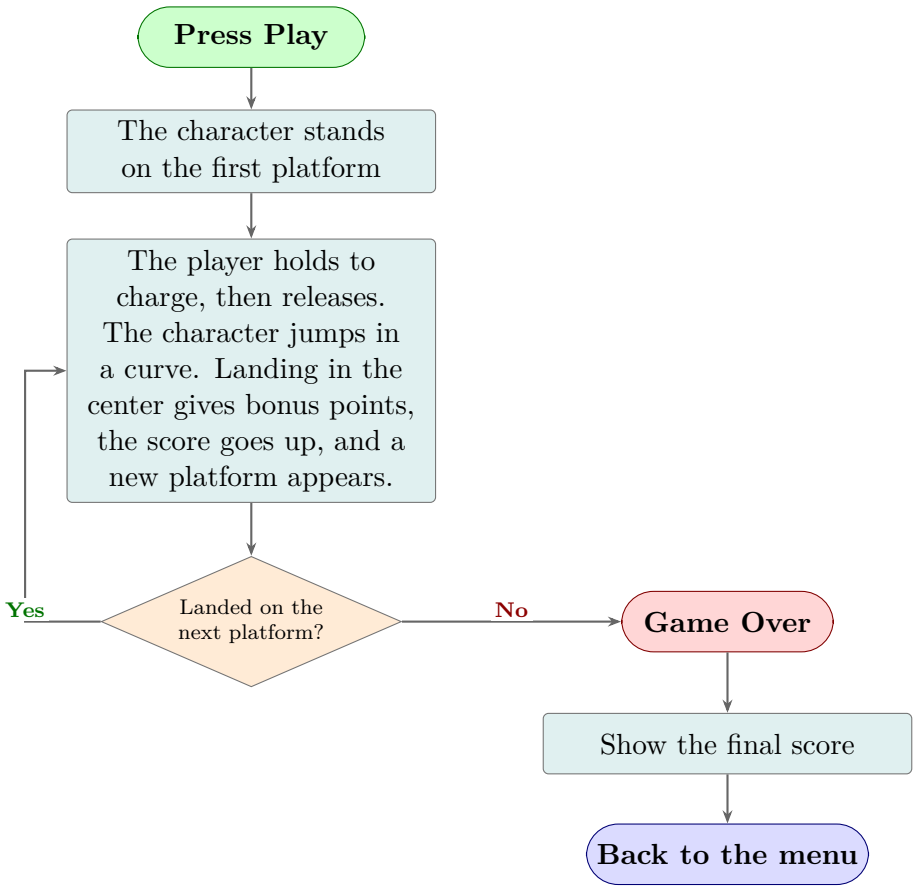


Figure 16: Flowchart for Tap And Hold.

11.9.2 Project Structure

Tap And Hold shares the same expanded server layout as Turris Casei, with a dedicated match manager and separate leaderboard client.

```
TapAndHold/
|-- Workspace/
|   +-- TapAndHold/ (Folder)
|       +-- ButtonTapAndHold           (Part, Tag="TapAndHoldTerminal")
|-- ReplicatedStorage/
|   |-- ArcadeRemotes/
|   |   +-- TapAndHold/ (Folder)
|   |       |-- StartTapAndHoldRequest     (RemoteEvent)
|   |       |-- SubmitJumpCharge           (RemoteEvent)
|   |       |-- MatchStatus               (RemoteEvent)
|   |       |-- ToggleAudio               (RemoteEvent)
|   |       |-- UpdateTapAndHoldLeaderboard (RemoteEvent)
|   |       |-- GetAudioState             (RemoteFunction)
|   |       +-- LobbySystem/ (Folder)
|   |           |-- CreateRoom            (RemoteFunction)
|   |           |-- JoinRoom              (RemoteFunction)
|   |           |-- StartMultiplayerMatch (RemoteEvent)
|   |           |-- RoomUpdate            (RemoteEvent)
|   |           |-- KickPlayer            (RemoteEvent)
|   |           |-- LeaveRoom             (RemoteEvent)
|   |           |-- UpdateLiveScore       (RemoteEvent)
```

```
|   |           |-- SpectateTarget        (RemoteEvent)
|   |           +-- PlayerDied            (RemoteEvent)
|   |-- TapAndHold_Assets/ (Folder)
|   |   |-- tapsuelo1, tapsuelo2          (Platform models)
|   |   |-- tapsuelo3, tapsuelo4          (Platform models)
|   |   +-- Model                         (Floor model)
|   +-- Modules/
|       +-- TapAndHoldModule           (ModuleScript)
|-- ServerScriptService/
|   |-- TapAndHoldServer               (Script)
|   +-- TapAndHoldMatchManager         (ModuleScript)
+-- StarterPlayer/
    +-- StarterPlayerScripts/
        +-- TapAndHoldClient           (LocalScript)
```

11.9.3 Time Complexity

Every step in the game runs in $O(1)$ time, always costing the same small amount no matter how far the player gets. Turning the hold time into a jump is two short calculations. Checking the landing is a few comparisons and one distance. Choosing the next platform goes through a fixed list of 4 platform types plus one extra roll. Creating a new platform works out one coordinate and removes the oldest one from a list that never holds more than 4. The match manager keeps some data for the room with up to 50 players, but the work done on each jump stays $O(1)$.

11.9.4 Space Complexity

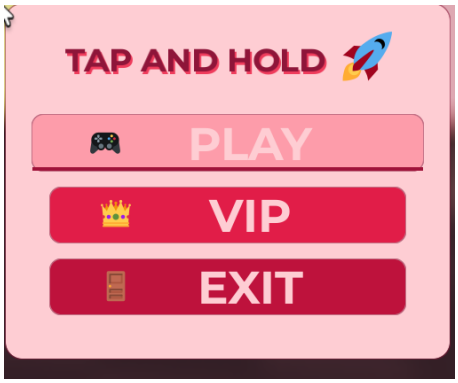
The memory the game logic uses is $O(1)$. There are never more than 4 platforms in the world at once, and the oldest one is removed whenever a new one appears. Checking a landing only needs the character's position and the current platform's position and size. Turning the hold into a jump uses just a couple of numbers. The match manager keeps some data for each room (the player list, the score table, who is spectating) for up to 50 players, and that does not depend on how many jumps the player has made.

11.9.5 Game Features and Implementation Status

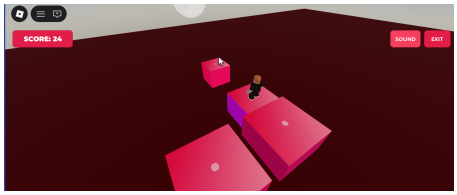
Feature	Description	Status
Main Menu	Interface with Play, VIP, and Exit buttons.	Working
Menu Sounds	Audio feedback for navigation and selection.	Working
Gameplay Controls	Hold to charge jump, release to launch in a parabolic arc.	Working
Mute Button	In game toggle to turn all audio on or off.	Working
Exit Button (Game)	Allows the player to abort the run and return to the score menu.	Working
Score Menu	A screen after the game showing the final score and a replay option.	Working
Parabolic Jump Physics	Hold time converted to forward ($\times 2.2$) and upward ($35 + \times 0.2$) velocity; gravity $196.2\,\text{studs/s}^2$.	Working
Perfect Landing	Landing within 1.2 studs of center triggers a visual effect and bonus score.	Working
Platform Events	Random per platform: Normal 80%, Bonus $+5$ (10%), Penalty -1 (10%).	Working
Infinite Platform Generation	Procedural zigzag spawning from 4 weighted models; at most 4 visible at any time, oldest destroyed on spawn.	Working
Multiplayer Lobby	PIN protected rooms (four digit), up to 50 players, 300 s timeout, host kick controls.	Working
Live Score Broadcast	Real time score table sent to all room members after every jump.	Working
Spectator Mode	Players who fall can spectate any surviving participant.	Working
Anti AFK	Inactivity detection to prevent stalling in multiplayer matches.	Working
Top 10 Score Table	Global leaderboard via Roblox DataStore.	Working
VIP Donation Menu	Opens a menu with various donation tiers.	Working
Optimization	$O(1)$ per jump; max 4 platforms in memory at any time.	Working
Mobile Adaptation	Touchscreen hold and release input via `UniversalController`.	Working
Xbox Adaptation	Gamepad button hold via `UniversalController`.	Working
PlayStation Adaptation	DualSense and DualShock hold input via `UniversalController`.	Working

Table 17: Functionalities and current status of Tap And Hold.

