Design and Implementation of a 2D Action Platformer Game Using Godot

by

Md. Mahmudul Hasan Rimon
ID: CSE2103024077

Muminur Rahman
ID: CSE2103024081

Supervised by
Khadija Islam

Submitted in partial fulfillment of the requirements for the degree of
Bachelor of Science in Computer Science and Engineering



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
SONARGAON UNIVERSITY (SU)

September 2025

APPROVAL

The project titled "**Design and Implementation of a 2D Action Platformer Game Using Godot**" submitted by Md. Mahmudul Hasan Rimon (CSE2103024077), Muminur Rahman (CSE2103024081) to the Department of Computer Science and Engineering, Sonargaon University (SU), has been accepted as satisfactory for the partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Science and Engineering and approved as to its style and contents.

Board of Examiners

Khadija Islam
Lecturer
Department of Computer Science and Engineering
Sonargaon University (SU)

Supervisor

(Examiner Name and Signature)
Department of Computer Science and Engineering
Sonargaon University (SU)

Examiner 1

(Examiner Name and Signature)
Department of Computer Science and Engineering
Sonargaon University (SU)

Examiner 2

(Examiner Name and Signature)
Department of Computer Science and Engineering
Sonargaon University (SU)

Examiner 3

DECLARATION

We, hereby, declare that the work presented in this report is the outcome of the investigation performed by us under the supervision of **Khadija Islam, Lecturer,** Department of Computer Science and Engineering, Sonargaon University, Dhaka, Bangladesh. We reaffirm that no part of this project has been or is being submitted elsewhere for the award of any degree or diploma.

Countersigned

(Khadija Islam)
Supervisor

Signature

Md. Mahmudul Hasan Rimon
ID: CSE2103024077

Muminur Rahman
ID: CSE2103024081

ABSTRACT

This project showcases the development of a 2D Action Platformer game utilizing the open-source Godot engine and freely available assets. The game incorporates advanced features, including dynamic combat and lighting, and visual shaders across three distinct levels. It shows how the storyline, art, animation, programming, etc., can be combined to offer a complete package and explores the vital role game engines play in the game development pipeline. Through the application of the Action Platformer subgenre, the project demonstrates the fundamentals of game design, namely environment design, player interaction, weapon mechanics, lighting, sound, and platforming systems. Incorporating evolved game development techniques such as visual scripting, a fully functional prototype was developed, showcasing the application of game design principles in creating an engaging, playable game. Challenges in integrating assets and programming-related optimization revealed important insights for independent game developers. Ultimately, the project emphasizes the potential for independent games to be developed successfully using existing assets.

ACKNOWLEDGMENT

At the very beginning, we would like to express our deepest gratitude to the Almighty Allah for giving us the ability and the strength to finish the task successfully within the schedule time.

We are auspicious that we had the kind association as well as supervision of **Khadija Islam**, Lecturer, Department of Computer Science and Engineering, Sonargaon University whose hearted and valuable support with best concern and direction acted as necessary recourse to carry out our project.

We are particularly grateful to **Prof. Bulbul Ahamed**, Head of the Department, Computer Science and Engineering, Sonargaon University, for his kind concern and precious suggestions.

We are also thankful to all our teachers during our whole education, for exposing us to the beauty of learning.

Finally, our deepest gratitude and love to our parents for their support, encouragement, and endless love.

LIST OF ABBREVIATIONS

2D	Two-dimensional
2.5D	Two-and-a-half-dimensional
3D	Three-dimensional
AI	Artificial Intelligence
AR	Augmented Reality
CPU	Central Processing Unit
DFD	Data Flow Diagram
FPS	First-person Shooter
GPU	Graphics Processing Unit
GUI	Graphical User Interface
RPG	Role-playing Game
UI	User Interface
VR	Virtual Reality

TABLE OF CONTENTS

Title			Page No.
DECLARATION …….……..			iii
ABSTRACT ……...……..			iv
ACKNOWLEDGEMENT ……..……			v
LIST OF ABBREVIATION ……..……..			vi
CHAPTER 1			1 – 5
INTRODUCTION TO 2D GAMES			
1.1	Introduction ……..…….....		1
1.2	Objectives ……...…….....		1
1.3	Features of a 2D Game ……..…….....		1 – 3
	1.3.1	Game Design …….	2
	1.3.2	Art and Animation …….	2
	1.3.3	Programming and Game Engines ……..	2
	1.3.4	Level Design …….	2
	1.3.5	Feedback Systems …….	2
	1.3.6	Narrative Integration …….	2
1.4	Why Use Godot? …….…….....		3
1.5	Features of Godot ……...…….....		3 – 5
	1.5.1	Accessibility ……..	4
	1.5.2	Scripting Languages ……..	4
	1.5.3	Community ……...	4
	1.5.4	Assets ……	4
	1.5.5	Cross-platform Support …….	4
	1.5.6	Licensing ……...	4
	1.5.7	Customization ……...	4
	1.5.8	2D Game Development Proficiency ……..	4
	1.5.9	Learning Resources ……...	5
	1.5.10	Target Audience ……	5
CHAPTER 2			6 – 11
PLATFORMER GAMES			

2.1 Introduction ……..... 6
2.2 Types of Platformer Games ……. 7 – 11
2.2.1 Action Platformer …….. 7
2.2.2 Precision Platformer …….. 7 – 8
2.2.3 Puzzle Platformer …….. 8
2.2.4 3D Platformer …… 9
2.2.5 Cinematic Platformer …… 9 – 10
2.2.6 Platform Fighter …… 10
2.2.7 Auto-Runner …… 11
2.3 Why Action Platformer? ……. 11

CHAPTER 3 12 – 20

BASIC CONCEPTS OF GODOT

3.1 Introduction …….. 12
3.2 Scenes …….. 12 – 13
3.3 Nodes ……... 13 – 18
3.3.1 Node2D ……. 14
3.3.2 Sprite2D …… 15
3.3.3 AnimatedSprite2D …… 15
3.3.4 AnimationPlayer ……... 15
3.3.5 AnimationTree …….. 15
3.3.6 Area2D …….. 15
3.3.7 AudioStreamPlayer ……... 16
3.3.8 Control …….. 16
3.3.9 Button ……... 16
3.3.10 Camera2D ……. 16
3.3.11 CanvasItem ……... 16 – 17
3.3.12 CanvasLayer ……. 17
3.3.13 CharacterBody2D ……. 17
3.3.14 CollisionShape2D ……. 17
3.3.15 ColorRect …….. 17
3.3.16 GPUParticles2D …… 17
3.3.17 ParallaxBackground …….. 18
3.3.18 Timer ……. 18

3.3.19 VBoxContainer ……. 18
3.4 Signals …….. 18 – 19
3.5 GDScript …….. 19 – 20

CHAPTER 4 21 – 45
GAME DESIGN
4.1 Flowchart ……. 21 – 23
4.2 Data Flow Diagram …….. 24 – 25
4.2.1 Level 0 DFD ……. 24
4.2.2 Level 1 DFD ……. 24 – 25
4.3 Use Case Diagram ……... 25 – 26
4.4 Environment Design ..……..... 26 – 30
4.4.1 Tilesets …….. 26 - 28
4.4.2 Station Environment ……. 28 – 29
4.4.3 Sky Environment …….. 29
4.4.4 Underground Environment ……... 29 – 30
4.5 Player Mechanics ……. 30 – 34
4.5.1 Walking ……...…….. 30
4.5.2 Jumping ……..……... 30 – 32
4.5.3 Dashing ……. 32 – 33
4.5.4 Ducking ……. 33 – 34
4.6 Enemy AI Behaviors …….…….…..... 34 – 38
4.6.1 Soldier ……... 34 – 35
4.6.2 Drone …… 35 – 36
4.6.3 Saw ……... 36 – 38
4.7 Boss Fight Design …….……... 38 – 39
4.8 Weapon System …..…….…..……... 39 – 41
4.8.1 Aim Direction ……..... 39 – 40
4.8.2 Crosshair Placement …….. 40 – 41
4.8.3 Weapon Options ...……. 41
4.9 Lighting System ...…….…..... 41 – 43
4.10 Sound System …….. 43
4.11 Health Circle ……...……. 44
4.12 Transition Gates ……... 45

CHAPTER 5 46 – 67

DEVELOPMENT AND TESTING

5.1 Implementation Details ……..……... 46 – 62

5.1.1 Global Settings ……. 46

5.1.2 Main Menu ……... 46 – 47

5.1.3 Player Animation ……. 47 – 48

5.1.4 Player Movement ……. 49 – 50

5.1.5 Weapon Shooting ……. 50

5.1.6 Flash Shader ……. 50 – 51

5.1.7 Weapon Selection ……. 52

5.1.8 Weapon Animations ……. 52

5.1.9 Health Circle Shader …… 53

5.1.10 Soldier Engagement ……. 53 – 54

5.1.11 Drone Engagement ……... 54 – 55

5.1.12 Saw Engagement …….. 55

5.1.13 Monster Engagement ……... 55 – 56

5.1.14 Bullet Firing ……. 57

5.1.15 Rocket Explosion ……. 57 – 58

5.1.16 Transition Gate ……. 58

5.1.17 Camera Shaking Effect ……. 58 – 60

5.1.18 Damage Particle Effect ……. 60 – 62

5.2 Challenges Faced ……..... 62

5.2.1 Visual Node Selection ……... 62

5.2.2 Node Interconnection …… 62

5.2.3 Animation Timing ……. 62

5.2.4 Signal Integrity …….. 62

5.3 Testing and Results ……..……..... 62 – 67

5.3.1 Test Table …….. 62 – 63

5.3.2 Output ……... 63 – 67

CHAPTER 6 68 – 70

CONCLUSION AND FUTURE WORKS

6.1 Limitations ……..…. 68

6.1.1 Development Constraints …….. 68
6.1.2 Visual Fidelity ……... 68
6.1.3 Simplified Visualization …… 68
6.1.4 Restrictive Player Movement …… 68
6.1.5 Limited Playtime ……... 68
6.1.6 Lack of Advanced Features ……... 68
6.2 Conclusion ……..…. 68 – 69
6.3 Future Works …… 69 – 70
6.3.1 Story-driven Approach …….. 69
6.3.2 Quest Systems ……... 69
6.3.3 Power-ups ……. 69
6.3.4 Multiple Character Implementation …….. 69
6.3.5 More Content …… 69
6.3.6 Particle Effects …….. 69
6.3.7 Sound Design …… 70
6.3.8 Enhanced Features …… 70
6.3.9 Visual Improvements …… 70
6.3.10 Cross-Platform Compatibility ……... 70

REFERENCES …….…….. 71

LIST OF TABLES

Table No.	Title	Page No.
Table 1.1	Comparison Between 2D and 3D Games	2 – 3
Table 4.1	Weapon Types and Characteristics	41
Table 5.1	Test Table	62 – 63

LIST OF FIGURES

Figure No.	Title	Page No.
Fig.2.1	Celeste (2018) – Example of a Platformer	6
Fig.2.2	Hollow Knight (2017) – Example of an Action Platformer	7
Fig.2.3	Geometry Dash (2013) – Example of a Precision Platformer	8
Fig.2.4	Fez (2012) – Example of a Puzzle Platformer	8
Fig.2.5	Super Mario 64 (1996) – Example of a 3D Platformer	9
Fig.2.6	Limbo (2010) – Example of a Cinematic Platformer	10
Fig.2.7	Brawlhalla (2014) – Example of a Platform Fighter	10
Fig.2.8	Temple Run (2011) – Example of an Auto-Runner	11
Fig.3.1	File Structure of Our Project, Showcasing Resource Folders for Audio, Graphics, Levels, Projectiles, etc.	12
Fig.3.2	Overview of Major Scenes in Our Project, Including Player, Drone, Monster, and Level Scenes	13
Fig.3.3	Internal Components of the “Player” Scene, Showing Core Components Such as Camera2D, CollisionShape2D, and Timers	14
Fig.3.4	Signal Connections from the AnimationSprite2D Node Used to Trigger Events Like Animation Completion	19
Fig.3.5	Sample GDScript Code Demonstrating Input Handling in Our Game	
Fig.4.1	Flowchart of the Station Level, Detailing Enemy Engagements, Player Input, and Transition to Other Levels	21
Fig.4.2	Flowchart of the Underground Level, Outlining the Player and the Soldier’s Actions and Transitions to Other Levels	22
Fig.4.3	Flowchart of the Sky Scene, Outlining Player’s Engagement with the Monster	23
Fig.4.4	Level 0 DFD, Showcasing the Game as a Single Process Interacting with External Entities	24
Fig.4.5	Level 1 DFD, Breaking Down the Game into Subprocesses Such as Player Input, Game Physics, and Graphics	25
Fig.4.6	Use Case Diagram Illustrating Interaction Between the Player and Core Game Features Including Movement, Weapon Switching, and Menu Navigation	26
Fig.4.7	Outside Tileset, Composed of Ground, Walls, and Decorative Elements	27
Fig.4.8	Inside Tileset, Featuring Platforms for the Indoor Sections of the Station Level	27
Fig.4.9	Wall Tileset Used to Build Boundaries	28

Fig.4.10	Fully Assembled Station Level, Featuring Transition Gates to Other Levels and Enemy Entities Such as the Soldier, Drone, and Saw	28 – 29
Fig.4.11	Sky Level Featuring the Monster entity, the Final Boss of Our Game	29
Fig.4.12	Underground Scene, Featuring Multiple Soldier Entities for Moderate Difficulty	30
Fig.4.13	Visual Representation of the Protagonist's Regular Jump Height	31
Fig.4.14	Visual Representation of the Extended Jump Height Achieved by Long-Pressing the "Spacebar"	31
Fig.4.15	Player Character Jumping Off a Cliff	32
Fig.4.16	Demonstration of Coyote Time – A Brief Period After Jumping Off a Cliff During Which the Player Can Still Jump	32
Fig.4.17	Player Character in a Jump State Before Dashing	33
Fig.4.18	Player Character's Position After Performing the Dashing Movement	33
Fig.4.19	Protagonist Under Enemy Fire, Ducking Bullets	34
Fig.4.20	Programming Logic Showcasing the Distance Check Between the Soldier and the Player	34
Fig.4.21	Circular Area2D Node Defining the Drone's Activation Area	35
Fig.4.22	Stationary Drone Before Detecting the Player	36
Fig.4.23	Drone Following the Player After Detection	36
Fig.4.24	Predefined Linear Path Followed by the Saw Entity	37
Fig.4.25	Saw Moving Along Predefined Path Part 1	37
Fig.4.26	Saw Moving Along Predefined Path Part 2	38
Fig.4.27	Monster Entity Alongside 8 Possible Bullet Release Points	39
Fig.4.28	Programming Logic Showcasing the Monster's Attack System	39
Fig.4.29	8-way Aiming System for Shooting Weapons	40
Fig.4.30	Crosshair Placement in Normal Stance	40
Fig.4.31	Adjusted Crosshair Placement While Ducking	41
Fig.4.32	Light Type 1	42
Fig.4.33	Light Type 2	42
Fig.4.34	Light Type 3	42
Fig.4.35	Light Type 4	42
Fig.4.36	Light Type 5	43
Fig.4.37	Node Configuration for Playing Explosion Sounds Using AudioStreamPlayer	43
Fig.4.38	Health Circle Design	44

Fig.4.39	Visual Representation of Health Circle Reduction After Player is Hit	44
Fig.4.40	Transition Gate Used to Move Between Levels	45
Fig.5.1	Globally Defined Parameters, Accessed by Multiple Scripts Throughout the Project	46
Fig.5.2	Main Menu Setup, Featuring Buttons for “Play”, “Quit”, and “Settings”	47
Fig.5.3	Signals from the AnimationPlayer Node, Used for Communicating to Other Scenes/Nodes About Changes in Animations	48
Fig.5.4	Godot Animation Tab, Used for Generating Frame-by-frame Animation	48
Fig.5.5	Programming Logic Showcasing the Player Movement, Including Directional Movement, Jumping, and Dashing	49
Fig.5.6	Programming Logic Defining the Weapon Shooting Process, Including Weapon Selection, and Reload Times	50
Fig.5.7	Flash Shader Used to Produce a Flash Effect Whenever the Player is Shot	51
Fig.5.8	Flash Shader in Action, Visualizing the Flash Effect	51
Fig.5.9	AnimationTree Node Used for Weapon Selection	52
Fig.5.10	Programming Logic for the Switching of Weapons	52
Fig.5.11	AnimationTree Node, Used for Defining the Points Where the Animation is Played	52
Fig.5.12	Programming Logic for the Health Circle Shader	53
Fig.5.13	Programming Logic for the Player’s Interactions with the Soldier Entity	54
Fig.5.14	Programming Logic for the Player’s Interaction with the Drone Entity	55
Fig.5.15	Programming Logic for the Player’s Interaction with the Saw Entity	55
Fig.5.16	Programming Logic for the Player’s Interaction with the Final Boss	56
Fig.5.17	Programming Logic for Handling the Firing of Bullets	57
Fig.5.18	“SpriteFrames” Setup for Rocket Explosion Animation	58
Fig.5.19	Programming Logic for Transition Gates	58
Fig.5.20	Programming Logic for the Camera Shaking Effect Part 1	59
Fig.5.21	Programming Logic for the Camera Shaking Effect Part 2	59
Fig.5.22	Programming Logic for the Camera Shaking Effect Part 3	59
Fig.5.23	Programming Logic for the Camera Shaking Effect Part 4	60
Fig.5.24	The Damage Particle Effect in Action, Particles Emit from the Soldier’s Body After Being Hit	60
Fig.5.25	Programming Logic for the Damage Particle Effect Part 1	60
Fig.5.26	Programming Logic for the Damage Particle Effect Part 2	61

Fig.5.27	Inspector Section of GPUParticles2D, Featuring Properties Such as Particle Amount, Spawn Position, Gravity	61
Fig.5.28	Soldier Showing Damage Points After Being Hit by the Player	64
Fig.5.29	Explosion Animation After the Player Hits the Soldier Using the Shotgun	64
Fig.5.30	The Player Hits the Drone Using the Rocket Launcher	65
Fig.5.31	Visualization of the Health Circle Reduction with Flash Effect After Player is Hit by the Soldier	65
Fig.5.32	Visualization of the Player Losing Health After Getting Hit by the Saw	66
Fig.5.33	Player Retreats to the Safe Zone When Fighting the Monster Entity	66
Fig.5.34	Death Animation of the Monster Entity is Shown After Being Killed by the Player	67

CHAPTER 1

INTRODUCTION TO 2D GAMES

1.1 Introduction

A 2D game uses a two-dimensional coordinate system to visualize the gameplay experience. It uses sprites, which are two-dimensional images that can be animated and moved. Sprites serve as the means of animating the characters, backgrounds, UI elements, enemies, etc. [1]. Examples of 2D games include Action Platformers, Card games, Role-playing Games, Tower Defense games, and Puzzle games [2].