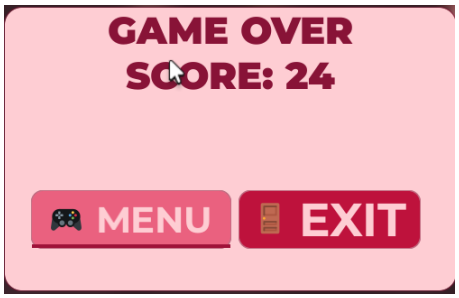
11.9.6 Game Results



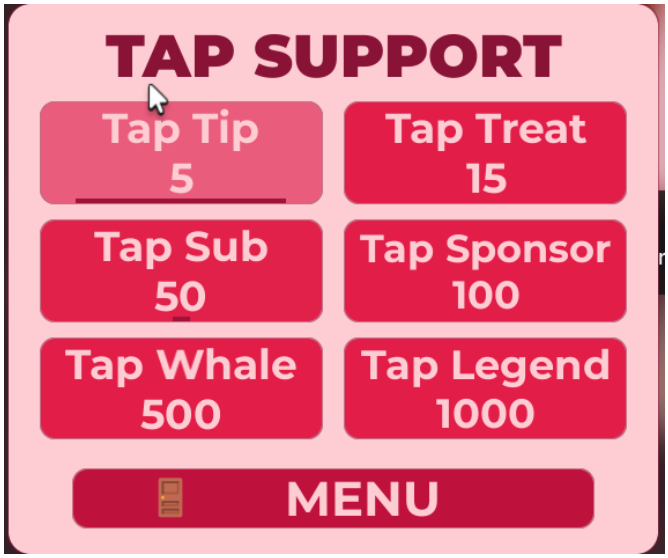
Main Menu



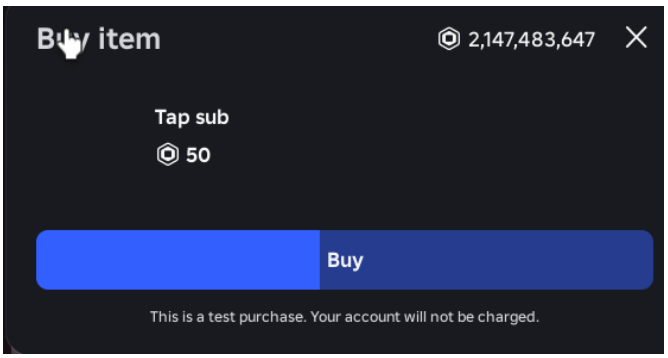
Gameplay



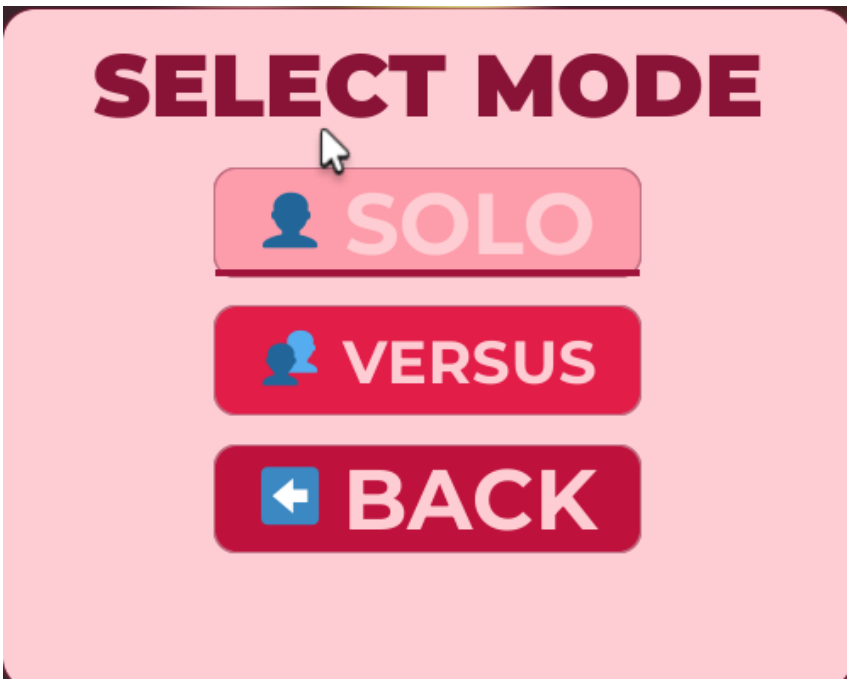
Score Menu



Play Mode Menu



Donation Interface



Global Score Table

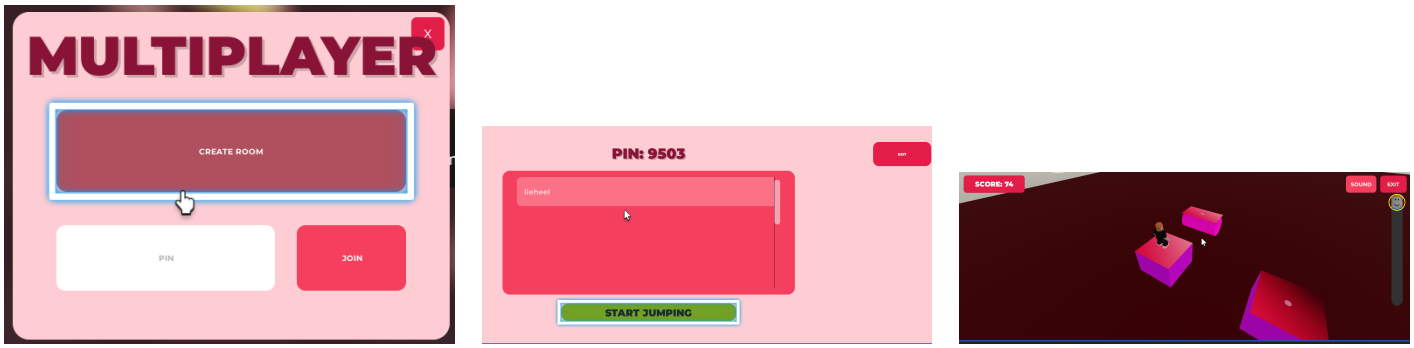


Figure 17: Multiplayer Lobby System: PIN entry, room creation, and versus mode.

11.10 ImNotASphere

ImNotASphere is a 3D ball platformer built inside Roblox Studio. The player rolls a sphere through three levels, each one with moving platforms, elevators, coins to collect, kill zones, and a finish line. There are also four power ups that change the speed, the size, the jump height, or let the ball float for a moment, plus a multiplayer lobby. The hard part to build was the ball physics, written from scratch: the module works out the push on the ball, how much control the player has in the air, how the ball rolls, and how far the elevators travel, all without using Roblox's normal character movement.

11.10.1 Rules of ImNotASphere

- **Main Goal:** Collect all four required Bitcoin coins on each of the three levels and reach the finish line.
- **Controls (PC):** WASD or arrow keys to roll the sphere, Space to jump. Also adapted for mobile touchscreen, Xbox, and PlayStation controllers via `UniversalController`.
- **Physics:** The sphere is driven by a custom force model. On the ground, acceleration is 80 units/s^2 up to a top speed of 40 units/s, with braking deceleration of 60 units/s^2. In the air, only 50% of the ground acceleration is available (AirControl = 0.5). Jump impulse is 50 units $\times$ mass.
- **Collectibles:** Standard Bitcoin coins score 5 points. Super Bitcoin coins score 15 points. Collecting all four required Bitcoins on a level unlocks the finish line.
- **Power ups (respawn every 5 s):**
  - Fast: 1.5 s duration, speed $\times 3$, acceleration $\times 2.5$.
  - Big: 6.0 s duration, sphere size $\times 2.5$.
  - Wings (ALAS): 3.5 s duration, magnetic levitation up to 20 studs with spring force 30 and damping 8.
  - Spring: 2.0 s duration, jump impulse $\times 2$.
- **Levels:** Three handmade levels, each containing geometry, moving platforms with elevator systems, collectibles, kill zones (death pit below the level), a spawn point, and a finish line.
- **Lives:** 8 lives total. Falling into a kill zone or off the level costs one life.
- **Multiplayer:** PIN protected rooms (four digit), up to 10 players, 300 s lobby timeout, host kick controls, live score broadcasting, and spectator mode for eliminated players.
- **Score Validation:** The server caps accepted scores at 490 and rejects results submitted in under 1 second to prevent exploits.
- **End Condition:** Losing all 8 lives ends the session. Clearing all three levels is a perfect run.

Flowchart

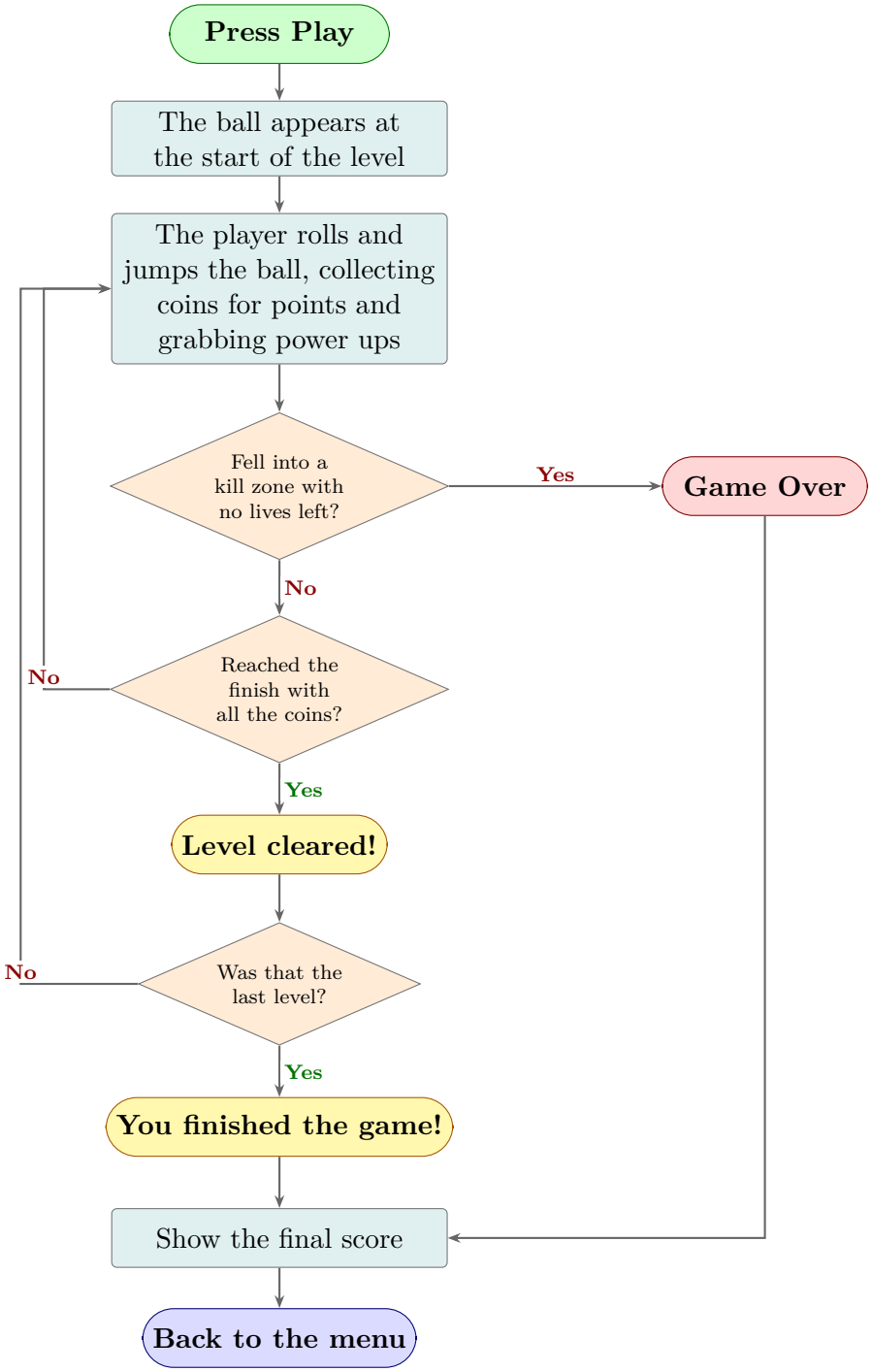


Figure 18: Flowchart for ImNotASphere.

11.10.2 Project Structure

ImNotASphere shares the same expanded server layout as Turris Casei and Tap And Hold, with a dedicated match manager and separate leaderboard client.

```
ImNotASphere/
|-- Workspace/
|   +-- ImNotASphere/ (Folder)
|       +-- ButtonImNotASphere     (Part, Tag="ImNotASphereTerminal")
|-- ReplicatedStorage/
|   |-- ArcadeRemotes/
```

```
|   |   +-- ImNotASphere/ (Folder)
|   |       |-- StartImNotASphereRequest   (RemoteEvent)
|   |       |-- MatchStatus               (RemoteEvent)
|   |       |-- ToggleAudio               (RemoteEvent)
|   |       |-- UpdateImNotASphereLeaderboard (RemoteEvent)
|   |       |-- GetAudioState             (RemoteFunction)
|   |       +-- LobbySystem/ (Folder)
|   |           |-- CreateRoom            (RemoteFunction)
|   |           |-- JoinRoom              (RemoteFunction)
|   |           |-- StartMultiplayerMatch (RemoteEvent)
|   |           |-- RoomUpdate            (RemoteEvent)
|   |           |-- KickPlayer            (RemoteEvent)
|   |           |-- LeaveRoom             (RemoteEvent)
|   |           |-- UpdateLiveScore       (RemoteEvent)
|   |           |-- SpectateTarget        (RemoteEvent)
|   |           +-- PlayerDied            (RemoteEvent)
|   |-- ImNotASphere_Assets/ (Folder)
|   |   |-- PlayerSphere                  (Part, player ball)
|   |   |-- Level1, Level2, Level3        (Models)
|   |   |   |-- Geometry/                 (Folder, optional)
|   |   |   |-- MovingPlatforms/          (Folder)
|   |   |   |   +-- Elevator1             (Model: Platform + StartPart +
   EndPart)
|   |   |   |-- Collectibles/             (Folder)
|   |   |   |   |-- Coin                  (Bitcoin, 5 pts)
|   |   |   |   +-- ExtraPoint            (Super Bitcoin, 15 pts)
|   |   |   |-- KillZones/               (Folder)
|   |   |   |   +-- DeathPit             (Transparent part below level)
|   |   |   |-- SpawnPoint               (Transparent part)
|   |   |   +-- FinishLine               (Part, unlocked after 4 Bitcoins)
|   +-- Modules/
|       |-- ImNotASphereModule         (ModuleScript)
|       +-- ImNotASphereLeaderboard    (ModuleScript)
|-- ServerScriptService/
|   |-- ImNotASphereServer             (Script)
|   +-- ImNotASphereMatchManager       (ModuleScript)
+-- StarterPlayer/
    +-- StarterPlayerScripts/
        +-- ImNotASphereClient         (LocalScript)
```

11.10.3 Time Complexity

Every function in the physics module runs in $O(1)$ time, always costing the same small amount. Working out the push on the ball is a fixed set of operations that checks at most 4 active power ups. Working out how the ball rolls is a single calculation, the jump is one multiplication, and the elevator distance is one subtraction between two points. There are no loops and nothing that grows while playing. The match manager keeps some data for the room with up to 10 players, and that does not depend on how long the game has been running.

11.10.4 Space Complexity

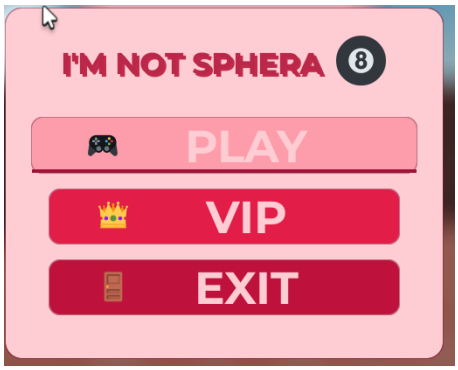
The memory the module uses is $O(1)$. The physics module does not store anything of its own: it takes the inputs, returns the result, and keeps no data between calls. The list of active power ups it receives never has more than 4 entries. The real game state (the ball's position and speed, the lives, the coins) lives in the 3D world as Roblox parts, handled by the client. The match manager keeps some data for each room for up to 10 players.

11.10.5 Game Features and Implementation Status

Feature	Description	Status
Main Menu	Interface with Play, VIP, and Exit buttons.	Working
Menu Sounds	Audio feedback for navigation and selection.	Working
Gameplay Controls	WASD to roll the sphere, Space to jump.	Working
Mute Button	In game toggle to turn all audio on or off.	Working
Exit Button (Game)	Allows the player to abort the run and return to the score menu.	Working
Score Menu	A screen after the game showing the final score and a replay option.	Working
Custom Physics Engine	Force based ball controller: ground acceleration 80, top speed 40, air control 0.5, jump impulse 50.	Working
Level System	Three handmade levels with geometry, moving platforms, elevators, collectibles, kill zones, and finish lines.	Working
Collectible System	Four required Bitcoin coins (5 pts each) unlock the finish line. Super Bitcoin coins score 15 pts.	Working
Power up System	Four power ups: Fast ($\times 3$ speed, 1.5 s), Big ($\times 2.5$ size, 6 s), Wings (levitation up to 20 studs, 3.5 s), Spring ($\times 2$ jump, 2 s). Respawn every 5 s.	Working
Elevator System	Moving platforms travel between StartPart and EndPart markers; distance calculated via `CalculateElevatorLimits`.	Working
Multiplayer Lobby	PIN protected rooms (four digit), up to 10 players, 300 s timeout, host kick controls.	Working
Live Score Broadcast	Real time score updates sent to all room members.	Working
Spectator Mode	Eliminated players can spectate any surviving participant.	Working
Anti cheat Validation	Server rejects scores above 490 or submitted in under 1 second.	Working
Top 10 Score Table	Global leaderboard via Roblox DataStore.	Working
VIP Donation Menu	Six donation tiers (5 to 1000 Robux) with purchase deduplication via DataStore.	Working
Optimization	$O(1)$ per physics tick; module is stateless with no dynamic allocations.	Working
Mobile Adaptation	Touchscreen input via `UniversalController`.	Working
Xbox Adaptation	Gamepad input via `UniversalController`.	Working
PlayStation Adaptation	DualSense and DualShock input via `UniversalController`.	Working

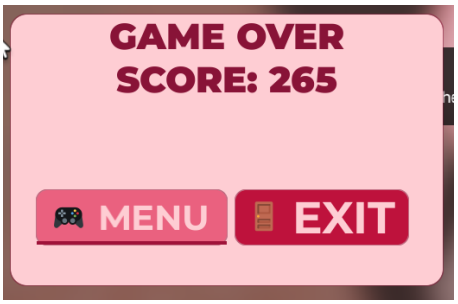
Table 18: Functionalities and current status of ImNotASphere.

11.10.6 Game Results



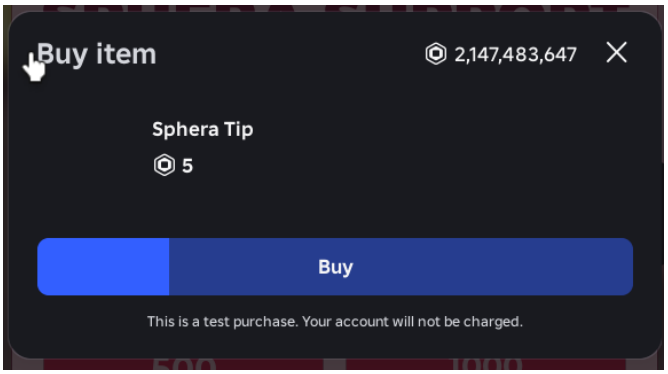
Main Menu

Gameplay

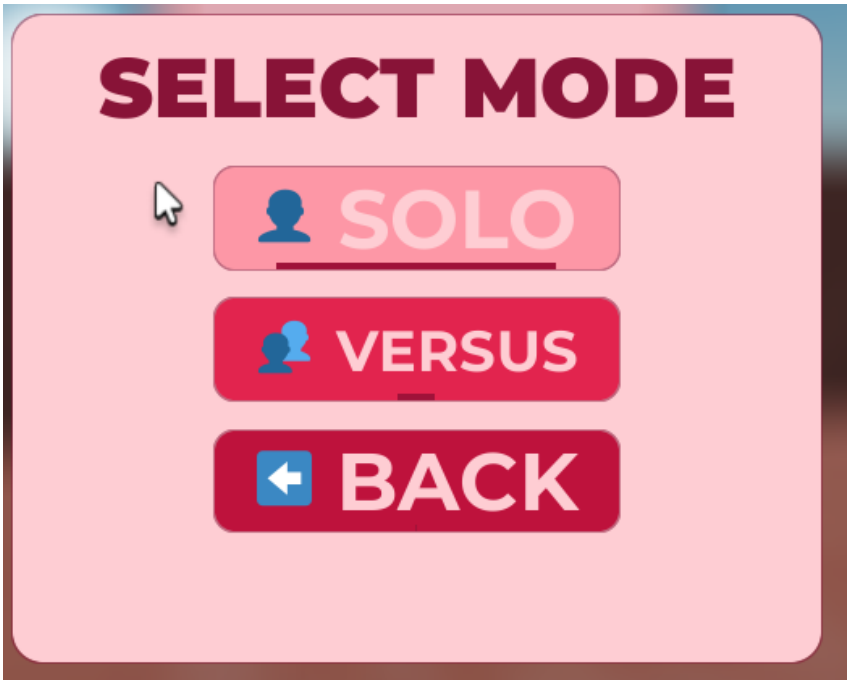


Score Menu

VIP Menu



Donation Interface



Global Score Table

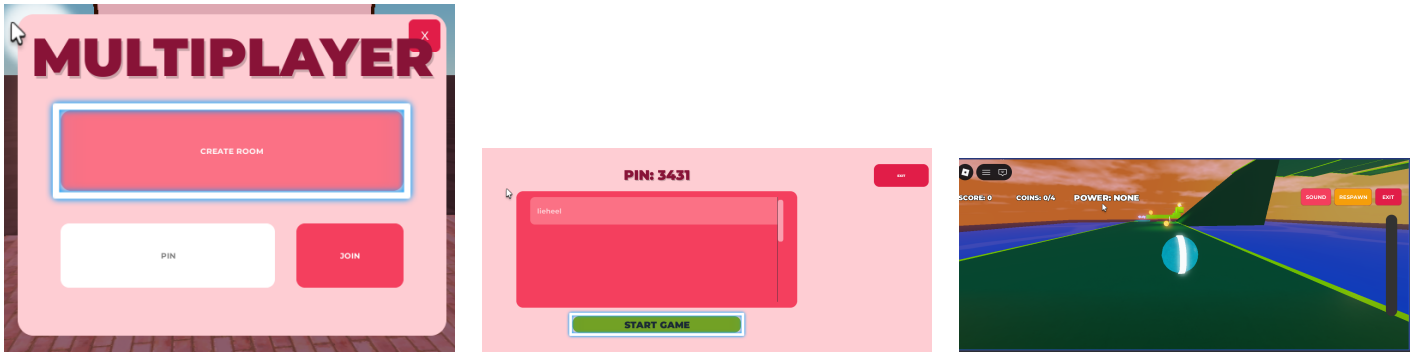


Figure 19: Multiplayer Lobby System: PIN entry, room creation, and versus mode.