2D games offer many advantages, such as having lower hardware requirements, simpler mechanics, and faster gameplay. The assets used in 2D games that display the characters, environments, and enemies require fewer resources to render. This makes developing 2D games less expensive and a more promising endeavor for independent game developers and small studios that have fewer resources [3].

However, the limitations of 2D games include their lack of visual depth and realistic graphics. Therefore, they are not as graphically immersive as 3D games. In certain genres of video games, such as Action, Adventure, and Racing, the demand for 3D tends to be higher. On a technical level, 2D games offer fewer functionalities regarding physics implementations, camera movement, AI, etc. [4]

1.2 Objectives

1. To develop a 2D game specializing in the Action Platformer subgenre [8].
2. Utilizing free assets and open-source tools for cost-effective development.
3. Explore the landscape of independent game development using the Godot game engine.
4. Explore visual scripting and node-based game design.
5. Implement advanced features such as visual shaders, dynamic combat, and lighting.
6. Test and evaluate gameplay mechanics and player experience.

1.3 Features of a 2D Game

A well-crafted 2D game integrates multiple components ranging from the storyline, design, art, programming, and animation, which collectively shape the Player's experience. This section explores the key features that define 2D games, including game design, art and

animation, programming and engines, level design, etc., highlighting how these elements work together to create compelling and immersive gameplay.

1.3.1 Game Design

Game Design encompasses the mechanics, storyline, characters, and the overall gameplay experience. For adhering to the vision of the game, developers follow a reference named the Game Design Document or GDD [5].

1.3.2 Art and Animation

Visual fidelity plays a crucial role in 2D games. Pixel art, hand-drawn artwork, or vector artwork is used to make the game visually appealing. Adobe Photoshop and Aseprite are popular choices for making sprites and animations [5].

1.3.3 Programming and Game Engines

Programming serves as the brain of every game. 2D game developers commonly use engines like Unity, Godot, and GameMaker Studio. Amongst them, Unity is known for offering a sturdy foundation for game development in both 2D and 3D [6]. Godot is becoming an increasingly popular choice for 2D games due to its free and open-source nature, strong community-driven approach, extensive documentation, and learning resources.

1.3.4 Level Design

Level Design refers to the obstacles, rewards, and the tempo of the game that serve the purpose of gaining and maintaining the interest of the Player [5].

1.3.5 Feedback Systems

Feedback Systems enhance the gameplay experience by providing visual signals, sound effects, and haptic feedback that help Players learn how to become better at the game [5].

1.3.6 Narrative Integration

Storytelling is a useful tool in games to increase Player immersion. Narrative components like dialogue systems can help foster an emotional bond with the Player [5].

Features	2D Games	3D Games
Gameplay	Simpler, more condensed.	Complex with significantly more moving parts.
Graphical Fidelity	Less detailed 2D images and sprites.	More detailed 3D models with realistic lighting.

Viewing Perspective	Fixed, or side-scrolling.	Dynamic with 3D camera angles.
Game Movement	Horizontal or vertical.	Able to move in all directions.
Game Complexity	Simpler by design.	More complex due to game mechanics, controls, and motion physics.
Development Time and Cost	Faster development with reduced cost.	Slower development path due to increased complexity and higher development cost due to increased resource usage.
Art Style	Pixel art, hand-drawn, or isometric.	Different types of 3D art (photorealistic, stylized).
Player Interaction	Limited, often only using the keyboard.	Limitless potential due to VR capabilities, advanced AI, and multiplayer component.
Player Immersion	Lower due to inferior graphical fidelity.	Higher due to realistic environments and sounds.
System Requirements	Lower system requirements. It can run on basically any device.	Higher system requirements. More taxing on the GPU and the CPU.
Target Audience	Casual gamers, indie gamers. Appeals to players who value gameplay over graphical fidelity.	Appeals to audiences who want immersion, realism, exploration, cinematic storytelling.
Common Genres	Platformers, Metroidvania, Arena Fighting, Roguelike, RPGs.	FPS, Open-world Adventure, Simulation, Racing, Sports.
Examples	Celeste, Hollow Knight, Stardew Valley	Red Dead Redemption 2, The Witcher 3, Grand Theft Auto V

Table 1.1: Comparison Between 2D and 3D Games

1.4 Why Use Godot?

Godot has become popular over the years for being a simple, effective, and versatile open-source game engine. It supports both 2D and 3D game development. Features such as a visual scripting system, community-made tools, scripts, and forums are what make Godot special to many game developers. On top of it all, it doesn't have a licensing fee, which makes it a plus point for indie game developers and small studios that have a limited budget [6].

1.5 Features of Godot

The distinguishing features of the Godot game engine are discussed in this section, including its user-friendly nature, cross-platform compatibility, exceptional support for 2D

games, and free licensing. These features make Godot an excellent choice for 2D game developers.

1.5.1 Accessibility

The Godot game engine features an intuitive, user-friendly UI that developers can pick up easily. The learning curve is relatively easier compared to other game engines.

1.5.2 Scripting Languages

Godot supports multiple languages apart from its own built-in scripting language, known as GDScript. It has native support for C#, and languages such as C++, Python, and Rust can be used via plugins.

1.5.3 Community

Due to the free and open-source nature of Godot, it is a community-driven project.

1.5.4 Assets

There are unofficial third-party resources available for use, which make the lives of game developers much easier.

1.5.5 Cross-platform Support

Godot supports a wide range of systems such as desktops, mobile, web, consoles, VR/AR, and others.

1.5.6 Licensing

Game developers who use Godot don’t have to pay for licensing or royalties once the game is released because it’s completely free and open-source.

1.5.7 Customization

User extensibility is one of the primary attractions of Godot. Due to the availability of the source code to the public, it can be modified according to the wishes of the developers.

1.5.8 2D Game Development Proficiency

Godot is considered by many to be the best and most optimized game engine for 2D game development.

1.5.9 Learning Resources

The official website for Godot features extensive documentation and tutorials. There are also plenty of online resources on platforms such as YouTube.

1.5.10 Target Audience

Primarily targeted towards indie game developers and small game studios, Godot caters to its audience by continually updating its features to compete or even surpass more established game engines.

CHAPTER 2

PLATFORMER GAMES

2.1 Introduction

A Platformer is a type of action-oriented game where the primary objective is to move the Player character between different points in a scene. Platform games typically feature levels with different terrain and platforms that require movements such as jumping to navigate the game's environment.

The concept of a Platformer game involves the Player passing their character across platforms to reach an endpoint while battling enemies and avoiding obstacles set along the path. These games commonly offer a side view, through movement in two or three dimensions, with the camera placed either behind the main character or in an isometric viewpoint.

Common movements in this genre include walking, running, jumping, attacking, and climbing. Generally, games of this genre contain multiple levels with increasing difficulty that may be bound together by boss fights, where the character must overcome a challenging enemy to progress to the next level. Logical puzzles and skill-based tryouts to master the game are common features in platformers [7].



Fig. 2.1: Celeste (2018) – Example of a Platformer

2.2 Types of Platformer Games

This section describes the common types of Platformer games that are widely available on the market and provides details about their distinguishing features.

2.2.1 Action Platformer

The Action Platformer adds a combat element to the original Platformer concept. Although the Player will navigate their environment by running and jumping, the primary interaction with enemies or the environment will involve shooting weapons or some other type of attack. “Action” symbolizes not the game’s traversal, but its combative style [8].



Fig. 2.2: Hollow Knight (2017) – Example of an Action Platformer

2.2.2 Precision Platformer

Precision Platformers are known for their extreme difficulty combined with checkpoint systems and instant restarts. These are considered by many to be the purest type of all Platformer games [8].



Fig. 2.3: Geometry Dash (2013) – Example of a Precision Platformer

2.2.3 Puzzle Platformer

A Puzzle Platformer is a type of Platformer game based on solving, as opposed to enemy combat or reaching an endpoint. Normally, these games are displayed on a single screen, even though scrolling Puzzle Platformers are just as practical [8].



Fig. 2.4: Fez (2012) – Example of a Puzzle Platformer

2.2.4 3D Platformer

3D Platformers add a 3rd dimension that drastically changes the gameplay experience since platforming games are based around running and jumping. This genre remains a niche today, likely due to the fact that they are incredibly difficult to design, especially the challenging detail of controlling the Player's viewpoint with a 3D camera system [8].



Fig. 2.5: Super Mario 64 (1996) – Example of a 3D Platformer

2.2.5 Cinematic Platformer

Cinematic Platformers are relatively uncommon 2D side-scrolling adventures that aim to give out a more "realistic" style of gameplay with fewer cartoon-like movements and characters, and more realistic physics. For instance, the main character won't be able to jump 10 feet into the air or adjust their jump speed while in the air. Normally, the Protagonist is also more characteristically vulnerable to threats, meaning that a fall from a decent height will be consequential [8].



Fig. 2.6: Limbo (2010) – Example of a Cinematic Platformer

2.2.6 Platform Fighter

A Platform Fighter adds the combat style of arena-based fighting with the jumping movement and the vertical multilevel aspect of side-scrolling platformers. Platform Fighters get a significant portion of their characteristics from Platformer games [8].



Fig. 2.7: Brawlhalla (2014) – Example of a Platform Fighter

2.2.7 Auto-Runner

Auto-Runners are distinctly unique from other platformer games because the Player only has control over when the Protagonist jumps, since the Protagonist moves towards a set path automatically. These can contain some variation besides the timing of the jumps; some Auto-Runners may have multiple correct paths that the Player chooses from. Nevertheless, Auto-Runners usually tend to be mostly single-input games. Therefore, the complete design is quite enticing for touchscreen/mobile games [8].



Fig. 2.8: Temple Run (2011) – Example of an Auto-Runner

2.3 Why an Action Platformer?

We chose the Action Platformer subgenre because it combines precise platforming with combat mechanics, offering engaging gameplay and opportunities for creative enemy AI and weapon systems. The Action Platformer subgenre is ubiquitous in the Platformer landscape. They are universally loved because of their straightforward nature: run, jump, fight, and destroy the enemies to finish the levels. Challenging enemy encounters and platforming systems add to the depth and replayability of this genre.

CHAPTER 3

BASIC CONCEPTS OF GODOT

3.1 Introduction

Godot's architecture is based around scenes and nodes. A scene is a reusable component, for example, the Player, an entity, or a level. Nodes are the fundamental building blocks of gameplay elements. Resource files are stored in Godot in a tree structure, which is used to connect all the scenes, nodes, and files containing soundtracks, sprites, tilesets, etc.

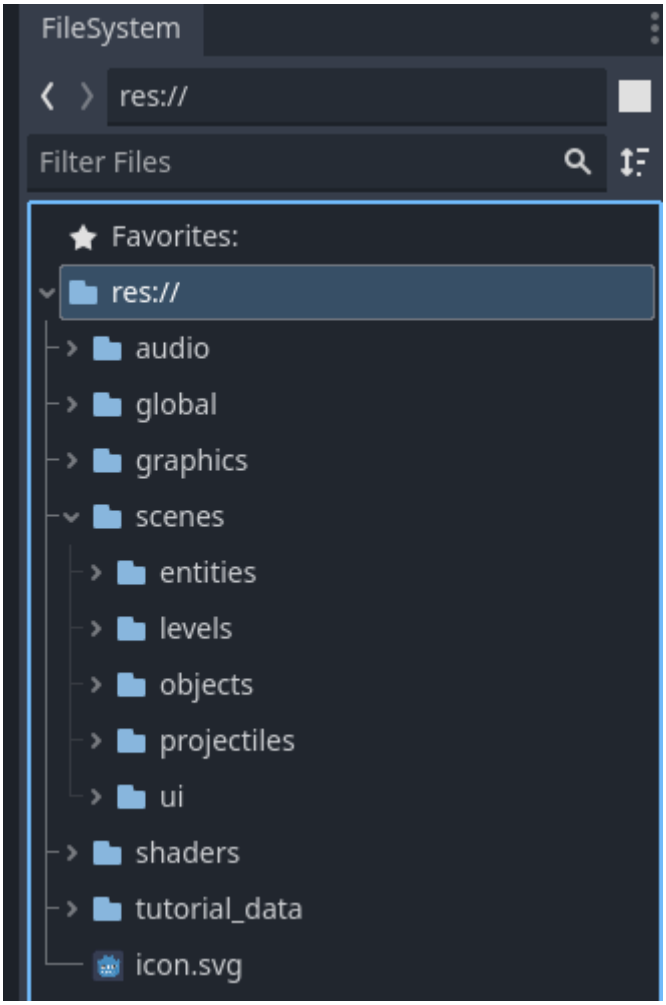


Fig. 3.1: File Structure of Our Project, Showcasing Resource Folders for Audio, Graphics, Levels, Projectiles, etc.

3.2 Scenes

Scenes are used to break down the entire game into smaller parts. A scene can be anything from a character, a weapon, a menu, a single level, etc. Scenes can be nested inside other scenes [9].

Scenes have the file extension ".tscn". Each of the scene files typically contains the properties and configuration of the nodes, groups, etc.

When the game is run, the scene files are instantiated in a data structure called the Scene Tree. The Scene Tree is the backbone of the game, which manages everything from rendering and physics to input processing.

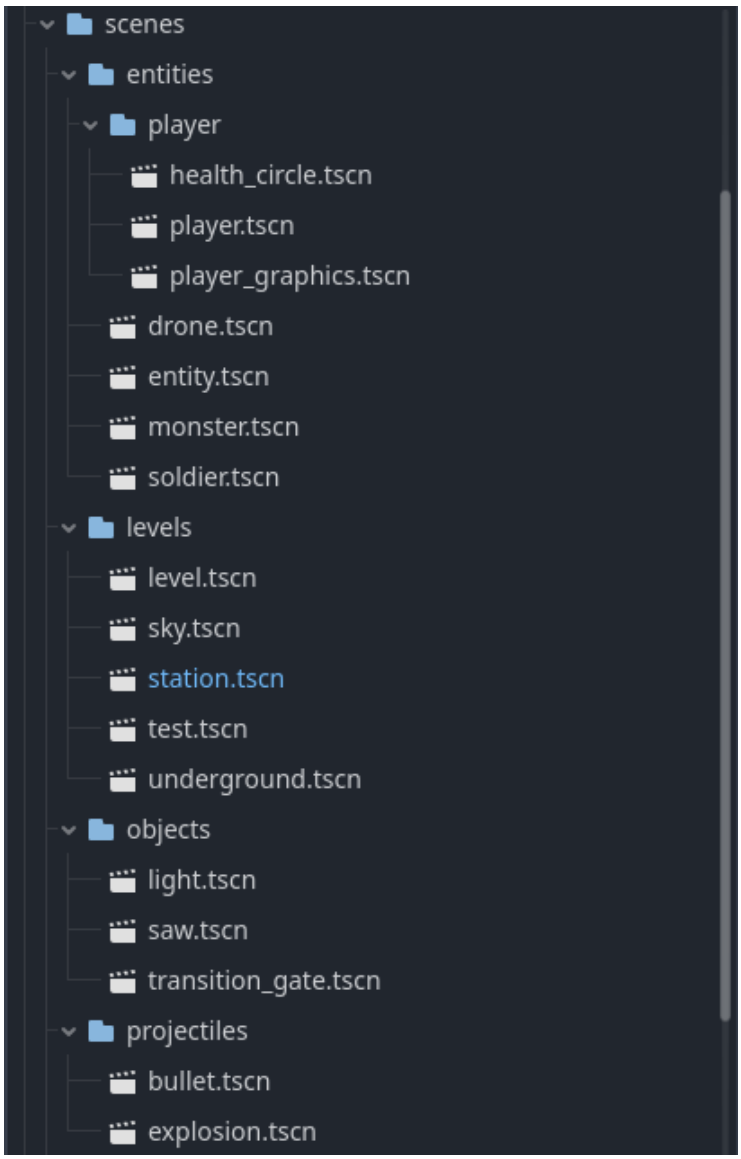


Fig. 3.2: Overview of Major Scenes in Our Project, Including Player, Drone, Monster, and Level Scenes

3.3 Nodes

A scene contains one or more nodes. Nodes are the smallest building blocks of the game that are arranged into trees [9]. Each node provides specific functionalities such as audio playback, scripting, rendering/physics, UI, etc.

For example, the “Player.tscn” file is composed of a CharacterBody2D node named “Player” which is used for detecting collisions and applying physics, a Camera2D node for tracking and following the Player character, and others.

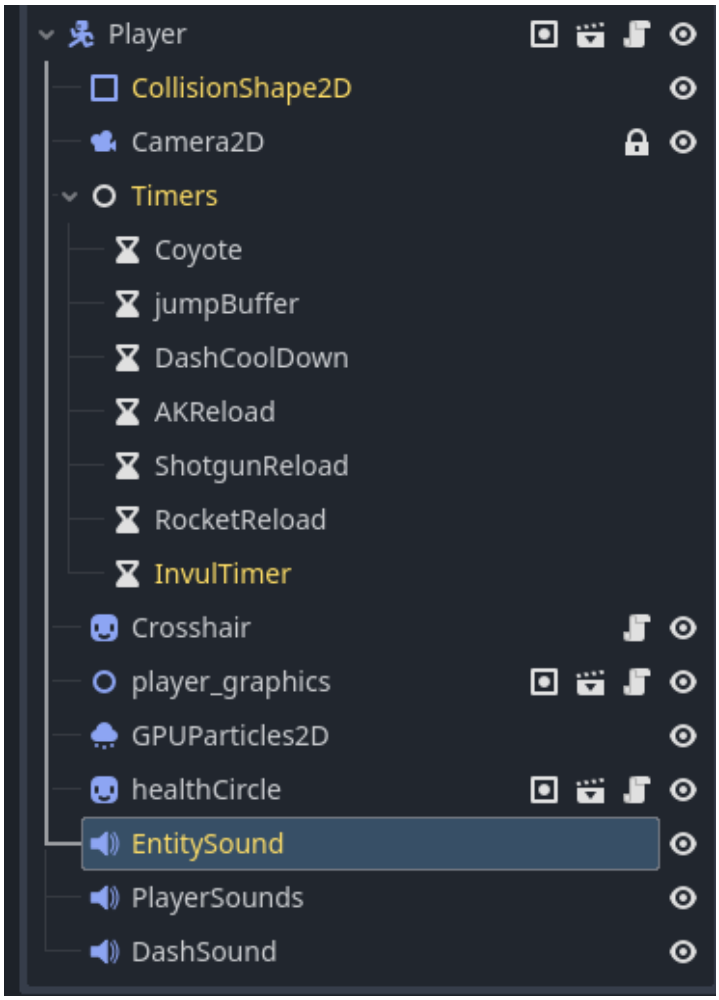


Fig. 3.3: Internal Components of the “Player” Scene, Showing Core Components Such as Camera2D, CollisionShape2D, and Timers

Descriptions of all the nodes used in this project and their use cases are given in the following subsections.