12 Shared Systems

The arcade has ten games in total. Instead of writing the same interface, input, and leaderboard code again inside each one, three systems are shared across all of them: the score table system (the leaderboards), the menu system (the main menu, the VIP screen, and the game over screen), and the universal controller (the input that works on every device). This section explains how each shared system is built and which games use it.

12.1 Score Table System

All ten games report scores to a persistent leaderboard, so every game has an online ranking. There are two mechanisms, and which one a game uses depends on its complexity:

- **Universal leaderboard** (seven games): Oxybelis, Avis Toucan, Resilio, Foxy Fruit, Firmament Defender, Firmament Siege, and Juice Cutter all report to the shared `UniversalLeaderboardModule`, which renders a single dynamic board (`UniversalLeaderboardClient`) that switches its title, icon, and color palette per game.
- **Dedicated score table** (three games): Turris Casei, Tap And Hold, and ImNotASphere each have their own in world board with a game specific leaderboard module and client. These three were given dedicated tables because they are the multiplayer games and need a board that lives permanently in their own arcade area.

Every board, universal or dedicated, is rendered as a physical part in the Workspace and updated in real time whenever any player finishes a session. The rest of this subsection describes the dedicated score table architecture; the universal leaderboard follows the same data flow pattern with a single shared module instead of one per game.

The state diagram below shows the idea: every board runs the very same code, and its `GameId` attribute is what decides which game it represents.

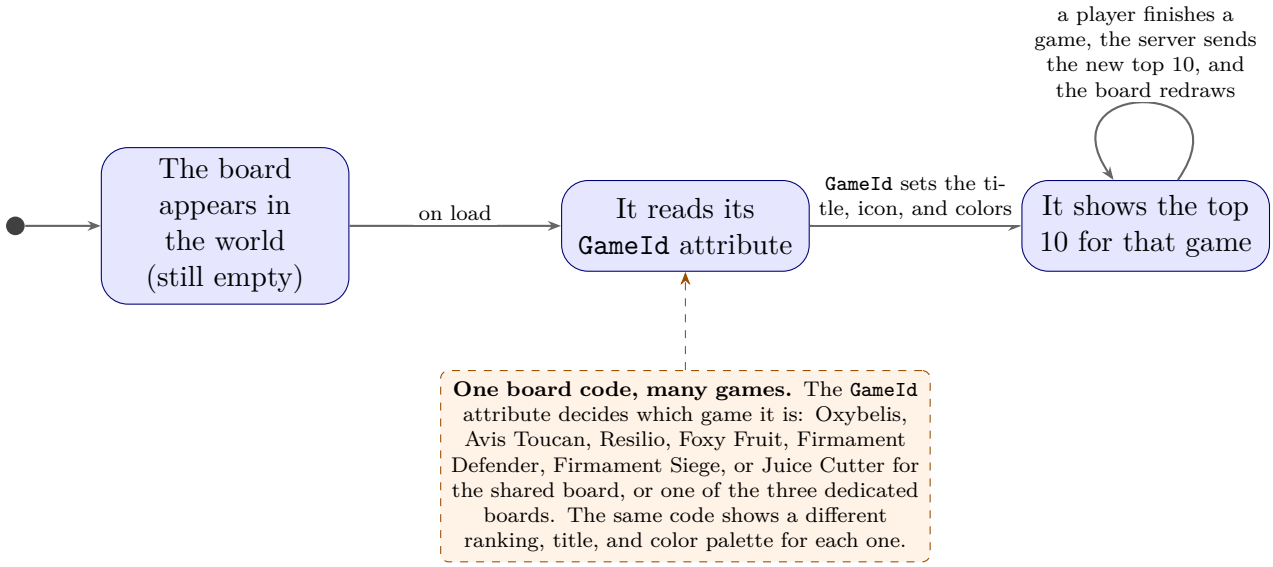


Figure 20: State diagram of a score table board. One code base behaves as any game's leaderboard depending on its `GameId` attribute.

12.1.1 Architecture

Both mechanisms share the same three part architecture: a physical board in the Workspace, a module that handles DataStore persistence and ranking, and a client that renders the board. The only difference is whether these parts are game specific (dedicated) or shared across games (universal).

- **Workspace Board** (Part): the physical leaderboard that players see in the arcade. Dedicated games use a named part (e.g. `TurrisCaseiLeaderboardPart`); universal games use a tagged `UniversalLeaderboardPart` carrying a `GAMEID` attribute that tells the shared client which game's scores to show.
- **Leaderboard Module** (ModuleScript in ReplicatedStorage/Modules): contains the DataStore read/write logic and the sorting algorithm that maintains the top ranking. It is required by the server and called after every game over event. Dedicated games have their own module; universal games all share `UniversalLeaderboardModule`.
- **Leaderboard Client** (LocalScript in StarterPlayerScripts): listens for the `Update...Leaderboard` RemoteEvent and rebuilds the board UI whenever new data arrives. Dedicated games have their own client; universal games all share `UniversalLeaderboardClient`.

Dedicated architecture (example: Turris Casei):

```
|-- Workspace/
|   +-- TurrisCaseiLeaderboardPart     (named Score Table board)
|-- ReplicatedStorage/
|   |-- ArcadeRemotes/
|   |   +-- TurrisCasei/
|   |       +-- UpdateTurrisCaseiLeaderboard  (RemoteEvent)
|   +-- Modules/
|       +-- TurrisCaseiLeaderboard     (ModuleScript: DataStore + ranking)
+-- StarterPlayer/
    +-- StarterPlayerScripts/
        +-- TurrisCaseiLeaderboardClient (LocalScript: UI renderer)
```

Universal architecture (shared by the other seven games):

```
|-- Workspace/
|   |-- UniversalLeaderboardPart (Tag="LeadboradOxybelis", GAMEID="Oxybelis")
|   |-- UniversalLeaderboardPart (Tag="LeadboradResilio",  GAMEID="Resilio")
|   +-- ... one tagged part per universal game ...
|-- ReplicatedStorage/
|   |-- ArcadeRemotes/
|   |   +-- GlobalBoard/
|   |       +-- UpdateUniversalLeaderboard  (RemoteEvent, shared)
|   +-- Modules/
|       +-- UniversalLeaderboardModule  (ModuleScript: shared DataStore +
   ranking)
+-- StarterPlayer/
    +-- StarterPlayerScripts/
        +-- UniversalLeaderboardClient  (LocalScript: shared renderer)
```

12.1.2 Data Flow

1. A player finishes a session and the server calls the leaderboard module to report the score.

2. The module reads the current top-10 from DataStore, inserts the new score if it qualifies, sorts by score descending (ties broken by time ascending), and writes the updated list back.
3. The server fires `UpdateLeaderboard` to all clients with the new top-10 payload.
4. Each leaderboard client receives the event and redraws the physical board in the Workspace.

12.1.3 Games Using This System

All ten games are listed below with the mechanism each uses. The seven universal games share a single module and client, distinguished only by their `GAMEID`; the three dedicated games each have their own module and client.

Game	**Mechanism**	**RemoteEvent**	**Leaderboard Module**
Oxybelis	Universal	UpdateUniversalLeaderboard	UniversalLeaderboardModule
Avis Toucan	Universal	UpdateUniversalLeaderboard	UniversalLeaderboardModule
Resilio	Universal	UpdateUniversalLeaderboard	UniversalLeaderboardModule
Foxy Fruit	Universal	UpdateUniversalLeaderboard	UniversalLeaderboardModule
Firmament Defender	Universal	UpdateUniversalLeaderboard	UniversalLeaderboardModule
Firmament Siege	Universal	UpdateUniversalLeaderboard	UniversalLeaderboardModule
Juice Cutter	Universal	UpdateUniversalLeaderboard	UniversalLeaderboardModule
Turris Casei	Dedicated	UpdateTurrisCaseiLeaderboard	TurrisCaseiLeaderboard
Tap And Hold	Dedicated	UpdateTapAndHoldLeaderboard	TapAndHoldLeaderboard
ImNotASphere	Dedicated	UpdateImNotASphereLeaderboard	ImNotASphereLeaderboard

Table 19: Leaderboard mechanism for all ten games.

12.2 Menu System

Every game opens the same style of menu: a main screen with PLAY, VIP, and EXIT buttons, a VIP donation screen, and a game over screen. Rather than rebuild this for each game, the menus are centralized into two shared modules in `ReplicatedStorage/Menu`. The split exists because the first seven games are single player and need only a simple menu, while the last three add a multiplayer lobby that requires a much larger menu module.

```
ReplicatedStorage/
+-- Menu/ (Folder)
    |-- Menu1to7   (ModuleScript: single player menus, games 1-7)
    +-- Menu8to10  (ModuleScript: menus + multiplayer lobby, games 8-10)
```

The table below shows exactly which game uses which menu module:

Menu Module	Games	Type
Menu1to7	Oxybelis, Avis Toucan, Resilio, Foxy Fruit, Firmament Defender, Firmament Siege, Juice Cutter	Single player menus
Menu8to10	Turris Casei, Tap And Hold, ImNotASphere	Menus + multiplayer lobby

Table 20: Menu module assignment per game.

The state diagram below shows how a player moves between the menu screens. The Lobby states only exist for the three multiplayer games that use `Menu8to10`; the other seven use `Menu1to7` and never enter the Lobby.

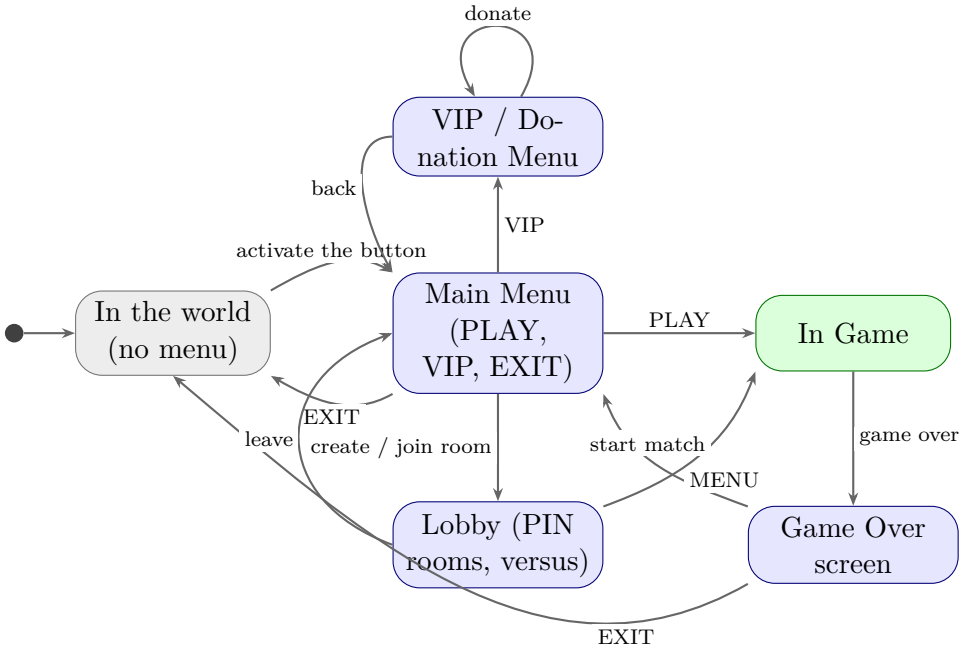


Figure 21: State diagram of the menu system. The Lobby states belong to `Menu8to10` (the three multiplayer games); `Menu1to7` skips them.

12.2.1 Menu1to7 (Single Player Menu)

`Menu1to7` drives the menus for the seven single player games: Oxybelis, Avis Toucan, Resilio, Foxy Fruit, Firmament Defender, Firmament Siege, and Juice Cutter. Each game passes a configuration table describing its title, icon, color palette, and donation products, and the module builds the UI from that data. This is why all seven games share an identical menu layout but each has its own colors and branding.

The module exposes four public functions:

- `AbrirMenuPrincipal(config)` builds the main menu with the PLAY, VIP, and EXIT buttons from the config that was passed in.
- `AbrirMenuVIP()` builds the donation grid, with one button for each product in the config.
- `AbrirGameOver(score, perfect)` shows the final score with the MENU and EXIT options.
- `CerrarMenu()` closes all the menu screens and gives the camera and the player controls back.

Beyond layout, the module handles focus management (`EstablecerEnfoque`): it applies a blur effect, freezes the camera, disables the default character controller, and turns off proximity prompts while a menu is open. Keyboard and gamepad navigation is routed through the universal controller so the same up/down/confirm/back logic works on every platform.

12.2.2 Menu8to10 (Menu + Multiplayer Lobby)

`Menu8to10` drives the menus for the three multiplayer games: Turris Casei, Tap And Hold, and ImNotA-Sphere. It includes everything `Menu1to7` does, plus a full lobby system: room creation with a PIN code, a screen to join a room by its PIN, a room player list with host kick controls, a button to start a versus match, and a spectator view. Instead of a config table passed per call, it exposes a `Config` table (title, color palettes for both menu and lobby, donation products) and a `Callbacks` table that the game's client script fills in with handlers like `OnCreateRoom`, `OnJoinRoom`, `OnStartMatch`, and `OnKickPlayer`.

The lobby uses a second color palette (`LobbyColors`) distinct from the main menu palette, so the lobby screens are visually separated from the game's branded menu.

12.3 Universal Controller

The universal controller (`UniversalController`, a ModuleScript in StarterPlayerScripts) is the single input layer shared by all ten games. Its job is to map one logical action set per game onto four different input methods: keyboard, mouse, gamepad, and on screen touch controls. Each game calls one bind function, and the controller wires up every platform at once. This is what makes the "adapted for mobile, Xbox, and PlayStation" claim true for every game without per platform code inside the games themselves.

The state diagram below shows how the controller switches its mode based on the last device the player used. The game actions stay the same; only the way they are read changes.

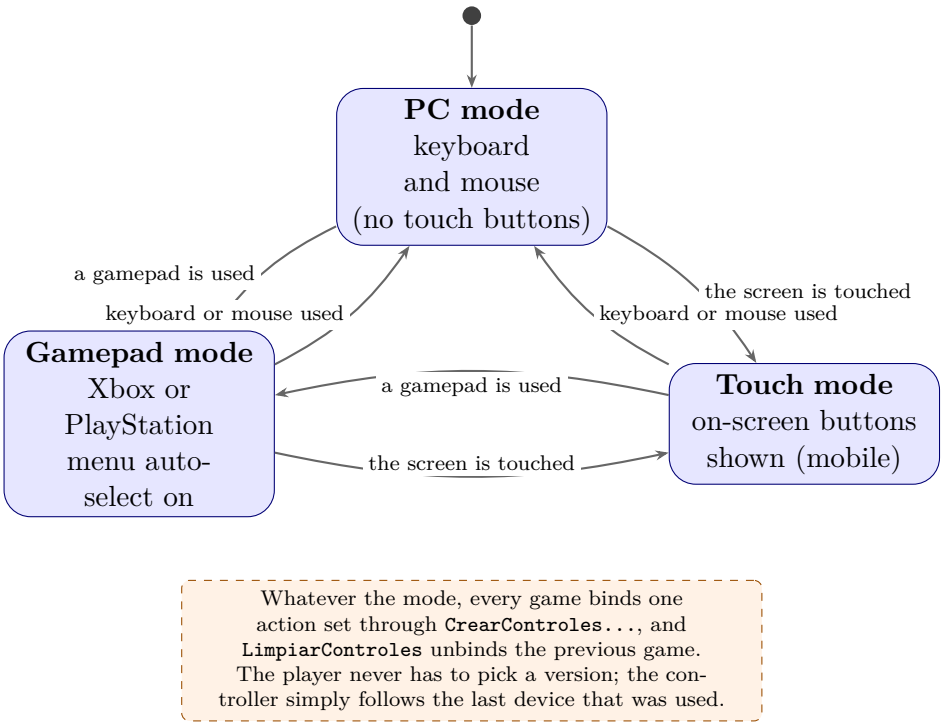


Figure 22: State diagram of the universal controller. It follows the last input device, switching between PC, gamepad, and touch.

12.3.1 Menu Navigation Helpers

Three functions provide cross platform menu navigation, used by both menu modules:

- `EsBotonSeleccion(input)` is the confirm key: Enter, Space, or the A button on a gamepad.
- `EsBotonAtras(input)` is the cancel key: Escape or the B button on a gamepad.
- `ObtenerDireccionMenu(input)` turns WASD, the arrow keys, or the direction pad into a direction.

12.3.2 Per Game Control Bindings

Each game has a dedicated bind function that registers its keyboard and gamepad actions via `ContextActionServ` and, on touch devices, spawns the matching on screen controls. When a new game starts, `LimpiarControles` unbinds the previous game's actions and destroys its virtual controls, so only one control set is ever active.

All bind functions share the `CrearControles` prefix (e.g. `CrearControlesOxybelis`); the table below lists each by its game specific suffix.

Game (bind suffix)	Control Scheme	Touch UI
Oxybelis	four direction direction pad	Dual cross pad
Flappy (Avis Toucan)	Single flap button	Tap buttons
Resilio	Left/right + launch	Cross pad variant
FoxyFruit	four direction direction pad	Dual cross pad
Firmament (Defender)	four direction + fire	Cross pad + fire button
FirmamentSiege	Twin stick + fire	Dual joystick + fire
JuiceCutter	Joystick + cut	Gamepad / touch swipe
Turris (Casei)	Single drop button	Drop button
TapAndHold	Hold / release button	Hold jump button
ImNotASphere	Twin stick + jump + power	Dual joystick + 2 buttons

Table 21: Per game control bindings in the Universal Controller (function = `CrearControles` + suffix).

12.3.3 Virtual Control Widgets

For touch devices, the controller builds the on screen UI from a small set of reusable widgets, all sharing one neon visual style: a four direction cross pad, a circular action button, a draggable analog joystick (with normalized vector output), and a Resilio specific cross pad with a wide launch button. A connection to `LastInputTypeChanged` automatically hides the touch controls and enables gamepad GUI navigation when a controller is detected, and shows them again on touch input.

12.4 Multiplayer Lobby System

Three of the ten games support multiplayer: Turris Casei, Tap And Hold, and ImNotASphere. They all share the exact same lobby system, so it is documented once here rather than three times. On the server it is driven by a match manager, and on the client by a fixed set of RemoteEvents and RemoteFunctions (`CreateRoom`, `JoinRoom`, `StartMultiplayerMatch`, `RoomUpdate`, `UpdateLiveScore`, `PlayerDied`, `SpectateTarget`, `KickPlayer`, and `LeaveRoom`).

Rooms are private and protected by a four digit PIN. There are no public lobbies and no random matchmaking: a player can only join a room if someone shares the PIN with them outside the game, through a chat or a message. This is a deliberate safety choice for a young audience, not a technical limit, and the host can remove any player from the room at any time.

The sequence diagram below follows the whole flow over time, between the host, a guest, and the server, from creating a room to announcing the winner. The arrows that go to both players (with a dot on the guest's lifeline) are the broadcasts that the server sends to everyone in the room.