3.3.1 Node2D

Node2D is a 2D game object that is inherited by every 2D-related node. It has a position, rotation, scale, and skew [10].

In our game, Node2D acts as the parent node to all the 2D entities, including the Player character, enemy entities, and the environment. It allowed us to efficiently manage the spatial relationships between them and apply transformations to every node during transitions to other levels.

3.3.2 Sprite2D

A node that can visualize a 2D texture. The visualization can be an area from a large texture, or a frame from a sprite sheet [10].

We used Sprite2D in our game to display assets such as walls, floors, and lights.

3.3.3 AnimatedSprite2D

AnimatedSprite2D is akin to the sprite2D node, but it carries multiple textures as frames for animation [10].

In our game, AnimatedSprite2D was employed to animate characters such as the Protagonist. For example, the Protagonist has separate animations for idle, running, jumping, and attacking that were all managed using this node.

3.3.4 AnimationPlayer

An AnimationPlayer node is used for regular animation playback. It contains a collection of resources and custom blend times between animation transitions [10].

We relied on AnimationPlayer to animate key characters such as the Protagonist, and enemy entities such as the Soldier and the Monster.

3.3.5 AnimationTree

A node used for advanced animation transitions within an AnimationPlayer node [10].

AnimationTree was instrumental for managing animations when the Player switches weapons.

3.3.6 Area2D

An area of 2D space that detects other objects entering or exiting the area. It also keeps a record of which objects haven't left it yet. This node can locally affect or override physics parameters such as gravity [10].

We used Area2D extensively to allow the enemy entities to detect the Player character. For example, the Drone entity has an attack radius that was implemented using Area2D. When the Player character enters the attack radius, the drone activates and begins chasing the Player.

3.3.7 AudioStreamPlayer

This node plays an audio stream non-positionally and is ideal for UIs, menus, or background soundtracks [10].

AudioStreamPlayer handles all audio in our game, including the background soundtrack and sound effects for firing weapons and explosions.

3.3.8 Control

Base class for all UI-related nodes. It features a rectangle that delineates its contents, either an anchor position relative to its parent control or the current viewport, and offsets relative to the anchor [10]. Control nodes are used for handling user input, layout, and drawings for GUIs.

The Control node is the foundation for all UI elements in our game, such as the main menu.

3.3.9 Button

The Button node is the standard themed button that can hold text or an icon, and highlight them according to a specific theme [10]. Button inherits from Control, and thus possesses all the properties and functionalities of a Control node, alongside its own specific features for creating interactive buttons.

In our game, the Button node is used in the main menu. The main menu features three buttons labeled “Play”, “Quit”, and “Credits” respectively. The Player can interact with these buttons to navigate the game. For example, clicking on the “Play” button will start the game, and “Quit” button will end it.

3.3.10 Camera2D

Camera2D is the default camera node for 2D scenes. It forces the current layer to scroll whilst tracking this node, which makes it efficient to program scrollable scenes [10].

Camera2D follows the Player character throughout the levels and keeps them centered on the screen. Additionally, the Camera2D node helped us implement a camera shaking effect when the Player hits the Soldier.

3.3.11 CanvasItem

CanvasItem is the abstract base class for all the elements in 2D space. Canvas items are categorized in a tree, and children inherit and extend their parent’s transform [10].

All the visual elements in our game, such as the sprites, UI elements, and particle effects, inherit from CanvasItem. This node allowed us to layer foreground and background components effectively.

3.3.12 CanvasLayer

A node used for decentralized rendering of objects inside a 2D scene [10].

CanvasLayer puts the UI elements in our game, such as the main menu, on their own layer, which makes it easier for us to distinguish and use them.

3.3.13 CharacterBody2D

CharacterBody2D is a distinguished class for physics bodies that are intended to be user-controlled. They are not impacted by physics anyhow, but they affect other physics bodies in their trajectory [10].

The Player character and enemy entities are all implemented as CharacterBody2D nodes. This enabled us to implement their movement, for example, dashing, ducking, and jumping.

3.3.14 CollisionShape2D

A node that supplies a Shape2D or CollisionObject2D parent and permits manipulating it. Thus, a detection shape can be given to an Area2D, or convert a PhysicsBody2D into a solid object [10].

In our game, CollisionShape2D is used to define the collision area for the Player and enemy entities. It detects the collision between the Protagonist and the Drone, or the Saw.

3.3.15 ColorRect

Used to visualize a rectangle containing a solid color [10].

We used the ColorRect node in our game to create a transition effect that fades the screen to black whenever the Player moves between levels.

3.3.16 GPUParticles2D

A 2D particle emitter node used to produce a range of particle systems and effects. It generates a range of particles at a given interval [10].

The special effects in our game, such as the damage particle effect, which emits particles after the Soldier is shot by the Player, are produced using the GPUParticles2D node.

3.3.17 ParallaxBackground

A node used to produce a parallax scrolling background using any number of ParallaxLayer child nodes [10].

The ParallaxBackground node helped us add depth to our game environment by making the background scroll slower than the foreground, which creates a smooth sense of motion as the Player progresses through the environments.

3.3.18 Timer

A countdown timer that offers a simple way to manage time-based logic in the Godot engine [10].

Multiple Timer nodes were used throughout the project to keep track of reload times, cooldown times for weapons, or jump mechanics such as dashing. For example, after dashing, a Timer node prevents the Player from dashing again for 0.5 seconds, which encourages strategic movement and balances the gameplay experience.

3.3.19 VBoxContainer

VBoxContainer is a type of container used to decide its child controls vertically, which are automatically reordered when their minimum size shifts [10].

The main menu in the game uses a VBoxContainer node to organize the buttons (“Play”, “Quit”, “Credits”). This node ensures that the buttons are automatically resized according to the screen resolution, and the space between them is consistent.

3.4 Signals

Nodes tend to give off signals when some activity occurs. This characteristic gives us the opportunity to make nodes communicate with each other instead of hardcoding them. It provides flexibility when it comes to structuring scenes [9].

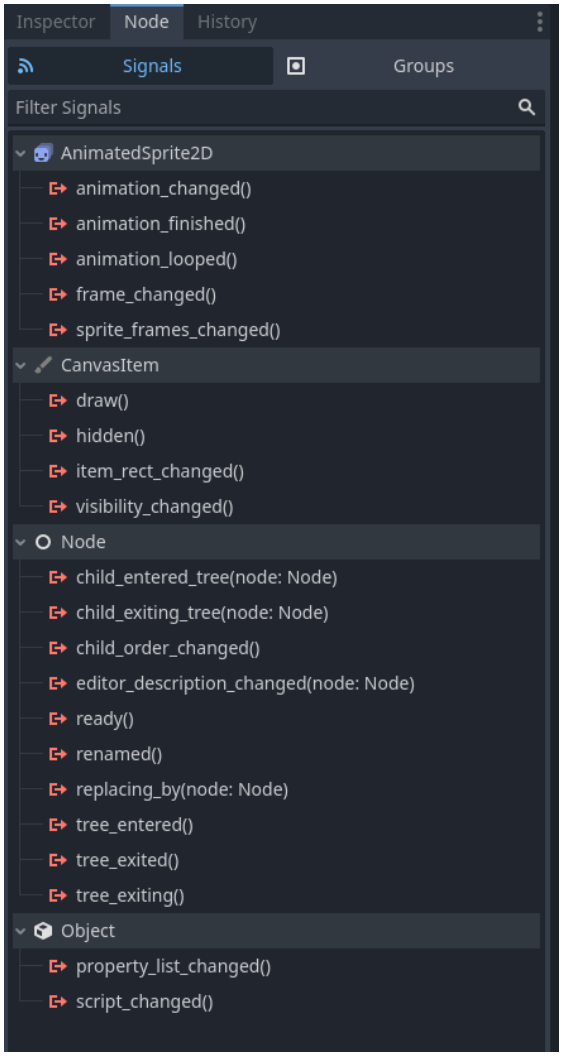


Fig. 3.4: Signal Connections from the AnimationSprite2D Node Used to Trigger Events Like Animation Completion

3.5 GDScript

GDScript is a high-level, object-oriented, imperative, and gradually typed programming language designed for Godot. It employs an indentation-based syntax akin to languages such as Python. The goal of GDScript is to be optimized and smoothly integrated into the Godot engine, and introduce a great level of flexibility for content production [11].

```gdscript
func _on_button_pressed(button):
    $"Button Sounds".play()  # Play press sound
    await $"Button Sounds".finished # Wait until the sound finishes
    # Then do the action
    match button.name:
        "Play":
            get_tree().change_scene_to_file("res://scenes/levels/station.tscn")
        "Quit":
            get_tree().quit()
        "Settings":
            print("Settings menu coming soon")
func _exit_tree() -> void:
    (BgMusic.get_node("AudioStreamPlayer") as AudioStreamPlayer).play()
```

Fig. 3.5: Sample GDScript Code Demonstrating Input Handling in Our Game

CHAPTER 4

GAME DESIGN

4.1 Flowchart

A flowchart is a type of diagram commonly used in various fields to present complex processes in a simpler manner [12]. Rectangles, diamonds, and other shapes are used to communicate the steps of the program and various sequences that can take place.

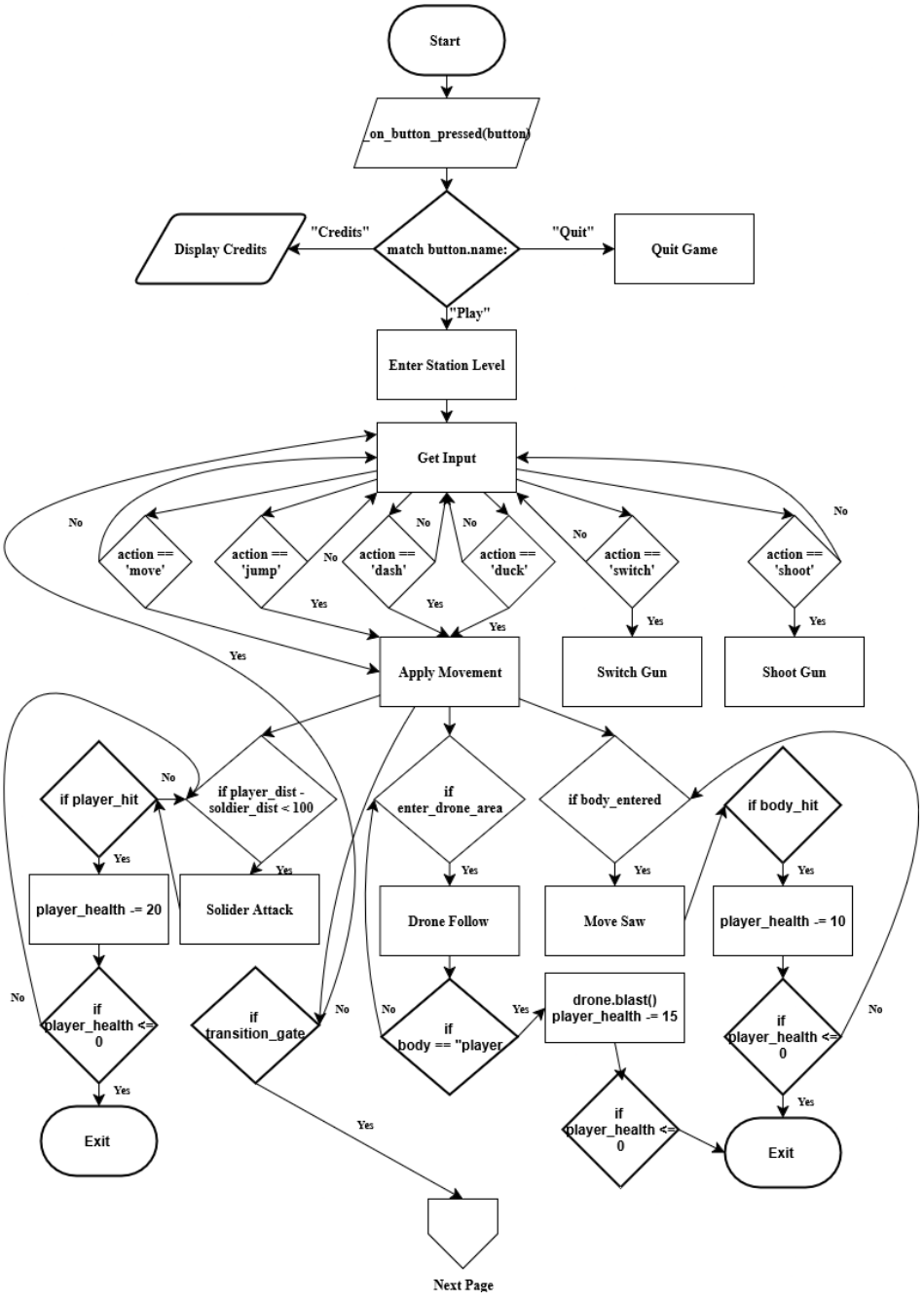


Fig. 4.1: Flowchart of the Station Level, Detailing Enemy Engagements, Player Input, and Transition to Other Levels

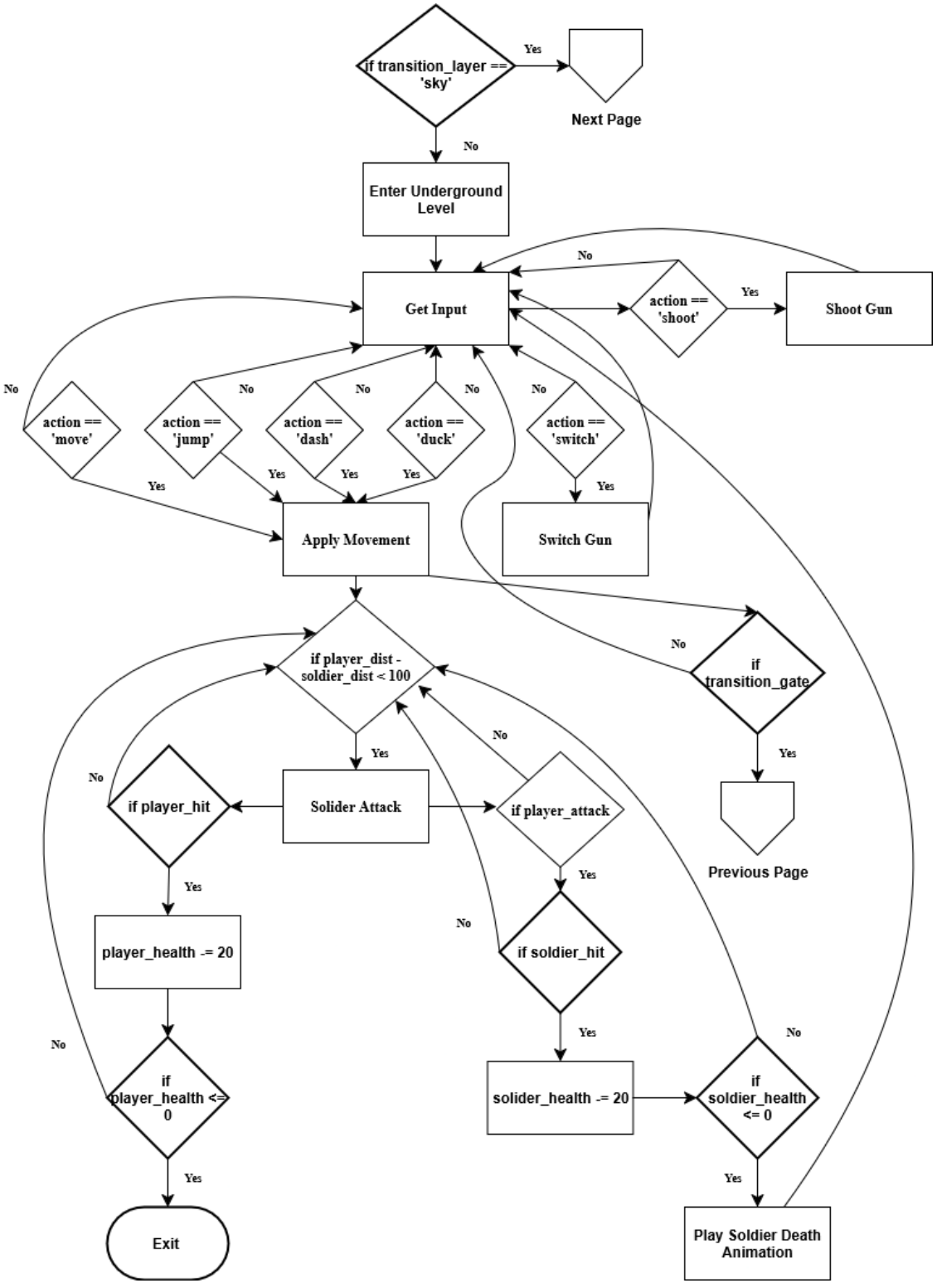


Fig. 4.2: Flowchart of the Underground Level, Outlining the Player and the Soldier's Actions and Transitions to Other Levels