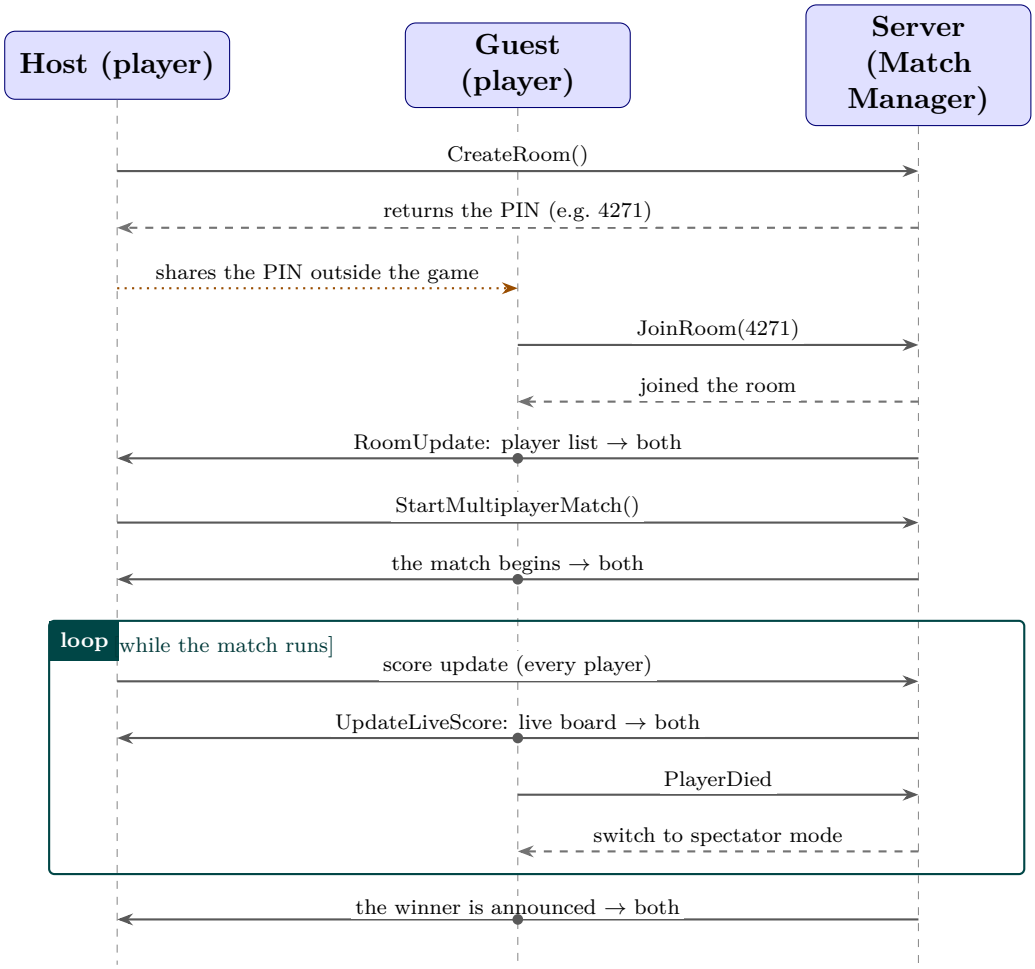


Figure 23: Sequence diagram of a private multiplayer match, from creating the PIN room to the final winner.

13 In Game Data Analytics System

On top of the games themselves, the project includes a full data analytics system that tracks how each game is played and how much money it brings in. All of this data is shown on about fifteen physical boards inside a bank themed building in the game world, turning plain DataStore numbers into clear, live dashboards that any player can walk up to and read. The boards cover the stats for each game, overviews across all the games, the revenue, and live tickers of the most recent scores and donations.

The data flow diagram below shows the whole path: from a player finishing a game or making a donation, through the module that stores and aggregates the data, to the server that serves it, and finally to the boards the player reads.

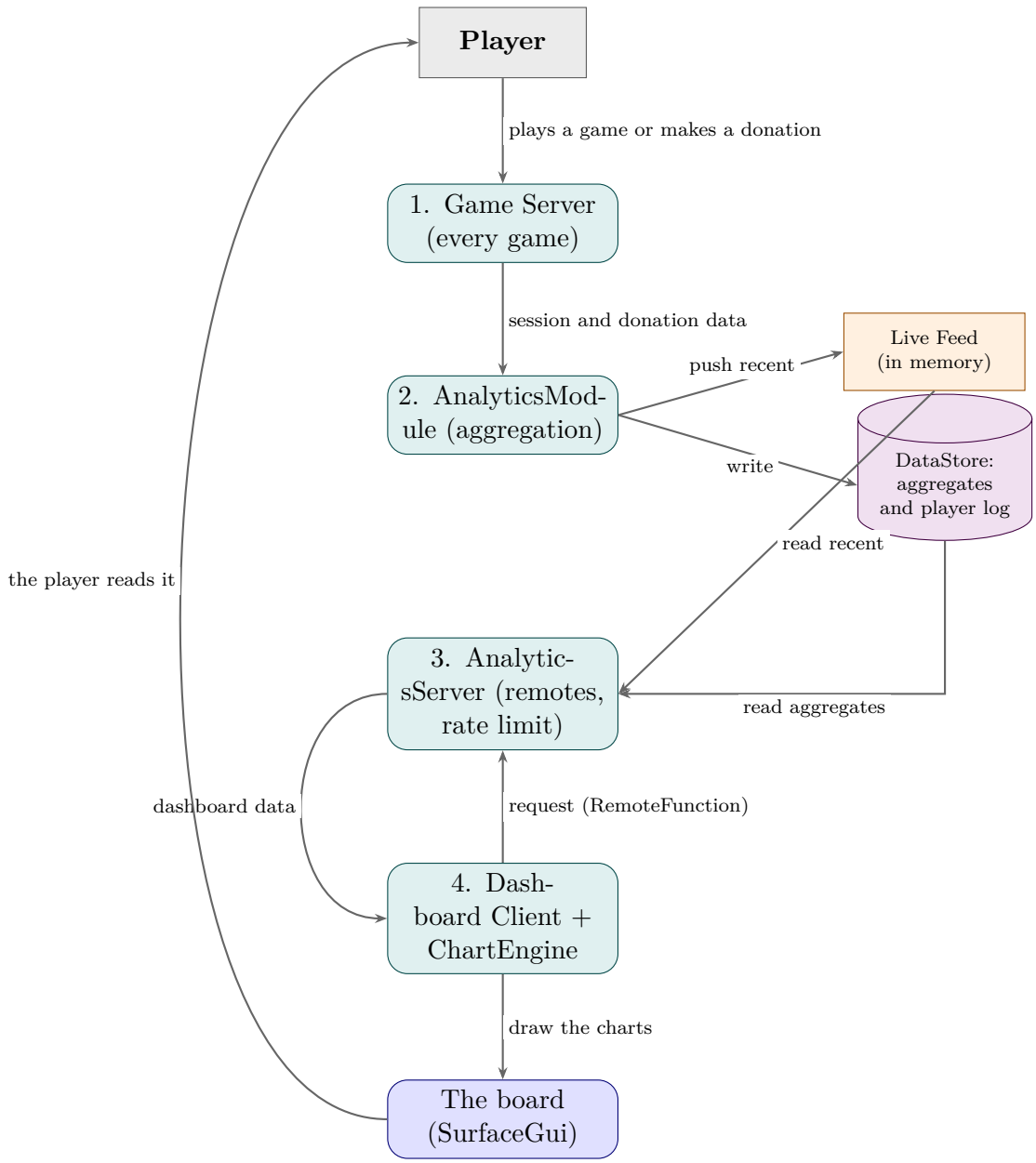


Figure 24: Data flow diagram of the analytics system, from gameplay to the boards on display.

13.1 Architecture

The analytics system is built from four components, following the same client and server separation as the games:

- **AnalyticsModule** (ModuleScript in ReplicatedStorage/Modules): the data layer. It records finished sessions and donations, aggregates them into per game statistics, persists everything to DataStore, and builds the dashboard payloads consumed by the boards.
- **AnalyticsServer** (Script in ServerScriptService): the network layer. It exposes RemoteFunctions for each dashboard type, rate limits requests per player and per board, and broadcasts a periodic `DashboardUpdated` event so open boards refresh themselves.
- **ChartEngine** (ModuleScript in ReplicatedStorage/Modules): the rendering layer. It draws charts (bars, lines, histograms, donuts, and more) directly onto `EditableImage` canvases using a shared dark palette and per game brand colors.
- **AnalyticsDashboardClient** (LocalScript in StarterPlayerScripts): the display layer. It finds every part tagged `AnalyticsBoard` via `CollectionService`, reads its `GameId` attribute to decide which dashboard to render, and draws the board as a `SurfaceGui` on the physical part.

```
Analytics System Structure
|-- Workspace/
|   +-- BankBuilding/ (themed building)
|       +-- ~15 Parts (Tag="AnalyticsBoard", Attribute GameId="...")
|-- ReplicatedStorage/
|   |-- AnalyticsRemotes/ (Folder)
|   |   |-- GetGameDashboard      (RemoteFunction)
|   |   |-- GetGeneralDashboard   (RemoteFunction)
|   |   |-- GetMoneyDashboard     (RemoteFunction)
|   |   |-- GetRecentDonations    (RemoteFunction)
|   |   |-- GetRecentScores       (RemoteFunction)
|   |   +-- DashboardUpdated      (RemoteEvent)
|   +-- Modules/
|       |-- AnalyticsModule       (ModuleScript: data + aggregation)
|       +-- ChartEngine           (ModuleScript: chart rendering)
|-- ServerScriptService/
|   +-- AnalyticsServer           (Script: remotes + rate limiting)
+-- StarterPlayer/
    +-- StarterPlayerScripts/
        +-- AnalyticsDashboardClient (LocalScript: board renderer)
```

13.2 Data Collected Per Game

Every game server already calls two one line helpers at the end of a run: `ReportSessionAsync` after each game over, and `ReportDonationAsync` inside its `ProcessReceipt` handler. These feed the analytics module, which maintains a per game aggregate in the `Analytics_Aggregates_v1` DataStore and a per player log in `Analytics_PlayerLog_v1`. From these, the system derives:

Category	Metrics
Engagement	Total sessions, unique players, replay rate (sessions per player), average and total time played.
Scores	Average score, median score (estimated from a histogram of 20 buckets), max score seen, top 3 scores with player names.
Retention	Players grouped into 5 buckets by session count (1, 2 to 4, 5 to 9, 10 to 19, 20 or more).
Activity	Sessions by hour of day (24 buckets) and by weekday (7 buckets).
Revenue	Total donations (Robux), top 3 donors, revenue per hour and per weekday, revenue per player and per hour played.
Levels	Per level average completion time and max level reached (for games with levels).
Time Leaders	Top 3 players by cumulative time played.
Live Feeds	In memory ring buffers of the 40 most recent scores and donations across all games, newest first.

Table 22: Data collected and derived per game by the analytics module.

To keep DataStore reads low while many boards are open, aggregates are cached for 30 seconds, and writes use `UpdateAsync` so concurrent sessions never overwrite each other.

13.3 Chart Engine

The chart engine renders every visual directly onto an `EditableImage`, so the boards display real pixel drawn charts rather than stacked UI frames. It provides a reusable set of chart types, all themed with a dark palette and the same per game brand colors used elsewhere in the project:

Chart Type	Typical Use
Vertical bar chart	Sessions or revenue compared across games.
Horizontal bar chart	Ranked comparisons (top scores, top donors).
Line chart	Activity or revenue trends across hours / weekdays.
Area chart	Filled trend lines for cumulative metrics.
Histogram	Score distribution across the 20 score buckets.
Scatter plot	Two variable comparisons (e.g. time vs. score).
Donut chart	Proportional breakdowns (retention buckets, revenue share).
Sparkline	Compact inline trend with no axes, used in summary tiles.