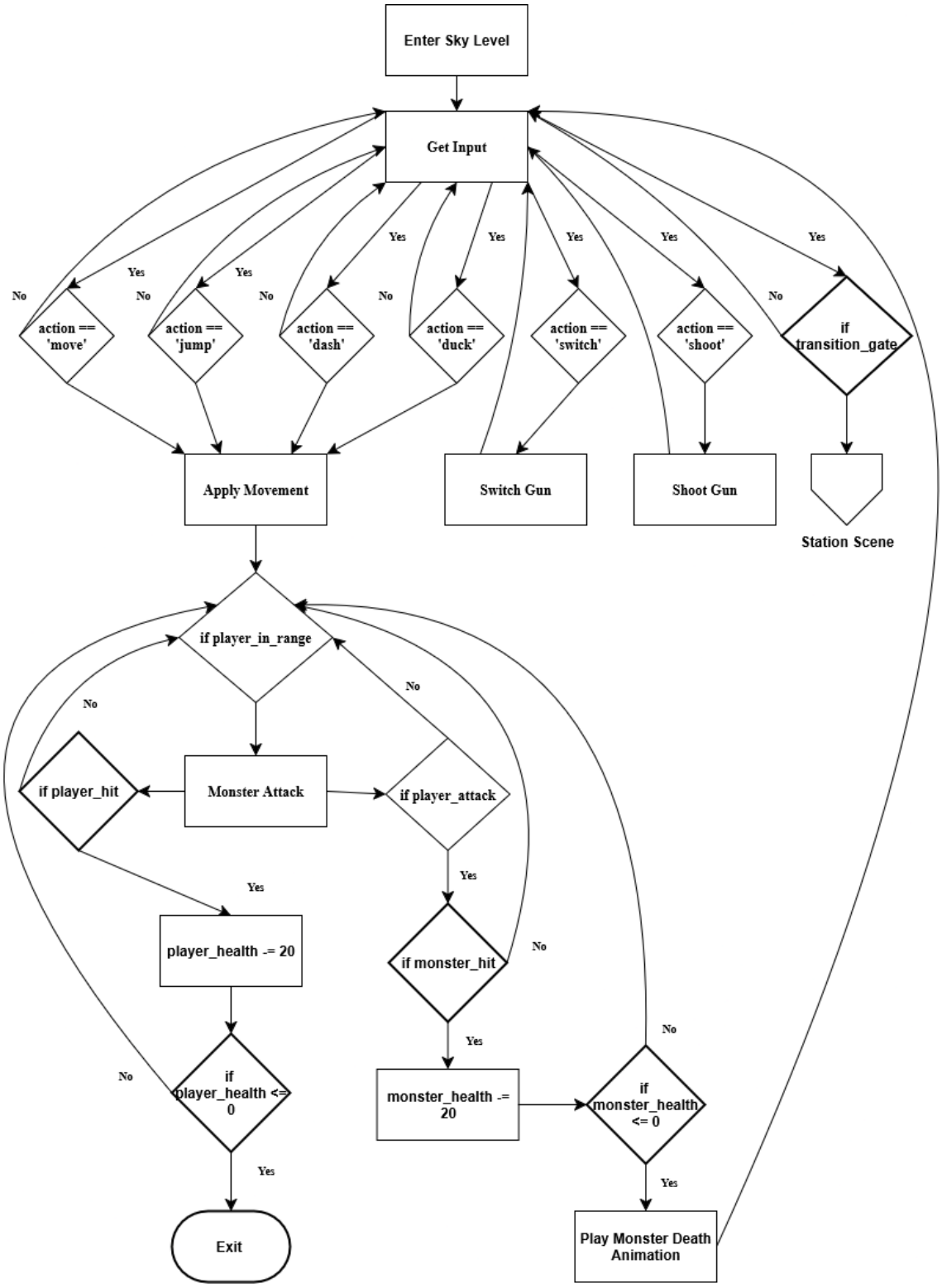


Fig. 4.3: Flowchart of the Sky Scene, Outlining Player's Engagement with the Monster

4.2 Data Flow Diagram

A data flow diagram or DFD designs the flow of information for any process or systemic element. Using definitive symbols such as rectangles, circles, arrows, and text labels, it shows the data inputs, outputs, storage points, and the path between each destination. DFDs are used to analyze an existing process, or system model, or create a new one [13].

4.2.1 Level 0 DFD

Level 0 DFD is also referred to as a Context Diagram. It provides a general overview of the entire system or process being analyzed or modeled. A single high-level process and its relationship to external entities are used for a quick view into the system [13].

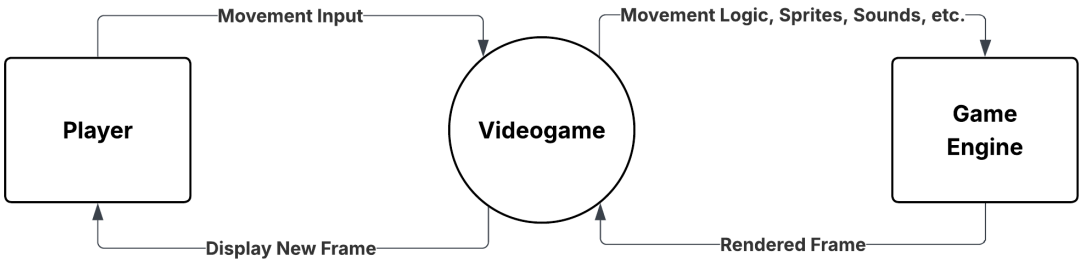


Fig. 4.4: Level 0 DFD, Showcasing the Game as a Single Process Interacting with External Entities

4.2.2 Level 1 DFD

Level 1 DFD breaks down the Context Diagram further to accentuate the primary function of the system. The high-level process of the Context Diagram is divided into subprocesses to provide a deeper understanding of the system or process being modeled or analyzed [13].

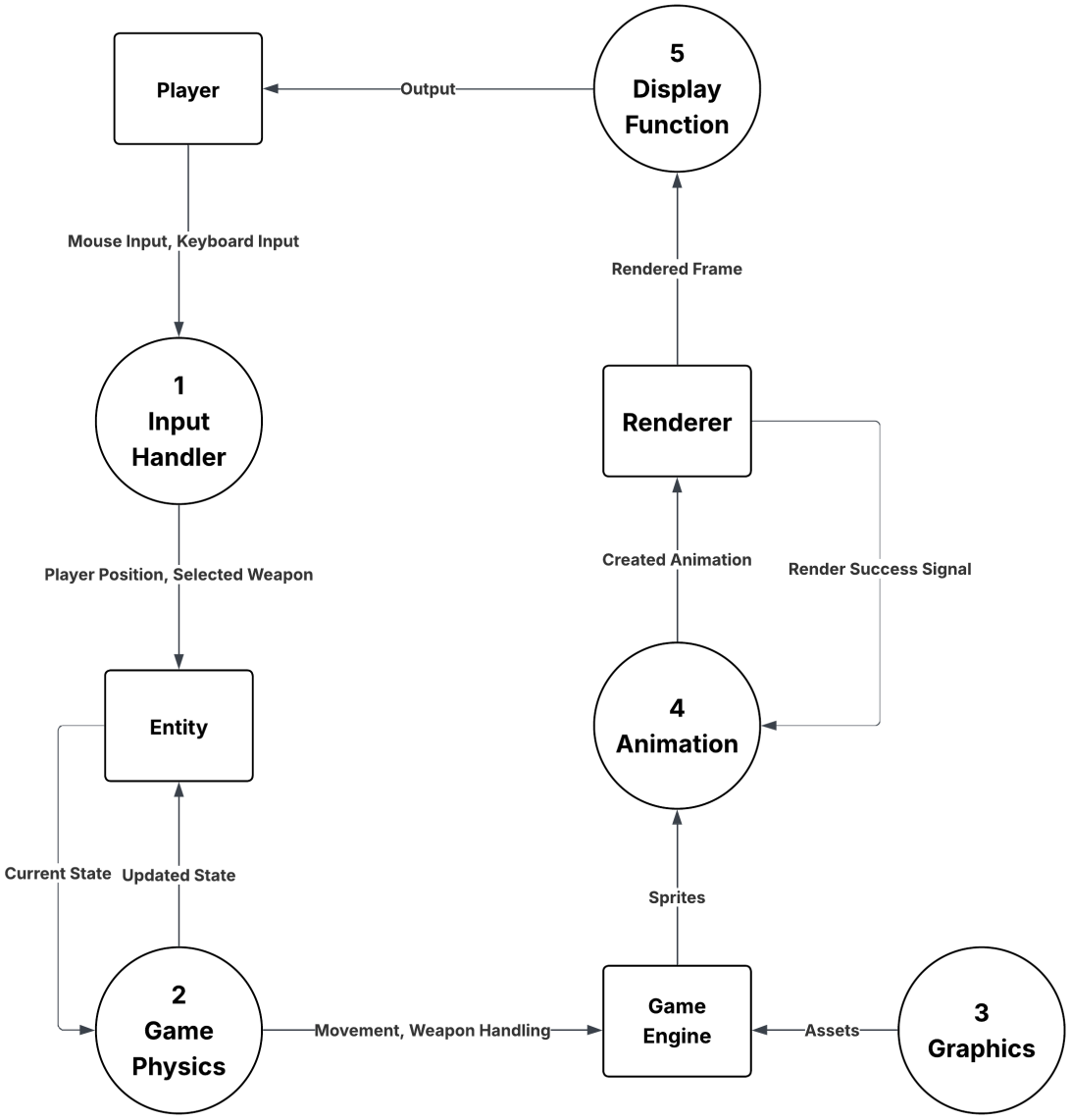


Fig. 4.5: Level 1 DFD, Breaking Down the Game into Subprocesses Such as Player Input, Game Physics, and Graphics

4.3 Use Case Diagram

Use Case Diagrams are part of the Unified Modeling Language, or UML, which is a modeling toolkit for building diagrams. It is used to summarize the scheme between the system's users or actors and their relationships with the system. It can help effectively discuss and navigate the scenarios where the system, or application, interacts with people, organizations, or external systems [14].

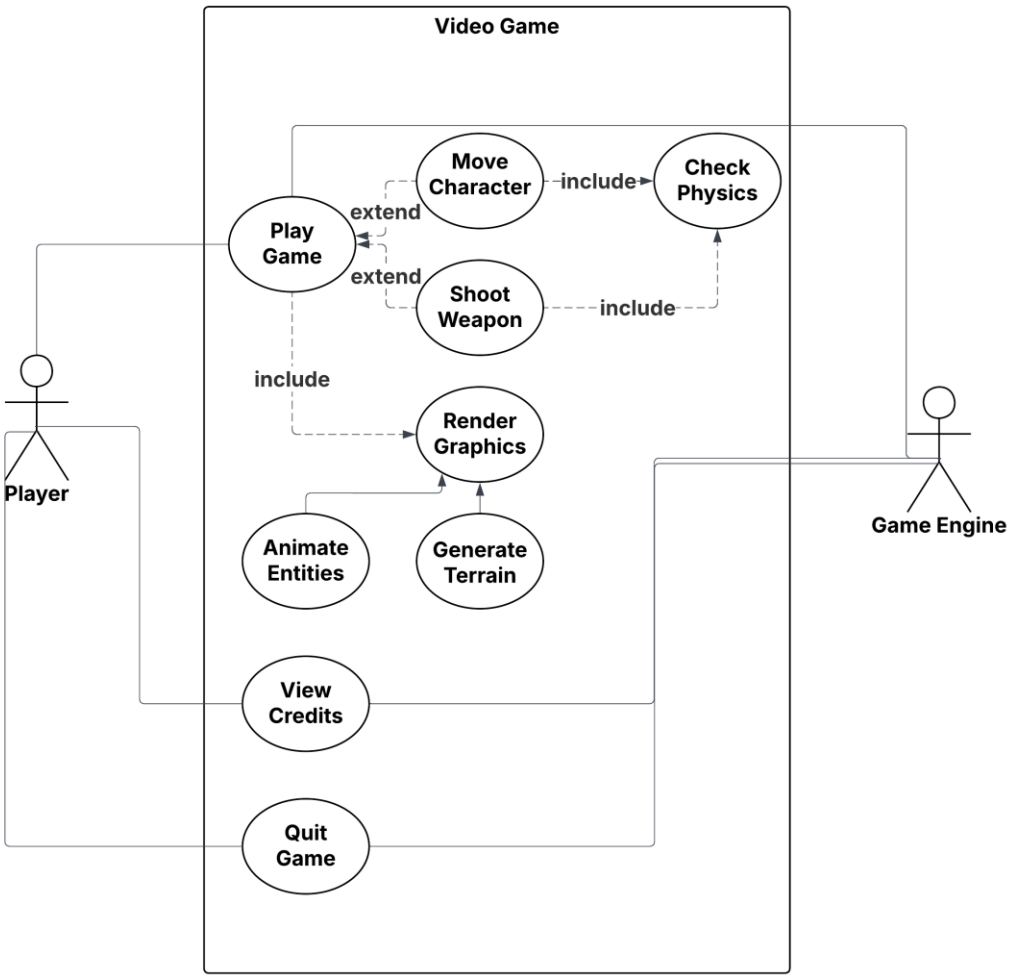


Fig. 4.6: Use Case Diagram Illustrating Interaction Between the Player and Core Game Features Including Movement, Weapon Switching, and Menu Navigation

4.4 Environment Design

The game levels are divided into three different environments: Station, Sky, and Underground. Player, by default, starts the game in the Station scene. Transition layers and gates are used to move between the scenes. The environments are created using 16x16 tiles from tilesets.

4.4.1 Tilesets

Tilesets are sets of small, reusable images called “tiles” that are configured in a grid. Each independent tile represents a distinct part of the game environment. A tile can be used to symbolize a piece of ground, a wall, or a decorative element. Different tiles can be used to create a detailed environment quickly and efficiently [15].

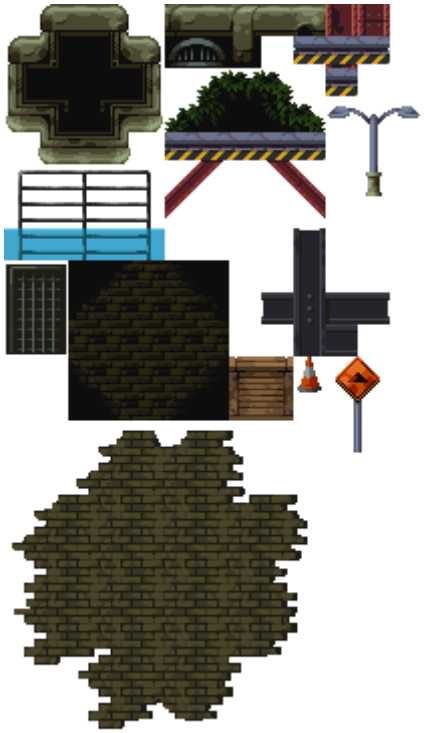


Fig. 4.7: Outside Tileset, Composed of Ground, Walls, and Decorative Elements

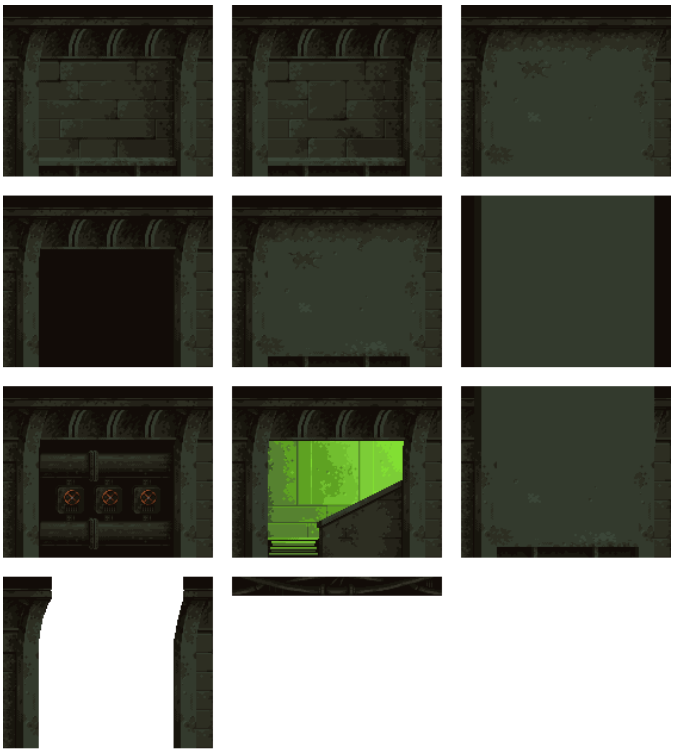


Fig. 4.8: Inside Tileset, Featuring Platforms for the Indoor Sections of the Station Level

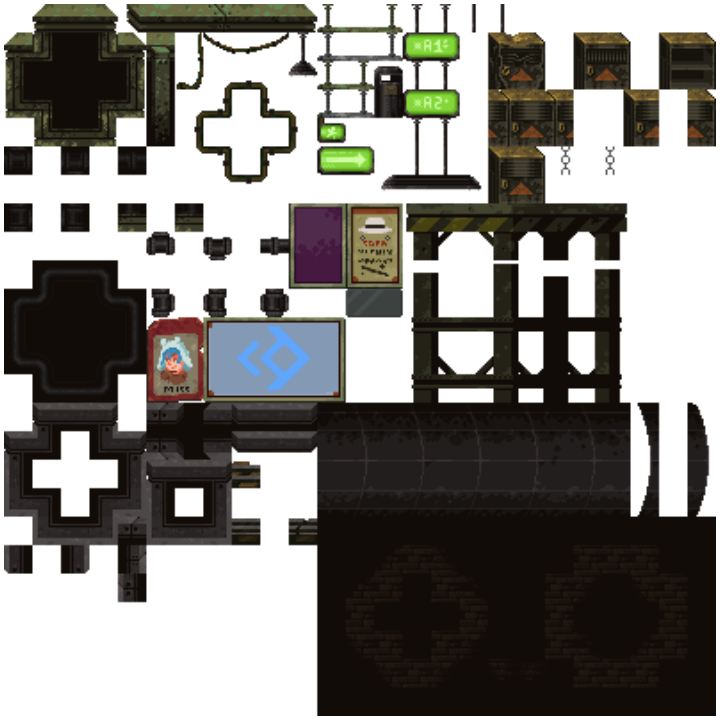


Fig. 4.9: Wall Tileset Used to Build Boundaries

4.4.2 Station Environment

Station environment is the default environment. The Protagonist spawns here once the game starts. It contains several enemy entities: Soldiers on different platforms, a Drone, and a Saw.



Fig. 4.10: Fully Assembled Station Level, Featuring Transition Gates to Other Levels and Enemy Entities Such as the Soldier, Drone, and Saw

4.4.3 Sky Environment

Sky environment features the boss battle with a Monster enemy entity and a transition gate to the Station scene. It is the final level of the game, and defeating the Monster entity will result in the Player winning the game.



Fig. 4.11: Sky Level Featuring the Monster entity, the Final Boss of Our Game

4.4.4 Underground Environment

The Underground environment consists of different platforms with Soldiers and a transition gate to the Station scene.



Fig. 4.12: Underground Scene, Featuring Multiple Soldier Entities for Moderate Difficulty

4.5 Player Mechanics

Player mechanics refers to the physics behind the movements available to the Player. Walking, jumping, dashing, and ducking are the primary movements.

4.5.1 Walking

The basic Player movements are moving along the horizontal axis, that is, moving left or right using the combination of the mouse and keyboard. The mouse can be used to point the Protagonist towards a specific direction, and the left and right arrow keys allow the Player to move left or right along the environment. The Player's movement features acceleration and friction to make the movement feel smooth.

4.5.2 Jumping

Jumping can allow the Protagonist to move to a different platform, avoid obstacles, or bullets, etc. Pressing the “Spacebar” allows the Player to jump vertically, and long-pressing the “Spacebar” allows for a higher jump.