Table 23: Chart types provided by the Chart Engine.

13.4 Board Types

Each board is a tagged part whose `GameId` attribute selects which dashboard the client renders on it. There are three families of board: one dedicated dashboard per game, several cross game summary boards, and two live tickers.

Board (GameId)	Content
Per game (×10)	One board per game (Oxybelis, Avis Toucan, Resilio, Foxy Fruit, Firmament Defender, Firmament Siege, Juice Cutter, Turris Casei, Tap And Hold, ImNotASphere) showing that game's full dashboard: engagement, scores, histogram, activity, and revenue.
General	Cross game overview: every game's headline numbers plus combined activity and retention.
Money	Revenue dashboard: donations per game, revenue per hour and per player.
MoneyG	Revenue focused chart variant of the money dashboard.
RealMoney	Live ticker of the most recent donations across all games (newest first).
RealPlayer	Live ticker of the most recent scores across all games (newest first).
LastMessage	Auxiliary board for the latest broadcast message.

Table 24: The board types that make up the bank building (around fifteen boards in total).

13.5 Data Flow

1. A player finishes a game or makes a donation. The game server calls `ReportSessionAsync` or `ReportDonationAsync`.
2. `AnalyticsModule` updates the per game aggregate and per player log in DataStore via `UpdateAsync`, and pushes the event onto the live feed ring buffers.
3. A board exists in the world tagged `AnalyticsBoard`. On spawn, the client reads its `GameId`, renders an empty board immediately, then requests real data through the matching RemoteFunction.
4. `AnalyticsServer` returns the dashboard payload (rate limited per player and per board).
5. The client passes the payload to `ChartEngine`, which draws the charts onto an `EditableImage` shown on the board's `SurfaceGui`.
6. Every 30 seconds the server fires `DashboardUpdated`; open boards refresh. The two live ticker boards poll every few seconds for a near real time feed.

14 Fireworks System

Besides the ten arcade games, the island has a fireworks stand. A player walks up to it, opens a menu of nineteen fireworks and launches one into the sunset sky. Every other player in the server sees the same show at the same time, and the launcher additionally gets a short cinematic camera that chases the shell up and pulls back on the burst.

The system is deliberately split so that the server owns the rules and the client owns the spectacle: the server decides *whether* a launch is allowed and broadcasts *which* firework went off, while each client renders the whole thing locally. Nothing about the visual effect is replicated, which is what keeps a busy sky cheap, and it also means every player can independently choose not to see other people's fireworks.

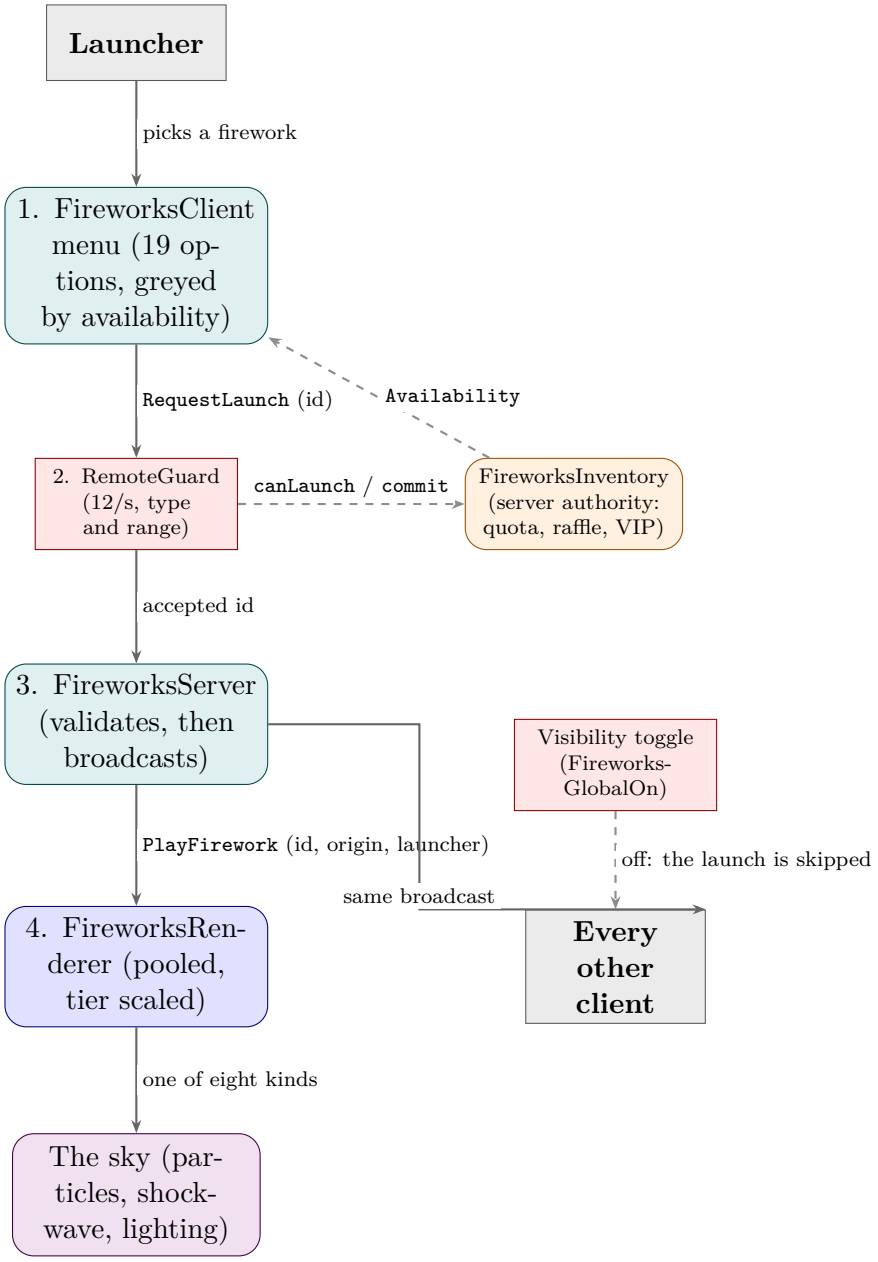


Figure 25: Data flow of the fireworks system, from the launch request to the sky.

14.1 Architecture

The system is made of five scripts plus a shared UI module, following the same client and server separation as the games:

- **FireworksModule** (ModuleScript in ReplicatedStorage/Modules): the data layer. It holds the remote names, the VIP Game Pass id, the two timing constants, and the `Types` table: nineteen definitions that describe every firework purely as data (colours, apex height, particle speed, spread, lifetime, drag, gravity and a `kind` field).
- **FireworksInventory** (ModuleScript in ServerScriptService): the economy layer. It is the single authority on who may launch what. It tracks per player quotas, the free gift, the raffle, the time milestones and the VIP pass, and pushes a per type availability table back to each client so the

menu can grey out what is not currently launchable.

- **FireworksServer** (Script in ServerScriptService): the network layer. It creates the remotes, wires the physical stand button, validates every request through `RemoteGuard` and, once a launch is accepted, broadcasts `PlayFirework` to all clients with the firework id, the world origin and the player who launched it.
- **FireworksClient** (LocalScript in StarterPlayerScripts): the interface layer. It builds the selection menu, handles keyboard and gamepad navigation, shows the toast notifications, drives the cinematic camera for the launcher, and decides whether an incoming broadcast should be rendered at all.
- **FireworksRenderer** (ModuleScript in StarterPlayerScripts): the rendering layer. It turns a firework id into particles. It pools every part it creates, scales particle counts by device tier, and caps how many shows may run at once.
- **ToggleButton** (ModuleScript in StarterPlayerScripts): the shared pill toggle used by the visibility button in the top right corner.

```
Fireworks System Structure
|-- Workspace/
|   |-- FireworksButton        (Part: the stand the player activates)
|   +-- FireworkEffects/       (Folder: every pooled part lives here)
|-- ReplicatedStorage/
|   |-- FireworksRemotes/ (Folder)
|   |   |-- RequestLaunch      (RemoteEvent: client -> server, firework id)
|   |   |-- OpenMenu           (RemoteEvent: server -> client, open the menu)
|   |   |-- PlayFirework       (RemoteEvent: server -> all, id + origin +
   launcher)
|   |   |-- Notify             (RemoteEvent: server -> client, economy messages)
|   |   +-- Availability       (RemoteEvent: server -> client, per type greying)
|   +-- Modules/
|       +-- FireworksModule    (ModuleScript: the 19 definitions + constants)
|-- ServerScriptService/
|   |-- FireworksServer        (Script: remotes, validation, broadcast)
|   |-- FireworksInventory     (ModuleScript: quotas, raffle, VIP)
|   +-- RemoteGuard            (ModuleScript: rate limit + argument checks)
+-- StarterPlayer/
    +-- StarterPlayerScripts/
        |-- FireworksClient    (LocalScript: menu, toasts, cinematic camera)
        |-- FireworksRenderer  (ModuleScript: particle rendering)
        |-- FireworksToggle    (LocalScript: visibility button)
        +-- ToggleButton       (ModuleScript: shared pill toggle UI)
```

14.2 The Launch Economy

Launching is not free, and the decision is never taken on the client. Every request passes through `FireworksInventory.canLaunch` and, if accepted, `commit`. The rules separate the sixteen normal fireworks (ids 1 to 16) from the three big shows (ids 17 to 19):

Rule	Effect
Normal allowance	Three launches of any firework from 1 to 16 per minute, for everybody.
Free gift	One extra firework per minute is added to a personal inventory, which caps at eight so it cannot be hoarded forever.
Raffle	Once per minute the player gets a one in one hundred chance of winning three uses of a big show.
Eight minute milestone	At eight minutes of play the player is granted two big launches. This is the reward that ties into the eight minute retention goal of the project.
Fifteen minute milestone	At fifteen minutes the player unlocks all access: six launches of anything, which expires fifteen minutes later.
VIP Game Pass	Owners get three free launches of any firework every three minutes, on top of everything above. Normal players spend their free quota first so VIP credit is saved for the big shows.
Rate limit	`RemoteGuard` allows at most twelve requests per second per player and validates the argument type and range before the handler ever runs.
Big cooldown	A global `BigCooldown` of 1.5 seconds between big shows, plus a `MinInterval` of 0.1 seconds between any two launches.