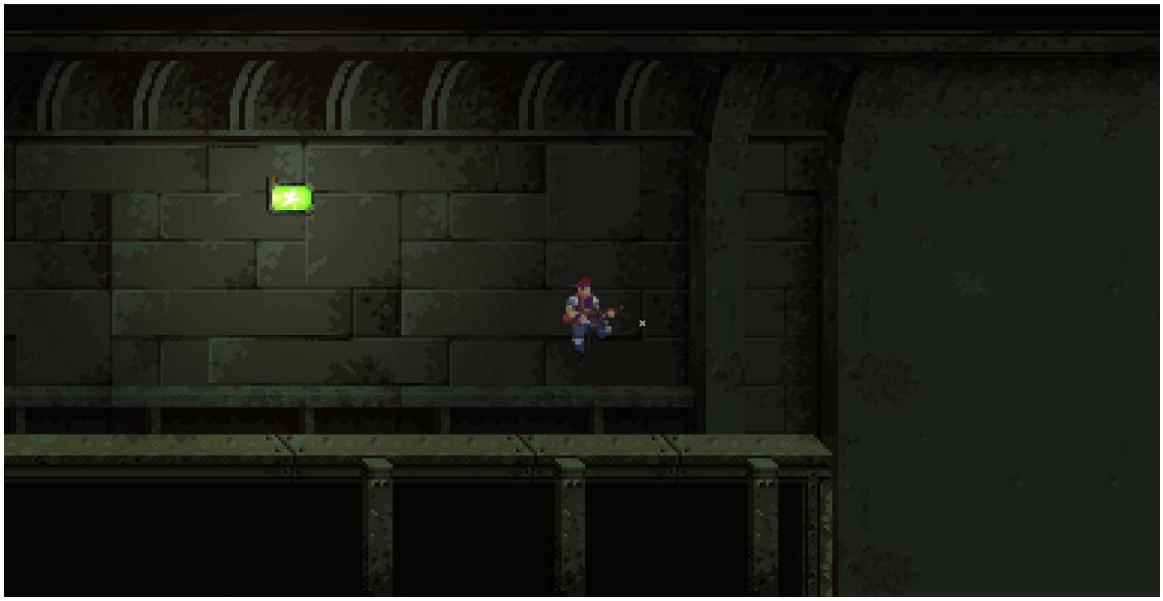


Fig. 4.13: Visual Representation of the Protagonist's Regular Jump Height



Fig. 4.14: Visual Representation of the Extended Jump Height Achieved by Long-Pressing the "Spacebar"

After the Player walks off a cliff, there is a small amount of time where the Player can still jump. This is referred to as Coyote time. It is used to increase the responsiveness of the game.



Fig. 4.15: Player Character Jumping Off a Cliff



Fig. 4.16: Demonstration of Coyote Time – A Brief Period After Jumping Off a Cliff During Which the Player Can Still Jump

4.5.3 Dashing

Dashing allows the Player the ability to gain a fixed amount of distance horizontally in quick succession. It is usable only when in a jump state. There's a waiting period before the dashing movement can be performed again, which is called the Dash Cooldown.

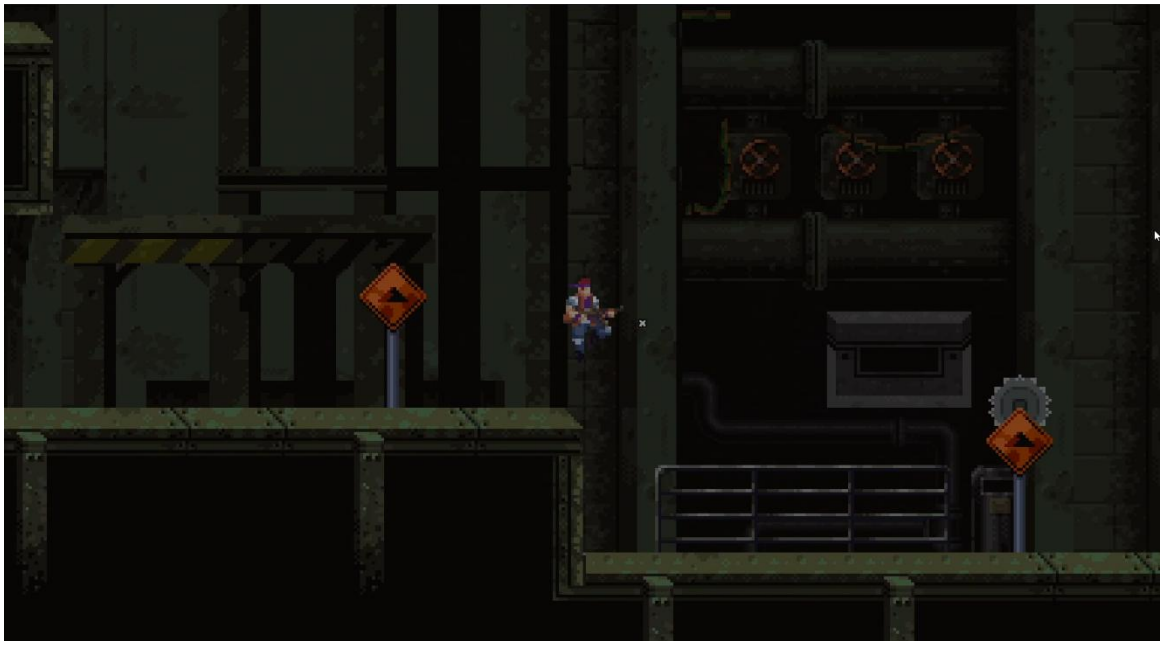


Fig. 4.17: Player Character in a Jump State Before Dashing



Fig. 4.18: Player Character's Position After Performing the Dashing Movement

4.5.4 Ducking

Ducking gives the Player the ability to dodge bullets from the Soldier or Monster enemy entities. The left "Ctrl" key is used to duck temporarily.



Fig. 4.19: Protagonist Under Enemy Fire, Ducking Bullets

4.6 Enemy AI Behaviors

Enemy AI characters have a defined area. When the Player character enters the area, the enemies start to attack. Each of the enemies: Soldiers, Drone, Saw, and Monster has its own unique characteristics. For example, when the Drone collides with the Protagonist, it explodes.

4.6.1 Soldier

The Soldier entity remains on a fixed platform and checks to see if the Protagonist is in a given range. When the Player enters the designated zone for the Soldier, it is attacked.

```
func check_player_distance():
    if position.distance_to(player.position) < 100:
        attack = true
        speed_modifier = 0
    else:
        attack = false
        speed_modifier = 1
```

Fig. 4.20: Programming Logic Showcasing the Distance Check Between the Soldier and the Player

The “check_Player_distance()” function in Fig. 4.20 is used to calculate the distance between the Protagonist and the Soldier. If the distance drops below a certain point, the Soldier detects the Protagonist, stops moving, and starts attacking.

4.6.2 Drone

The Drone entity has an attack radius, which is an area that defines the Drone’s activation zone. When the Protagonist enters it, it activates the Drone. The Protagonist is then followed by the Drone on the same platform. If the Drone collides with the Protagonist, it explodes.

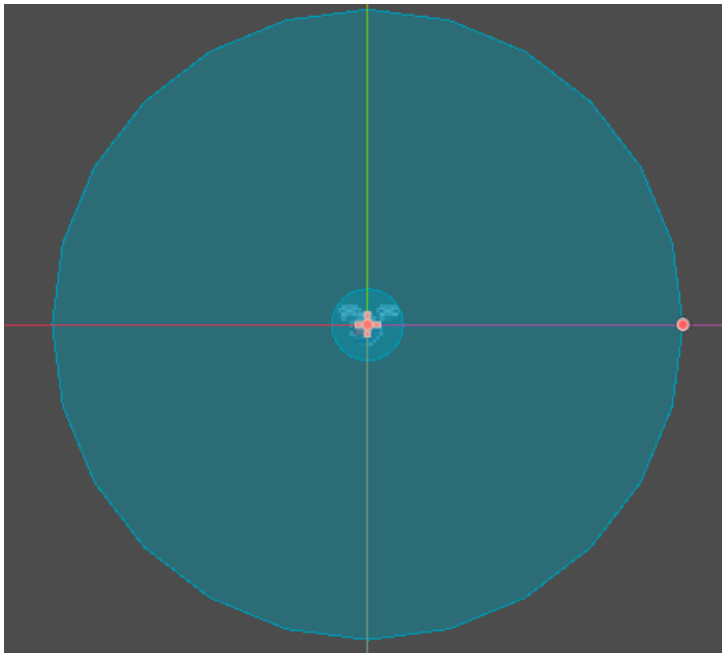


Fig. 4.21: Circular Area2D Node Defining the Drone’s Activation Area

The area denoted by the circle in Fig. 4.21 is the attack radius of the Drone entity. The Drone remains Stationary when the Player is outside of the attack radius.



Fig. 4.22: Stationary Drone Before Detecting the Player

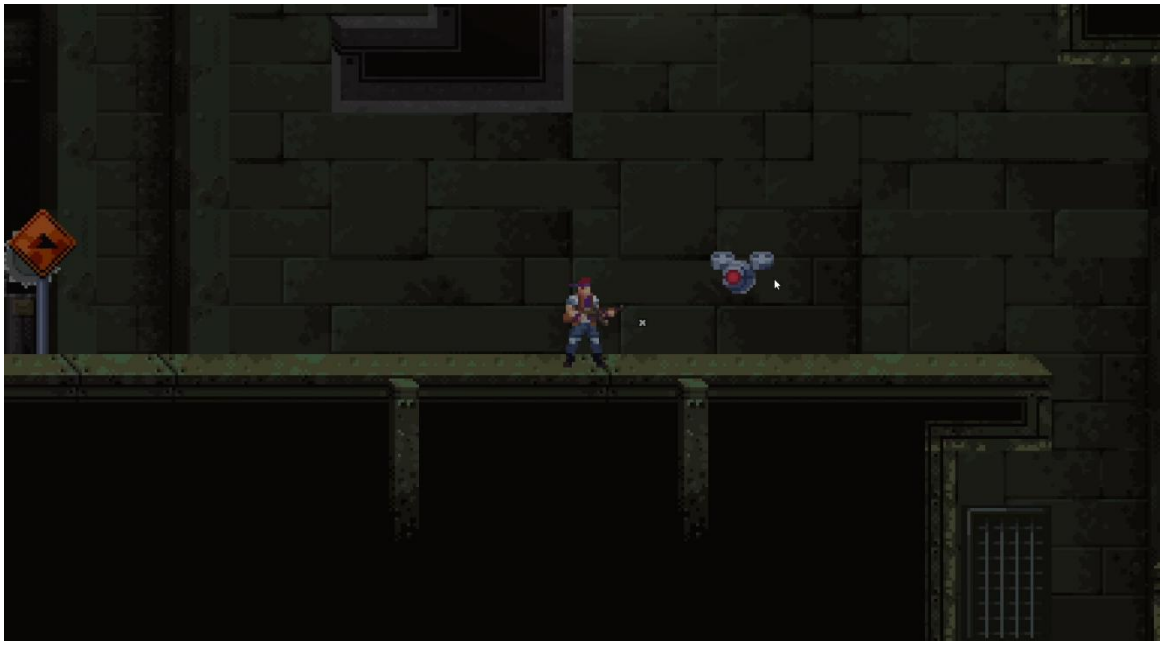


Fig. 4.23: Drone Following the Player After Detection

The Drone starts to follow the Player when it enters the attack radius, as shown in Fig. 4.23.

4.6.3 Saw

The Saw is situated around a platform and traverses a fixed path periodically. When the Saw collides with the Player, the Player loses health points.



Fig. 4.24: Predefined Linear Path Followed by the Saw Entity

The Saw moves along the path shown in Fig. 4.24, and the position constantly changes with time. The Player has to pay attention to the Saw's trajectory to avoid getting hit.

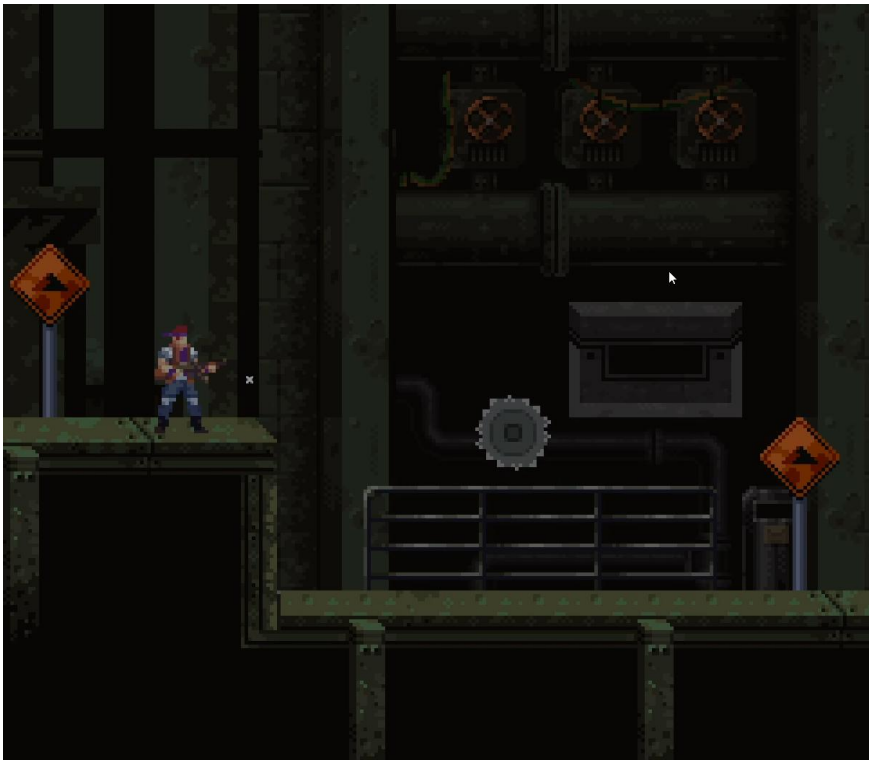


Fig. 4.25: Saw Moving Along Predefined Path Part 1

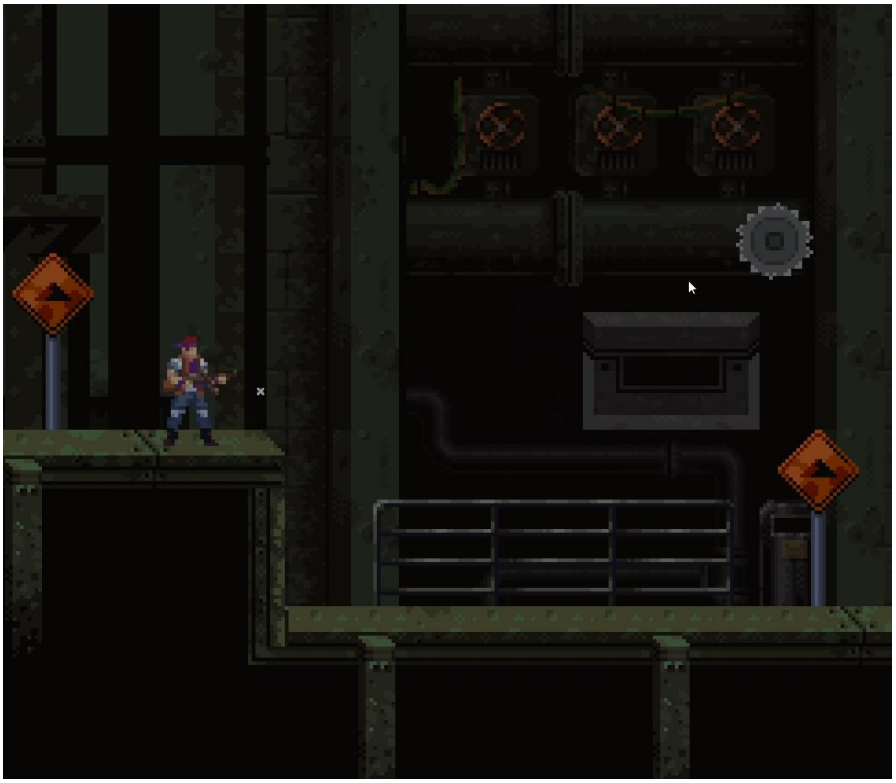


Fig. 4.26: Saw Moving Along Predefined Path Part 2

4.7 Boss Fight Design

The boss fight takes place in the Sky scene. The Player faces the Monster entity, which has a much wider attack radius than the Soldier and the Drone entity. It has 8 distinct bullet release points, as shown in the following figure, and 3 of those points are randomly selected at a time for shooting bullets.

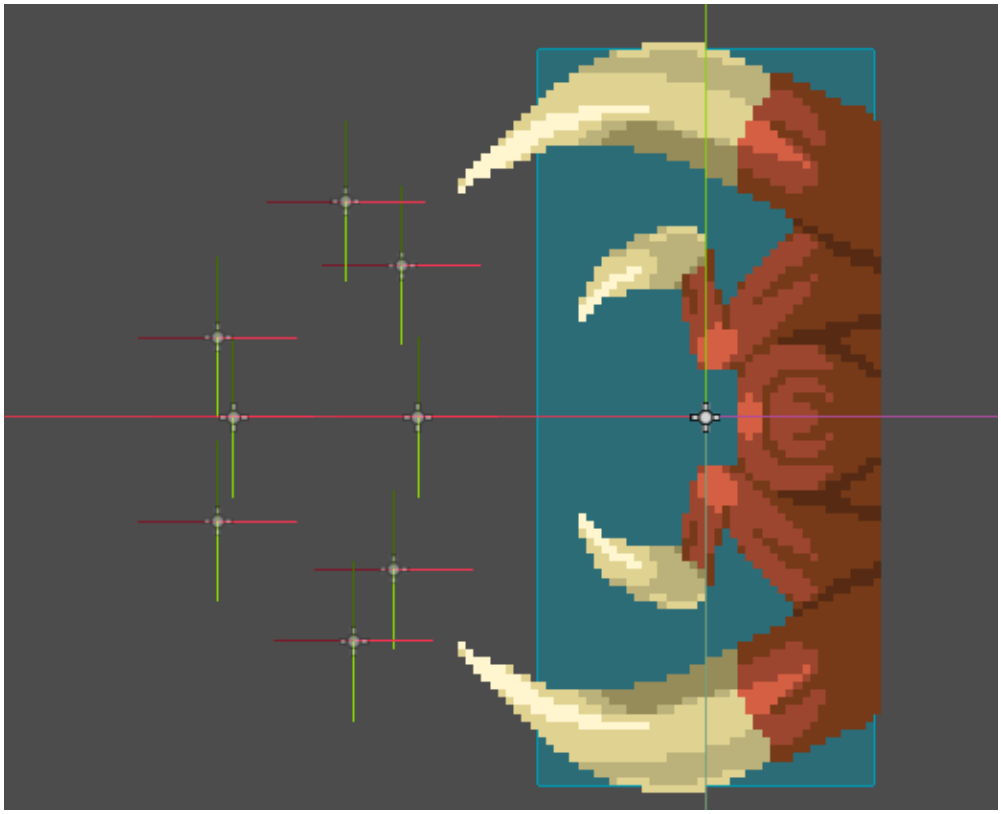


Fig. 4.27: Monster Entity Alongside 8 Possible Bullet Release Points

```
func trigger_attack():
    var option_index = rng.randi_range(2, $BulletOptions.get_child_count() - 1)
    var selected = $BulletOptions.get_child(option_index)
    for marker in selected.get_children():
        shoot.emit(marker.global_position, Vector2.LEFT, Global.guns.AK) # (position, direc
```

Fig. 4.28: Programming Logic Showcasing the Monster's Attack System

The "trigger_attack()" function in Fig. 4.28 uses a random number generator to select 3 out of the 8 available positions at a time, randomly. Then the selected positions are passed onto another function to fire the bullets.

4.8 Weapon System

The weapon system describes the aim directions and crosshair placement of the Protagonist, alongside their weapon selection.

4.8.1 Aim Direction

The Protagonist can select 8 specific release points or directions to shoot their weapons from. This is known as 8-way shooting. The aim direction is selected by moving the mouse along the horizontal and vertical axes.

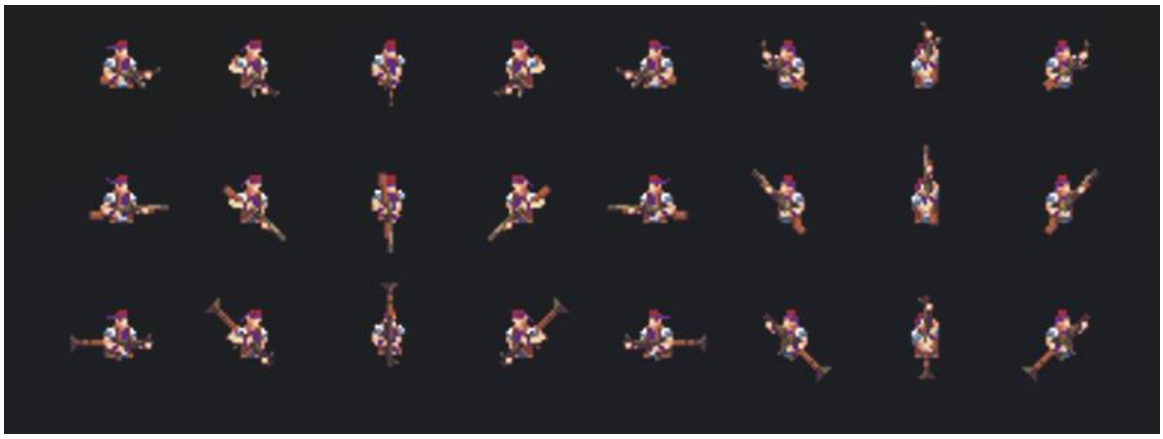


Fig. 4.29: 8-way Aiming System for Shooting Weapons

Each row in the above figure represents aim directions for a different weapon.

4.8.2 Crosshair Placement

The crosshair serves as a visual indicator of where the Protagonist is aiming the weapon. It can help the Player aim more accurately. Crosshair placement changes with the position of the Player. For example, when ducking, the crosshair will also move lower to compensate.



Fig. 4.30: Crosshair Placement in Normal Stance

The white cross in front of the Protagonist in Fig. 4.30 symbolizes the crosshair.



Fig. 4.31: Adjusted Crosshair Placement While Ducking

4.8.3 Weapon Options

The middle mouse button is used to switch between the three available weapons: the shotgun, the rocket launcher, and the AK. Different weapons fire at different rates, and each weapon does different damage to the enemy's health.

Certain enemy entities can only be eliminated using a certain weapon. For example, the Drone is only affected when hit by a rocket launcher. It provides a balance between the weapons by forcing the Player to switch to a rocket launcher if they want to attack the Drone.

Weapon Type	Damage Value	Fire Rate	Special Properties
AK	10	200	Quicker fire rate.
Shotgun	30	120	Balances damage value with fire rate.
RPG	80	100	The only weapon that can destroy Drones.

Table 4.1: Weapon Types and Characteristics

4.9 Lighting System

The game features 5 different types of lights installed on different platforms. Each of the lights flickers randomly to provide a dynamic lighting effect.

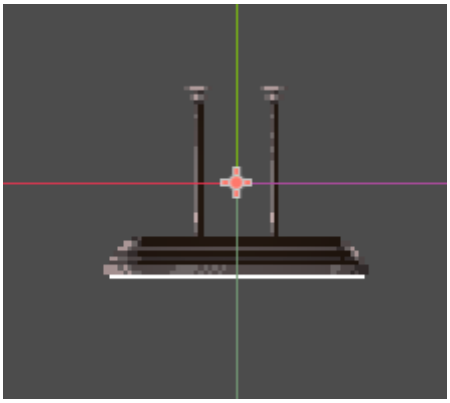


Fig. 4.32: Light Type 1

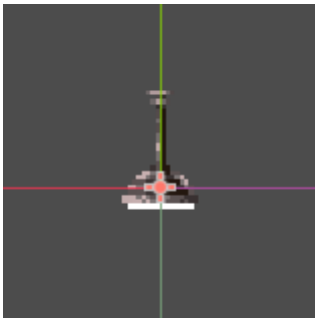


Fig. 4.33: Light Type 2

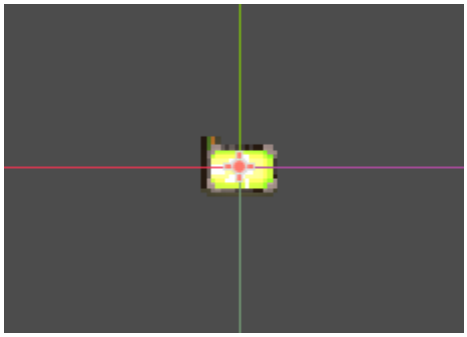


Fig. 4.34: Light Type 3

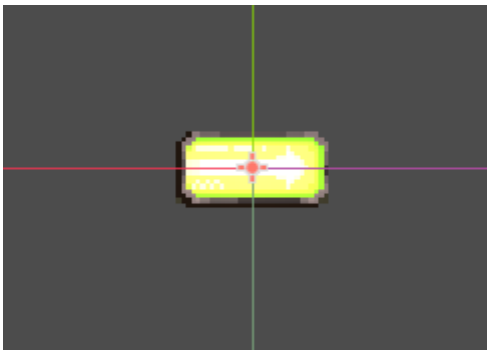


Fig. 4.35: Light Type 4

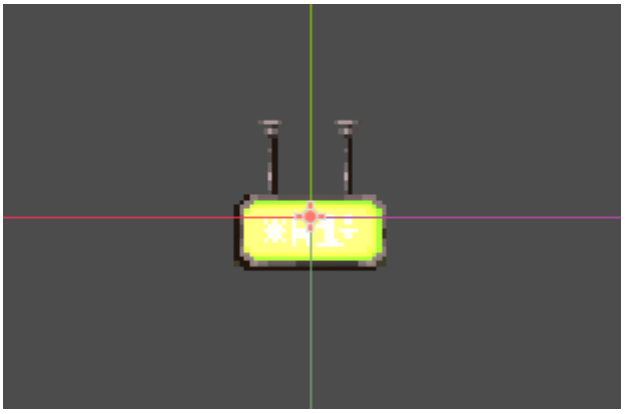


Fig. 4.36: Light Type 5

4.10 Sound System

The game features a background soundtrack, weapon sounds, explosion sounds, and sound effects for buttons and for movements such as dashing. Godot's built-in AudioStreamPlayer node is used to play the sounds.



Fig. 4.37: Node Configuration for Playing Explosion Sounds Using AudioStreamPlayer

4.11 Health Circle

Health circle denotes the current health of the Player. Whenever the Protagonist is hit by an enemy entity, the circle reduces. When the circle vanishes, the Protagonist dies, and the game ends.

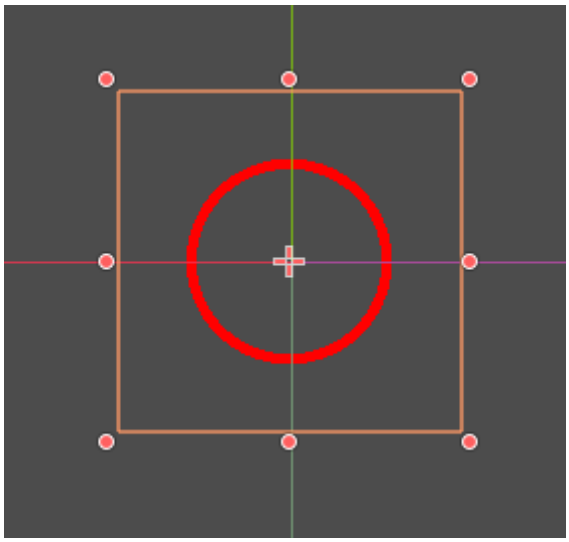


Fig. 4.38: Health Circle Design



Fig. 4.39: Visual Representation of Health Circle Reduction After Player is Hit

4.12 Transition Gates

Transition gates are used to transfer from one scene to another. The Station level features two transition gates that go to the Sky or the Underground scene, whereas the other levels feature a single transition gate back to the Station scene.

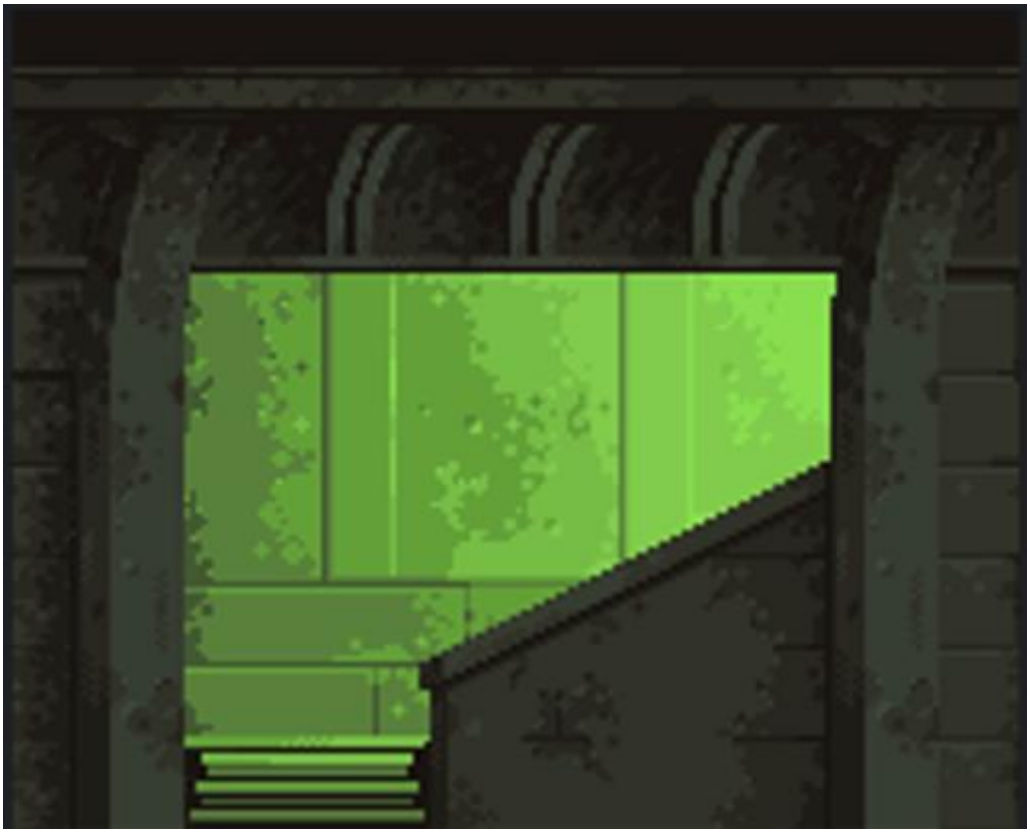


Fig. 4.40: Transition Gate Used to Move Between Levels

CHAPTER 5

DEVELOPMENT AND TESTING

5.1 Implementation Details

This section contains in-depth information about the internal working procedure of the techniques and programming logic used in our video game. It also features a test table for testing and screenshots showing the final outcomes of the project.

5.1.1 Global Settings

Global settings refer to the default values for the most commonly used items. Here are the speed, health, and damage parameters for the enemy entities that are stored. Gun graphics are preloaded to quickly access them and their damage, and speed information is also given. Lastly, the bullet sounds for each type of gun are preloaded for smoother engagements.

```
extends Node

enum guns {AK, SHOTGUN, ROCKET}
const enemy_parameters = {
    'drone' : {'speed' : 110, 'health' : 20, 'damage': 15},
    'soldier' : {'speed' : 50, 'health' : 60},
    'monster' : {'health' : 200}
}

const gun_data = {
    guns.AK: {'damage': 20, 'speed': 200, 'texture': preload("res://graphics/guns/projectiles/default.png")},
    guns.ROCKET: {'damage': 60, 'range': 100},
    guns.SHOTGUN: {'damage': 30, 'speed': 120, 'texture': preload("res://graphics/guns/projectiles/large.png")},
}

var enemy_data: Dictionary

const bullet_sounds = {
    guns.AK : preload("res://audio/ak_shoot.wav"),
    guns.SHOTGUN : preload("res://audio/shotgun_shoot.wav"),
    guns.ROCKET : preload("res://audio/rocket_shoot.wav")
    }

```

Fig. 5.1: Globally Defined Parameters, Accessed by Multiple Scripts Throughout the Project

5.1.2 Main Menu

The main menu allows Players to play, quit, or view the credits of the game once it starts. The game begins by displaying the main menu and playing the background music. When the buttons are focused, they can be pressed to send signals to other nodes and scenes to take the required action.



```
extends Node

func _ready():
    (BgMusic.get_node("AudioStreamPlayer") as AudioStreamPlayer).stop()
    # Grab focus on first button
    get_node("VBoxContainer/Play").grab_focus()
    # Auto-connect focus sounds
    for button in get_node("VBoxContainer").get_children():
        if button is Button:
            button.connect("mouse_entered", func(): _on_button_mouse_entered(button))
            button.connect("focus_entered", func(): _on_button_focus_entered(button))
            button.connect("pressed", func(): _on_button_pressed(button))

func _on_button_mouse_entered(button):
    button.grab_focus()

func _on_button_focus_entered(button):
    $"Button Sounds".play()  # Play hover/focus sound

func _on_button_pressed(button):
    $"Button Sounds".play()  # Play press sound
    await $"Button Sounds".finished # Wait until the sound finishes
    # Then do the action
    match button.name:
        "Play":
            get_tree().change_scene_to_file("res://scenes/levels/station.tscn")
        "Quit":
            get_tree().quit()
        "Settings":
            print("Settings menu coming soon")
func _exit_tree() -> void:
    (BgMusic.get_node("AudioStreamPlayer") as AudioStreamPlayer).play()

```

Fig. 5.2: Main Menu Setup, Featuring Buttons for “Play”, “Quit”, and “Settings”

5.1.3 Player Animation

Player animation is done by using the AnimationPlayer node. The node features an animation Player that is used for regular animation playback. It contains a collection of resources and custom blend times between animation transitions, which can be accessed using the signals [12].

The “Animation” tab in Godot is used to animate the Player character frame by frame.

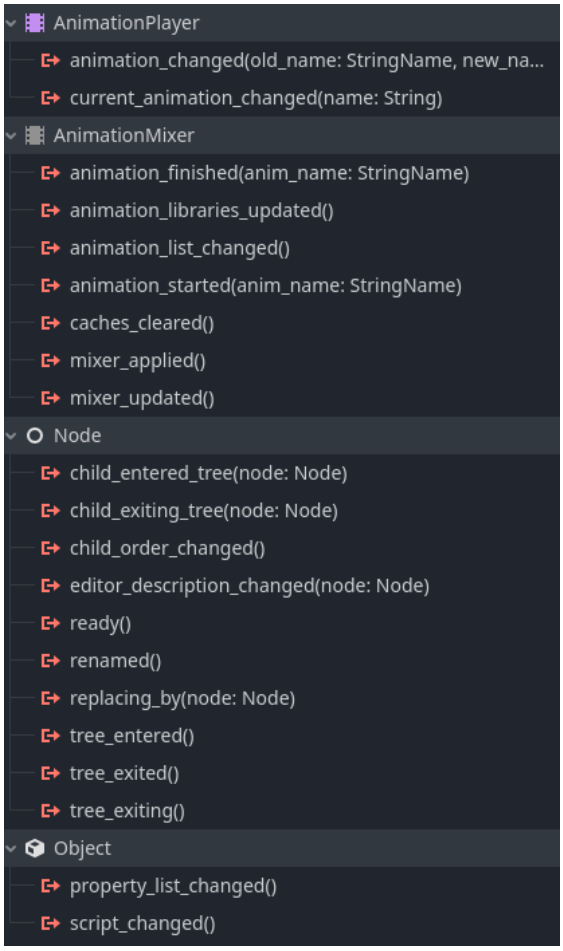


Fig. 5.3: Signals from the AnimationPlayer Node, Used for Communicating to Other Scenes/Nodes About Changes in Animations

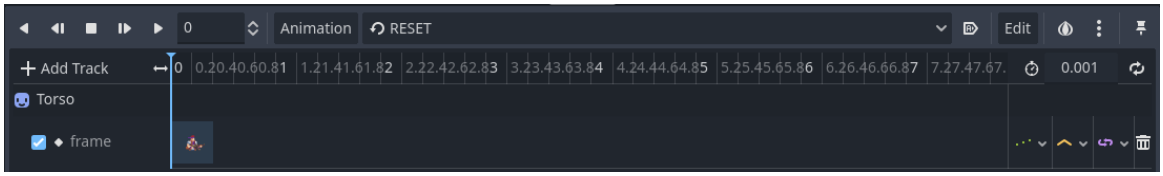


Fig. 5.4: Godot Animation Tab, Used for Generating Frame-by-frame Animation

5.1.4 Player Movement

```gdscript
func get_input():
    # 1.Horizontal Movement
    direction.x = Input.get_axis('left', 'right')

    # 4.Jump
    if Input.is_action_just_pressed('jump'):
        if is_on_floor() or $Timers/Coyote.time_left:#5
            jump = true
        if velocity.y > 0 and not is_on_floor(): #6
            $Timers/jumpBuffer.start()
    # 4.3
    if Input.is_action_just_released('jump') and not is_on_floor() and velocity.y < 0:
        faster_fall = true

    # 7. Dash
    if Input.is_action_just_pressed("dash") and velocity.x and not $Timers/DashCoolDown.time_left: # if pressed and we've movement
        dash = true
        $Timers/DashCoolDown.start()
    # 8. Duck
    ducking = Input.is_action_pressed("duck") and is_on_floor()
    # 9. Aim
    var aim_input_gamepad = Input.get_vector("aim_left", "aim_right", "aim_up", "aim_down")
    var aim_input_mouse = get_local_mouse_position().normalized()
    var aim_input = aim_input_gamepad if gamepad_active else aim_input_mouse
    if aim_input.length() > 0.5:#9.3
        aim_direction = Vector2(round(aim_input.x), round(aim_input.y))
    # switch guns
    if Input.is_action_just_pressed("switch"):
        current_gun = Global.guns[Global.guns.keys()[(current_gun + 1) % len(Global.guns)]]
        print(current_gun)

    # shoot
    if Input.is_action_just_pressed('shoot'):
        shoot_gun()

```

Fig. 5.5: Programming Logic Showcasing the Player Movement, Including Directional Movement, Jumping, and Dashing

Most of the movement is handled by the get_input() method shown in the above figure.