Table 25: Server side rules that govern who may launch which firework.

Whenever any of these values change, the server fires `Availability` with a table of booleans keyed by firework id. The client stores it and greys out the unavailable entries in the menu, so the player sees the economy without the client ever being trusted to enforce it.

14.3 Rendering Pipeline

The renderer never creates and destroys parts during a show. Two pools, one for burst emitters and one for rising shells, hand out parts and take them back, so a sky full of fireworks does not thrash the garbage collector. A third small pool serves the shockwave spheres and a fourth the ring segments.

Everything is scaled by device tier, detected once at startup:

Setting	PC	Console	Mobile	What it controls
Particle quality	1.00	0.72	0.45	Multiplier on every particle count.
Concurrent normal shows	18	18	8	Extra launches are silently dropped past this.
Concurrent big shows	2	2	1	Extra ones degrade to a single large burst.
Dynamic lights	yes	yes	no	Per burst `PointLight`. Neon already self glows, so phones skip it.
Ember layer	yes	yes	no	The lingering twinkle layer described below.

Table 26: Device tiers. The renderer reads the tier once and every later decision follows from it.

Three global constants tune the look of all nineteen fireworks at once: `APEX_SCALE` (how high they climb), `WIDTH_SCALE` (how wide the bursts open) and `SIZE_SCALE` (how thick the sparks are).

14.3.1 The Three Layer Burst

A single `Emit` call produces flat coloured confetti, so every burst is built from three emitters that split the same particle budget rather than adding to it:

1. **Ignition flash** (15 percent of the budget): a dense, almost white core with a very short lifetime and heavy drag. It reads as the moment of detonation.
2. **Coloured petals** (70 percent): the layer that gives each firework its recognisable shape, driven by the definition's speed, spread, drag, gravity and squash.
3. **Embers** (15 percent, skipped on mobile): slow, small, strobing sparks that live almost twice as long as the petals and hang in the air after the shape has gone.

Two further details do most of the work for free. Each particle's colour follows a curve that starts near white and cools into the firework's palette, mimicking how a real star burns white at ignition and only shows its colour as it cools. And each burst also spawns a single expanding Neon sphere as a shockwave, which costs one part and reads as the concussion of the blast. Fireworks with a `pistil` get a fourth emitter for the bright inner flower, on its own emitter on purpose: Roblox reads emitter properties live for particles already in flight, so reusing the core emitter would repaint the ignition flash mid burn.

The rising shell is a Neon ball with a `Trail` plus a small crackle emitter that sheds sparks during the climb. Definitions can set `silent = true` in their `ascent` table to suppress those sparks, which is what gives the slow, quiet fireworks their character.

14.4 Firework Kinds

The nineteen fireworks are data, not code. All of them route through a single dispatcher that reads the `kind` field and picks one of eight behaviours. This is where the actual logic of each firework lives.

Kind	Behaviour
`burst`	The default. A shell rises to its apex, then a single three layer burst opens. Eleven of the nineteen use it, and they differ only in their numbers.
`crossette`	The shell opens into a handful of large stars, and 0.32 seconds later each of those detonates again in a small secondary burst scattered around the first. That delay is what produces the cross grid.
`ring`	A normal burst plus a ring of segments tweened outward from the centre along a tilted plane, so the sphere ends up wrapped by a halo.
`mine`	No shell at all. The emitter fires from the ground upward in a narrow cone, producing a volcano fountain instead of an aerial shape.
`salute`	A shell rises and detonates with almost no fall: very short lifetime, heavy drag and a blue screen flash. It is a bang and a flash, not a shape.
`comet`	A single huge Neon head crosses the sky diagonally over 4.2 seconds, trailing a wide glowing tail and shedding stardust. The sky darkens for the crossing, and where it lands it triggers a shower of twenty two ordinary bursts.
`mega`	A very high shell, then a titanic central detonation followed by a dome of thirty sub fireworks scattered over a wide radius and staggered over 2.6 seconds.
`explosion`	A sixteen second show in three phases: a charge up orb that pulls energy inward for 2.2 seconds, a colossal blast with three expanding shock rings, then a sustained cascade of random bursts across a wide area.

Table 27: The eight rendering behaviours. A firework's `kind` field selects one of them.

The three big kinds (`comet`, `mega` and `explosion`) also darken the sky while they run. They do it by

lowering `Brightness`, pushing `ClockTime` to night and dimming the ambient colours, never by touching exposure, because exposure would dim the Neon particles too. Neon self illuminates, so the fireworks stay bright against a dark sky. While the override is active the renderer sets a `FireworkOverride` attribute on `Lighting`; the island lighting script watches for it and stops forcing the sunset until it clears, which makes the handoff seamless in both directions.

14.5 The Nineteen Fireworks

The table below is the complete `Types` table. Reading it alongside the kind descriptions above explains every firework: the shape comes from the interaction of spread, drag and gravity, the texture comes from squash, and the density comes from count.

Id	Name	Kind	Apex	Spd	Spr	Life	Size	Grav	Drag	Cnt
1	Peony	burst	145	62	175	1.9	1.8	-24	2.5	160
2	Chrysanthemum	burst	155	55	170	3.0	1.5	-45	2.0	160
3	Willow	burst	170	44	160	3.8	1.4	-78	4.5	150
4	Palm	burst	155	74	150	2.8	3.0	-34	1.3	80
5	Brocade	burst	165	52	175	3.8	1.8	-55	3.5	190
6	Crossette	crossette	140	50	150	1.3	2.0	-20	1.0	9
7	Fish / Bees	burst	135	36	180	2.2	1.2	-10	1.5	160
8	Kamuro	burst	180	40	165	4.2	1.3	-85	5.0	160
9	Dahlia	burst	145	82	150	2.3	3.4	-40	0.8	50
10	Horsetail	burst	165	18	70	3.2	1.4	-95	2.0	130
11	Strobe	burst	145	30	180	3.0	1.0	-14	2.5	180
12	Spider	burst	135	95	160	1.1	1.3	-8	0.2	80
13	Saturn	ring	150	48	175	1.9	1.6	-22	2.0	130
14	Titanium Salute	salute	140	fixed in the renderer: flash, no fall						
15	Mine	mine	–	88	45	1.6	1.6	-60	1.0	240
16	Ghost Shell	burst	150	52	175	2.6	1.6	-26	2.0	160
17	Tony	comet	choreographed: 4.2 s crossing, then 22 bursts							
18	Steamboat Springs	mega	240	choreographed: central blast, then a 30 shell dome						
19	EXPLOSION	explosion	150	choreographed: 16 s, charge orb then cascade						

Table 28: The complete definition table. Speed, spread, lifetime, size, gravity, drag and count are the particle parameters; the global scale constants are applied on top.

14.5.1 Why Each One Looks Different

- **Peony** is the reference shape: the widest spread, no squash and moderate gravity give a clean symmetric sphere. Every other burst is a deviation from it.
- **Chrysanthemum** keeps the sphere but adds squash 4, which stretches each spark into a streak, and heavier gravity so the petals curve downward as they fall.
- **Willow** trades speed for weight: the slowest of the wide bursts, the highest drag of the group and strong gravity, so the threads barely spread and then droop. Its ascent is silent, which suits the slow mood.
- **Palm** halves the particle count and nearly doubles the spark size, with low drag so the few fronds fly far. It is the only firework with a `pistil`, the bright inner flower on its own emitter.
- **Brocade** has the highest count of all (190) and a long lifetime, producing a dense glittering crown that lingers.
- **Crossette** emits only nine stars, which is the whole point: they have to be individually visible, because each one detonates again 0.32 seconds later.

- **Fish / Bees** has almost no gravity (-10) and the `erratic` flag, which gives every spark a random rotation speed. The result swims instead of falling.
- **Kamuro** is the extreme of the Willow idea: the highest apex, the highest drag, the longest lifetime and the strongest squash, for an ultra long falling curtain.
- **Dahlia** is the opposite extreme: the fewest particles, the largest sparks and the lowest drag, so a handful of huge stars shoot out in straight diagonals.
- **Horsetail** narrows the spread from 175 to 70 and uses the heaviest gravity in the set, so instead of a sphere it pours straight down as a waterfall.
- **Strobe** uses the `strobe` flag, which swaps the normal fade curve for an alternating visible and invisible sequence. With low gravity the sparks hang and blink like fireflies.
- **Spider** is the fastest (95) with almost no drag (0.2) and the shortest lifetime, so the legs shoot out dead straight and stop. Nothing has time to curve.
- **Saturn** is a Peony wrapped in a ring tilted 28 degrees, drawn in a contrasting violet so the halo reads against the blue sphere.
- **Titanium Salute** has no shape at all by design. The renderer overrides the particle values with a very short lifetime and heavy drag, and adds a blue screen flash instead of a white one.
- **Mine** skips the shell entirely and fires from the ground, with the second highest count (240) through a narrow 45 degree cone.
- **Ghost Shell** is a Peony with the `colorShift` flag: 0.25 seconds after the first burst a second wave opens in the last colour of the palette, so the shape appears to morph from green to violet.
- **Tony** is the comet. It is the only firework that crosses the sky rather than rising and falling, and the only one whose payload is other fireworks.
- **Steamboat Springs** is the finale: the highest apex in the project and a dome of thirty sub fireworks drawn from the more delicate burst types.
- **EXPLOSION** is the longest at sixteen seconds and the only one with a charge up phase, which exists purely to build anticipation before the blast.

14.6 Cinematic Camera

Only the player who launched a firework gets a camera move; everyone else keeps normal control. The camera follows the rising shell, then tweens back to frame the burst, and always returns to the player within a fixed budget derived from the firework kind, so it can never strand the player looking at the sky. Sky wide kinds (comet, mega, explosion) use a low ground camera that pans to follow the show, while ordinary bursts chase the shell from a close offset and pull back on detonation. Other players instead get a small toast in the corner naming who launched what.

14.7 Visibility Toggle

A busy server can fill the sky, and the big shows deliberately darken it, so each player can opt out of other people's fireworks. A pill button in the top right corner toggles a `FireworksGlobalOn` attribute

on the player. The button is built from the shared `ToggleButton` module: it turns grey when off, is reachable by mouse, touch, the F key and the right stick click on a gamepad, and scales its own size so the touch target stays usable on phones.

When the toggle is off, the client returns early from the `PlayFirework` handler for any launch that is not its own. That single early return happens before the renderer is called, so it suppresses the particles, the toast and the sky darkening all at once, at zero cost. The player's own fireworks always play, cinematic camera included, because the point is to mute other people rather than to lose the feature.