For horizontal movement, the left or the right axis is selected.

When the Player presses the spacebar to jump, the program checks if the Player is on the floor or if the Coyote timer has any time left before jumping. Jump buffer acts as an interval before the Player can jump again. When the Player is jumping, the speed at which they fall to the ground is increased because of the gravitational force.

For the dashing movement, a cooldown timer is used to find out the interval between the last time the dash was used. If the timer is 0, then the Player can dash again.

Ducking requires the Player to be on the floor.

There are input systems for both gamepad and mouse. The mouse input position is normalized to provide more accuracy and a smoother experience overall.

Furthermore, the Player can switch between guns by pressing the middle mouse button. And in order to ensure the selection remains in a given range, the modulus operation is used.

Finally, after pressing the left mouse button to fire a weapon, the control is passed to the shoot_gun() method.

5.1.5 Weapon Shooting

Whenever the Player presses the left mouse button to fire a weapon, the position of the bullet, or projectile release point, is measured using the Player's current position, aim direction, and crosshair distance. Each of the weapons has a reload time, and it is checked to ensure the weapon can fire upon pressing the trigger. Weapons can be shot while jumping, too. The "shoot_gun()" function in Fig. 5.6 defines the logic for shooting weapons.

```
func shoot_gun():
    var pos = position + aim_direction * crosshair_distance
    pos = pos if not ducking else pos + Vector2(0, y_offset)
    if current_gun == Global.guns.AK and not $Timers/AKReload.time_left:
        shoot.emit(pos, aim_direction, current_gun)
        $Timers/AKReload.start()
    if current_gun == Global.guns.SHOTGUN and not $Timers/ShotgunReload.time_left:
        shoot.emit(pos, aim_direction, current_gun)
        $Timers/ShotgunReload.start()
    if current_gun == Global.guns.ROCKET and not $Timers/RocketReload.time_left:
        shoot.emit(pos, aim_direction, current_gun)
        $Timers/RocketReload.start()
        $GPUParticles2D.position = $Crosshair.position
        $GPUParticles2D.process_material.set('direction', aim_direction)
        $GPUParticles2D.emitting = true

    if aim_direction.y == 1 and velocity.y >= 0: #velocity.y >= 0 not needed yet
        gun_jump = true
```

Fig. 5.6: Programming Logic Defining the Weapon Shooting Process, Including Weapon Selection, and Reload Times

5.1.6 Flash Shader

Whenever the Protagonist is hit by an enemy entity, it produces a flash effect where the colors of the silhouette of the Protagonist temporarily change. This effect was made by using a visual shader. Godot's built-in Visual shader tool was used to produce the flash effect.

The flash shader works by taking the base texture as input and blending it with a solid color. It then controls the blend strength with a parameter over time, quickly changing the values from 0 to 1, and then to 0 again, which changes the appearance of the texture.

In our case, the solid color is white.

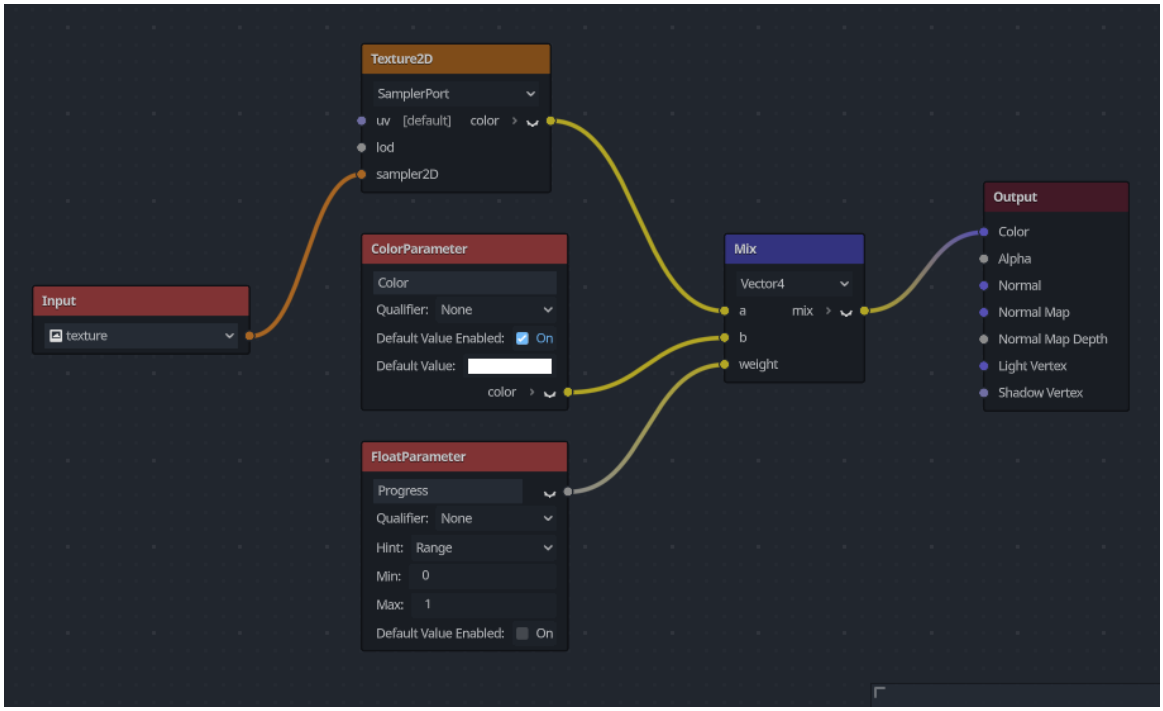


Fig. 5.7: Flash Shader Used to Produce a Flash Effect Whenever the Player is Shot



Fig. 5.8: Flash Shader in Action, Visualizing the Flash Effect

5.1.7 Weapon Selection

We used an AnimationTree node to visually program the weapon selection system. Each of the weapons represents a node. The nodes are connected directionally, which means the weapon selection follows a specific order. For example, we can go back and forth between a rocket and an AK as our selection, but in order to switch to the Shotgun from the AK, we have to go through the Rocket node first. Programming logic is used to cycle between the weapons.

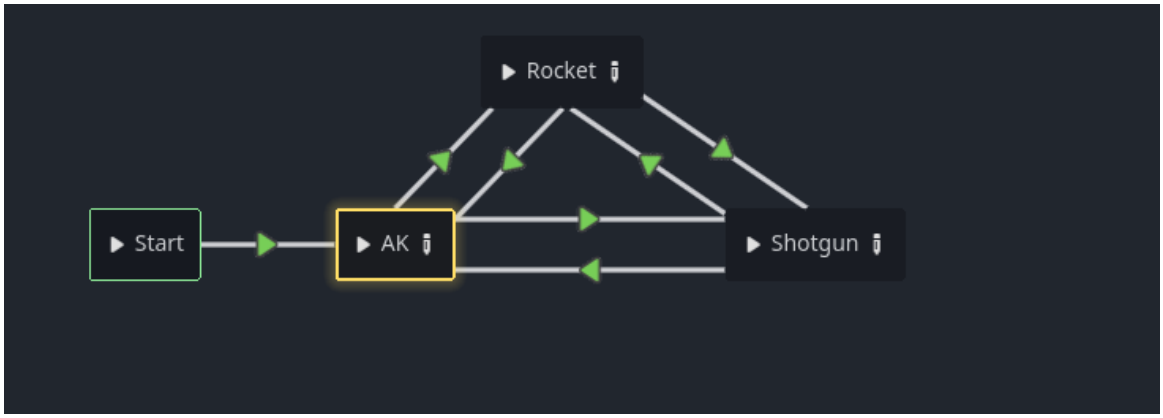


Fig. 5.9: AnimationTree Node Used for Weapon Selection

```
if Input.is_action_just_pressed("switch"):
    current_gun = Global.guns[Global.guns.keys()[(current_gun + 1) % len(Global.guns)]]
    print(current_gun)
```

Fig. 5.10: Programming Logic for the Switching of Weapons

5.1.8 Weapon Animations

Godot's AnimationTree node is used to create the animations for each weapon. It features 8 total points, and animations can be added to each of the points. AnimationPlayer can be used to view the available animations.

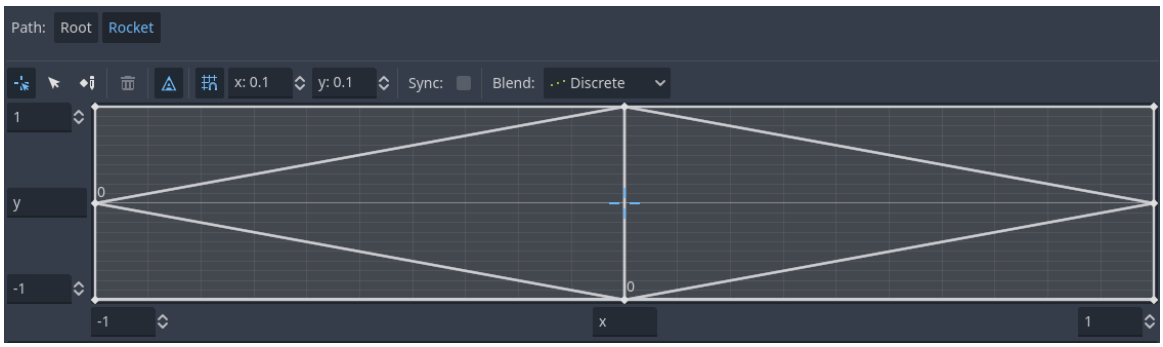


Fig. 5.11: AnimationTree Node, Used for Defining the Points Where the Animation is Played

5.1.9 Health Circle Shader

The health circle displays the current health level of the Player. A shader was implemented to dynamically make the shape of the health circle smaller as the Player loses health points. The health circle shader was hardcoded to fit our use case.

```
shader_type canvas_item;

uniform vec2 center = vec2(0.5, 0.5);
uniform float radius: hint_range(0.0, 1.0) = 0.3;
uniform float width: hint_range(0.0, 1.0) = 0.03;
uniform vec3 color: source_color = vec3(1.0);
uniform float alpha: hint_range(0.0, 1.0) = 1.0; // for the shader visibility
uniform float progress: hint_range(0.0, 1.0) = 1.0; // for showing the reducing health progres


void fragment(){
    float outer_circle = step(distance(center, UV), radius);
    float inner_circle = step(distance(center, UV), radius - width);
    float cutout = step((1.0 - progress) * TAU, atan(UV.x - center.x, UV.y - center.y) + PI);
    COLOR.a = (outer_circle - inner_circle) * alpha * cutout;

    COLOR.rgb = color;

}

```

Fig. 5.12: Programming Logic for the Health Circle Shader

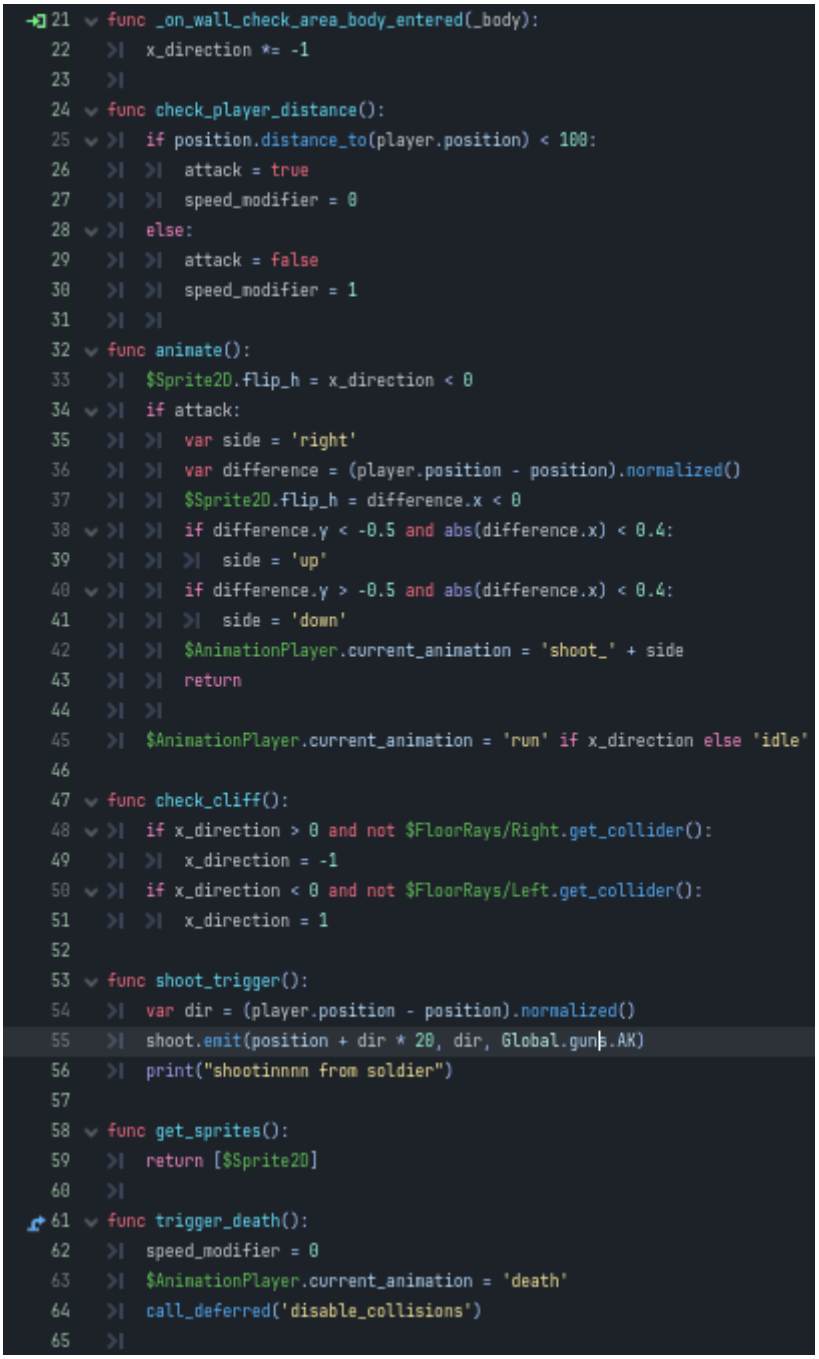
5.1.10 Soldier Engagement

Soldiers are featured on multiple platforms on the Station and Underground scenes. An Area2D node is used to check if the Player has reached the Soldier's area.

Upon checking the distance from the Player, if it's below a fixed unit of 100, the Player is attacked. Otherwise, the Player keeps patrolling its platform. When the Player reaches one end of the platform, its direction is reversed horizontally to make it go back.

The Soldier can only use the AK gun, and its shooting position is calculated based on the Player's position.

When the Player eliminates the Soldier, it triggers the Soldier's death animation.



```gdscript
func _on_wall_check_area_body_entered(_body):
    x_direction *= -1

func check_player_distance():
    if position.distance_to(player.position) < 100:
        attack = true
        speed_modifier = 0
    else:
        attack = false
        speed_modifier = 1

func animate():
    $Sprite2D.flip_h = x_direction < 0
    if attack:
        var side = 'right'
        var difference = (player.position - position).normalized()
        $Sprite2D.flip_h = difference.x < 0
        if difference.y < -0.5 and abs(difference.x) < 0.4:
            side = 'up'
        if difference.y > -0.5 and abs(difference.x) < 0.4:
            side = 'down'
        $AnimationPlayer.current_animation = 'shoot_' + side
        return

    $AnimationPlayer.current_animation = 'run' if x_direction else 'idle'

func check_cliff():
    if x_direction > 0 and not $FloorRays/Right.get_collider():
        x_direction = -1
    if x_direction < 0 and not $FloorRays/Left.get_collider():
        x_direction = 1

func shoot_trigger():
    var dir = (player.position - position).normalized()
    shoot.emit(position + dir * 20, dir, Global.guns.AK)
    print("shootinnnn from soldier")

func get_sprites():
    return [$Sprite2D]

func trigger_death():
    speed_modifier = 0
    $AnimationPlayer.current_animation = 'death'
    call_deferred('disable_collisions')
```

Fig. 5.13: Programming Logic for the Player's Interactions with the Soldier Entity

5.1.11 Drone Engagement

The Drone activates when the Player enters its Area2D node named PlayerDetectionArea. It then chases the Protagonist within the confines of the area. If the Player collides with the Drone, the Drone explodes, leaving behind an explosion animation. Subsequently, the Player's health circle is reduced by the Drone's damage points.

```gdscript
func _on_player_detection_area_body_entered(_body):
	active = true

func _on_collision_detection_area_body_entered(body):
	if body != self:
		blast.emit(global_position)
		body.hit(Global.enemy_parameters['drone']['damage'], body.get_sprites())
		queue_free()

func get_sprites():
	return [$AnimatedSprite2D]

func trigger_death():
	blast.emit(global_position)
	queue_free()
```

Fig. 5.14: Programming Logic for the Player's Interaction with the Drone Entity

5.1.12 Saw Engagement

The Saw enemy entity follows a fixed path once the game is running. It can only move forward or backward. When the Player entity collides with the Saw, it damages the Player's health.

```gdscript
func _ready():
	path = get_children()[-1]
	path_follow = PathFollow2D.new()
	path.add_child(path_follow)
	path_follow.progress = offset

func _process(delta):
	var direction = 1 if forward else -1
	path_follow.progress += speed * delta * direction
	position = path_follow.position

func _on_body_entered(body):
	if 'hit' in body:
		body.hit(10, body.get_sprites())
```

Fig. 5.15: Programming Logic for the Player's Interaction with the Saw Entity

5.1.13 Monster Engagement

The Monster is only able to move up to a certain point with respect to the Player's and the camera's position, which gives the Player a save zone to retreat to.

This enemy entity features a move and an attack timer. Every time the Monster attacks the Player, it faces a timer before it can attack again. And it changes its position vertically each time after the move timer ends.

It has 8 bullet release points and uses the AK weapon for firing at the Player. Only 3 of these bullet points are selected at a time by using a random number generator.

When the Monster is killed, it explodes, and the death animation plays. To display the blast, random number generators are used again, whilst being restricted to a fixed region.

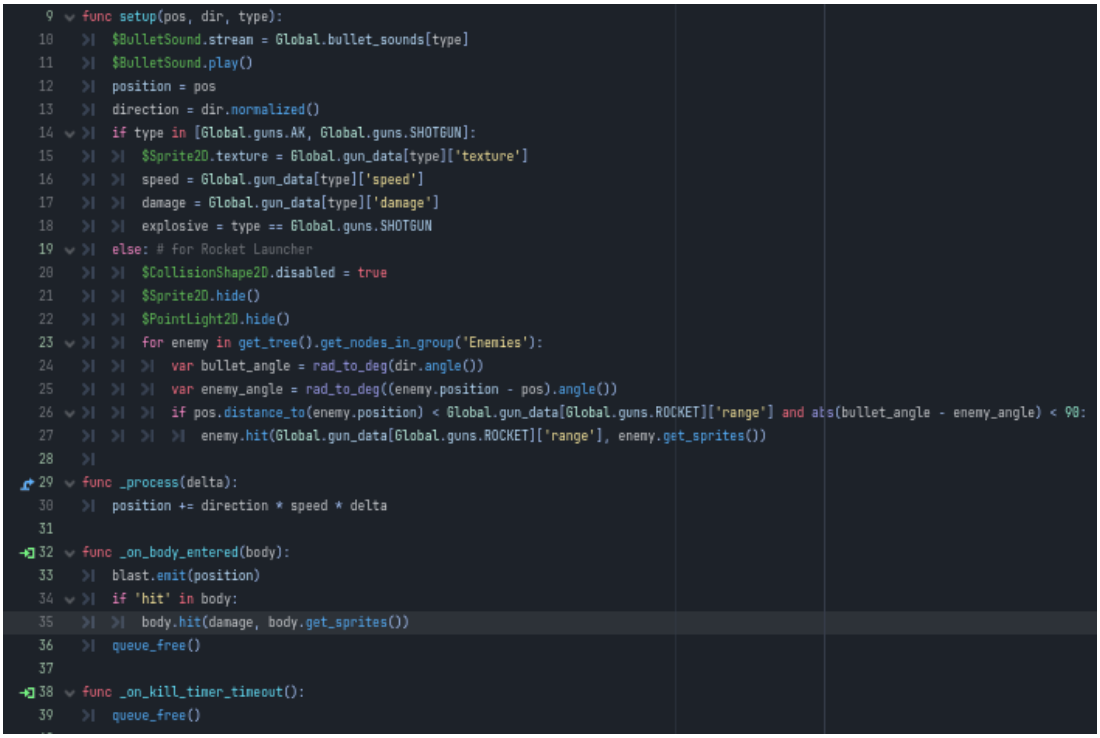
```
18  func _process(_delta):
19      var x = player.position.x + cam_size / 2 - 10
20      x = max(limits.x, min(limits.y, x)) # limiting boundary for movement of the monster
21      var y = player.position.y + y_offset
22      position = Vector2(x,y)
23
24  func _on_attack_timer_timeout() -> void:
25      $AnimationPlayer.current_animation = 'attack'
26      $Timers/AttackTimer.wait_time = rng.randf_range(0.5, 2.0)
27      $Timers/AttackTimer.start()
28
29  func _on_move_timer_timeout() -> void:
30      var tween = create_tween()
31      tween.tween_property(self, 'y_offset', rng.randf_range(y_range.x, y_range.y), 0.6)
32      tween.tween_callback($Timers/MoveTimer.start)
33
34  func trigger_attack():
35      var option_index = rng.randi_range(2, $BulletOptions.get_child_count() - 1)
36      var selected = $BulletOptions.get_child(option_index)
37      for marker in selected.get_children():
38          shoot.emit(marker.global_position, Vector2.LEFT, Global.guns.AK) # (position, direction, g
39
40  func trigger_death():
41
42      explode(position)
43      $Timers/MoveTimer.stop()
44      $Timers/AttackTimer.stop()
45      $AnimationPlayer.play("death")
46
47
48  func return_to_idle():
49      $AnimationPlayer.current_animation = 'idle'
50
51  func get_sprites():
52      return [$Sprite2D]
53
54  func explode(pos):
55      @warning_ignore('narrowing_conversion')
56      var rand_x = rng.randi_range(global_position.x - 20, global_position.x + 20)
57      @warning_ignore('narrowing_conversion')
58      var rand_y = rng.randi_range(global_position.y - 20, global_position.y + 20)
59      blast.emit(Vector2(rand_x, rand_y))
60
61
```

Fig. 5.16: Programming Logic for the Player's Interaction with the Final Boss

5.1.14 Bullet Firing

Whenever a bullet is fired, the bullet sound effect is played first. After that, the type of weapon used is to be found out. In the case of AK, the predefined texture, speed, damage, and explosive types are selected. Otherwise, if the rocket launcher is selected, the distance between the enemy entities and the Player is calculated in order to adhere to a range that causes damage.

When the bullet/rocket enters the area of the enemy entity, it sets up a blast in the case of an AK/shotgun, and an explosion in the case of the rocket launcher.



```
func setup(pos, dir, type):
    $BulletSound.stream = Global.bullet_sounds[type]
    $BulletSound.play()
    position = pos
    direction = dir.normalized()
    if type in [Global.guns.AK, Global.guns.SHOTGUN]:
        $Sprite2D.texture = Global.gun_data[type]['texture']
        speed = Global.gun_data[type]['speed']
        damage = Global.gun_data[type]['damage']
        explosive = type == Global.guns.SHOTGUN
    else: # for Rocket Launcher
        $CollisionShape2D.disabled = true
        $Sprite2D.hide()
        $PointLight2D.hide()
        for enemy in get_tree().get_nodes_in_group('Enemies'):
            var bullet_angle = rad_to_deg(dir.angle())
            var enemy_angle = rad_to_deg((enemy.position - pos).angle())
            if pos.distance_to(enemy.position) < Global.gun_data[Global.guns.ROCKET]['range'] and abs(bullet_angle - enemy_angle) < 90:
                enemy.hit(Global.gun_data[Global.guns.ROCKET]['range'], enemy.get_sprites())

func _process(delta):
    position += direction * speed * delta

func _on_body_entered(body):
    blast.emit(position)
    if 'hit' in body:
        body.hit(damage, body.get_sprites())
    queue_free()

func _on_kill_timer_timeout():
    queue_free()
```

Fig. 5.17: Programming Logic for Handling the Firing of Bullets

5.1.15 Rocket Explosion

Godot's "SpriteFrames" element is used to curate the explosion animation for the rocket launcher. The animation divides up the sprites per frame to give off a smoother effect.

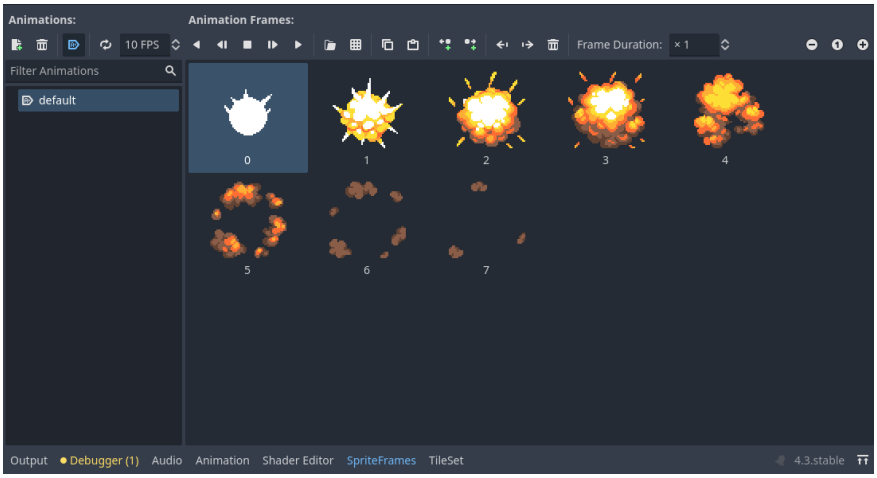


Fig. 5.18: “SpriteFrames” Setup for Rocket Explosion Animation

5.1.16 Transition Gate

Transition gates contain paths to all the different scenes, and when the Player enters the body of a transition gate, the TransitionLayer node changes the scene to the set level.

```
@tool
extends Area2D
#export_enum
@export_enum('station', 'sky', 'underground') var level = 'station' # setted default level 'station'
var levels = {} # dictionary for pushing levels

func _ready():
    if not Engine.is_editor_hint(): # if not in any editor
        levels['station'] = "res://scenes/levels/station.tscn"
        levels['sky'] = "res://scenes/levels/sky.tscn"
        levels['underground'] = "res://scenes/levels/underground.tscn"

func _on_body_entered(_body):
    TransitionLayer.change_scene(levels[level]) # using transition layer for adding a delay effect


# on @export_enum we're exporting along with storing three name of the levels as strings as our target scene.
# whenever we select one, our player can go there through transition gate
```

Fig. 5.19: Programming Logic for Transition Gates

5.1.17 Camera Shaking Effect

Whenever the Player manages to hit an enemy entity, the camera shakes momentarily to produce a special effect. This effect is produced using the Camera2D and FastNoiseLite nodes.

Firstly, a few variables are used as references to the Camera2D and FastNoiseLite nodes in the scene, as shown in Fig. 5.20. By applying offsets to the Camera2D node, we can

simulate a shaking screen. The FastNoiseLite is a built-in Godot class used to generate procedural noise patterns using the FastNoiseLite library, which makes the camera shaking effect smoother [16].

```
# For camera shaking effect:
var camera2D : Camera2D
var cameraShakeNoise : FastNoiseLite # noise map for camera shaking effect
```

Fig. 5.20: Programming Logic for the Camera Shaking Effect Part 1

The “ready()” function in Fig. 5.21 is called automatically when the scene is active. In this case, it is used to gather the Soldiers’ health values. Then it fetches the Camera2D node of the Player node and creates a new instance of FastNoiseLite for procedural generation.

```
func _ready():
    health = Global.enemy_parameters['soldier']['health']

    camera2D = get_node("../Player/Camera2D") # getting the player camera from scene tree
    cameraShakeNoise = FastNoiseLite.new() # creating new noisemap
```

Fig. 5.21: Programming Logic for the Camera Shaking Effect Part 2

Within the “hit()” function shown in the above figure, a tween is created to animate the camera shaking effect. Tweens are lightweight objects used for regular animation by scripting. They are more useful for animations where the final values are unknown in the beginning [17]. The initial intensity, ending intensity, and duration of the effect are passed on to the “startCameraShake()” method.

```
func hit(damage_amount: int, _sprites):
    if health <= 0:
        return
    health -= damage_amount
    # Show damage number
    DamageNumber.display_damage(damage_amount, damage_numbers.global_position)

    var camera_tween = get_tree().create_tween()
    camera_tween.tween_method(startCameraShake, 5.0, 1.0, 0.5) # initial intensity: 5.0, ending intensity: 1.0, duration: 0.5

    damage_particles.restart()
    damage_particles.emitting = true

    if health <= 0:
        trigger_death()
```

Fig. 5.22: Programming Logic for the Camera Shaking Effect Part 3

Finally, the “startCameraShake()” function in Fig. 5.23 changes the horizontal and vertical positional values of the Camera2D node to different intensity levels over time for a limited time frame to produce the effect.

```gdscript
func startCameraShake(intensity: float):
    # getting a float value from the noisemap and storing the return value in the offset variable
    var camera_offset = cameraShakeNoise.get_noise_1d(Time.get_ticks_msec()) * intensity
    # the value will increase/decrease by the intensity level
    camera2D.offset.x = camera_offset # this will move the camera slightly to create the effect
    camera2D.offset.y = camera_offset
```

Fig. 5.23: Programming Logic for the Camera Shaking Effect Part 4

5.1.18 Damage Particle Effect

After the Soldier is hit by the Player, white particles are scattered behind the body of the Soldier that signify the damage particle effect. The effect is applied using the GPUParticles2D node and making changes to the properties of the node in the Inspector section of Godot.



Fig. 5.24: The Damage Particle Effect in Action, Particles Emit from the Soldier's Body After Being Hit

```gdscript
# Damage Particles:
@onready var damage_particles: GPUParticles2D = $DamageParticles
```

Fig. 5.25: Programming Logic for the Damage Particle Effect Part 1

Firstly, a reference variable to the GPUParticles2D node is created as shown in Fig. 5.25. This variable allows us to keep track of the damage particles that are emitted and change their characteristics.

```gdscript
func hit(damage_amount: int, _sprites):
    if health <= 0:
        return
    health -= damage_amount
    # Show damage number
    DamageNumber.display_damage(damage_amount, damage_numbers.global_position)

    var camera_tween = get_tree().create_tween()
    camera_tween.tween_method(startCameraShake, 5.0, 1.0, 0.5) # initial intensity: 5.0, ending intensity: 1.0, duration: 0.5

    damage_particles.restart()
    damage_particles.emitting = true

    if health <= 0:
        trigger_death()
```

Fig. 5.26: Programming Logic for the Damage Particle Effect Part 2

Within the “hit()” function in Fig. 5.26, the reference variable is set to restart every time the function is called. And the “emitting” property of the GPUParticles2D is set to “true,” which emits the dash particles.

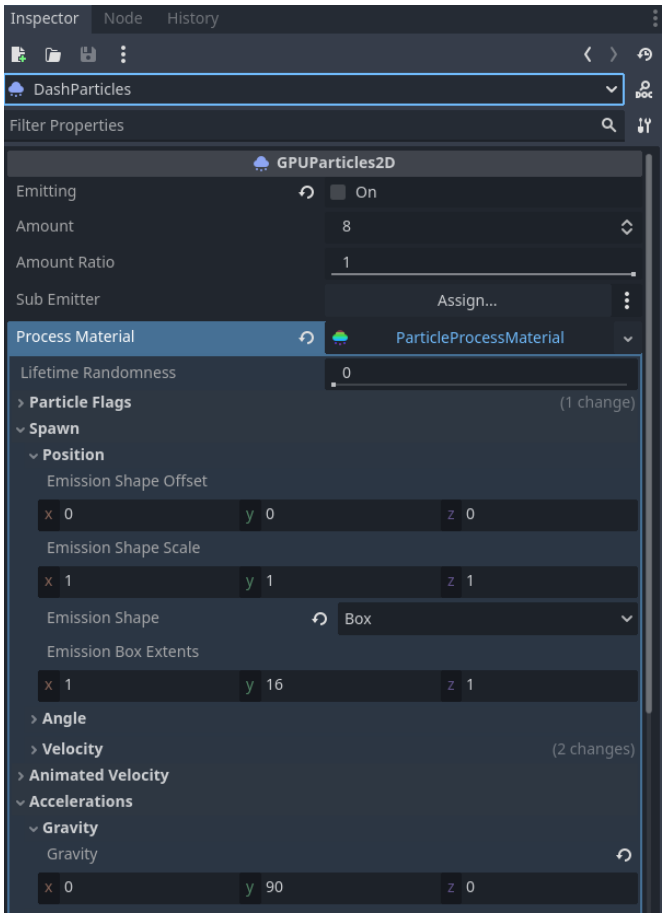


Fig. 5.27: Inspector Section of GPUParticles2D, Featuring Properties Such as Particle Amount, Spawn Position, and Gravity

In the “Inspector” section, we can change the characteristics of the emitted particles by changing the properties such as amount, amount ratio, etc. The “Position” flag denotes the initial positions of the particles. And the “Gravity” flag is used to emulate the gravity present in the physical realm.

5.2 Challenges Faced

This section details the major difficulties faced during the implementation of the project. Overcoming these challenges allowed us to make the game more feature-rich.

5.2.1 Visual Node Selection

Understanding all the visual nodes available and choosing appropriate ones that suit our needs.

5.2.2 Node Interconnection

Combining all the nodes and setting up their interconnectivity to achieve the proper outcome we want in a relevant and efficient manner.

5.2.3 Animation Timing

Animating frame by frame can be complex, and the slightest variations in timing can lead to a subpar animation.

5.2.4 Signal Integrity

Sending the right signal at the right time can be a challenging process to deal with.

5.3 Testing and Results

This section provides some test cases and screenshots from the key moments of the game in order to display the outcome of our project.

5.3.1 Test Table

Test tables are useful for providing a methodology for testing. They are used to evaluate test cases, or a selection of regular, extreme, and exceptional data intended to be used during testing. The expected result is compared against what actually happens during the program’s execution to analyze the functionalities of the piece of software [18].

Test Case	Expected Result	Actual Result	Status
Player Jumps	The Protagonist jumps with coyote time.	Success	Pass

Player is Hit by a Bullet	Player loses health points, and the health circle reduces; flash effect is shown.	Success	Pass
Player Collides with the Drone	The Drone's explosion animation is shown, and the Player loses health points; the health circle reduces.	Success	Pass
Player Hits Soldier	Soldier reduces health points; damage number and damage particle effects are shown.	Success	Pass
Soldier Detects Player	The Soldier fires a weapon at the Protagonist from a distance.	Success	Pass
Drone Detects Player	The Drone starts chasing the Player.	Success	Pass
Drone is Shot by a Rocket Launcher	The Drone is destroyed, and an explosion animation is shown.	Success	Pass
Player is Hit by Saw	Saw keeps moving; Player's health points are reduced.	Success	Pass
Player Presses Middle Mouse Button	The weapon is switched.	Success	Pass
Player Enter Transition Gate	Player changes scene to Sky, Station, or Underground level.	Success	Pass
Monster Detects Player	Monster follows the Player character across the environment; shoots bullets from 3 points, and moves up and down.	Success	Pass

Table 5.1: Test Table

5.3.1 Output

After the Soldier entity is hit by the Protagonist's bullet, its lost health point is displayed as text, as shown in Fig. 5.28.



Fig. 5.28: Soldier Showing Damage Points After Being Hit by the Player

In Fig. 5.29, the Shotgun weapon produces a unique explosion every time it hits the enemy entities.

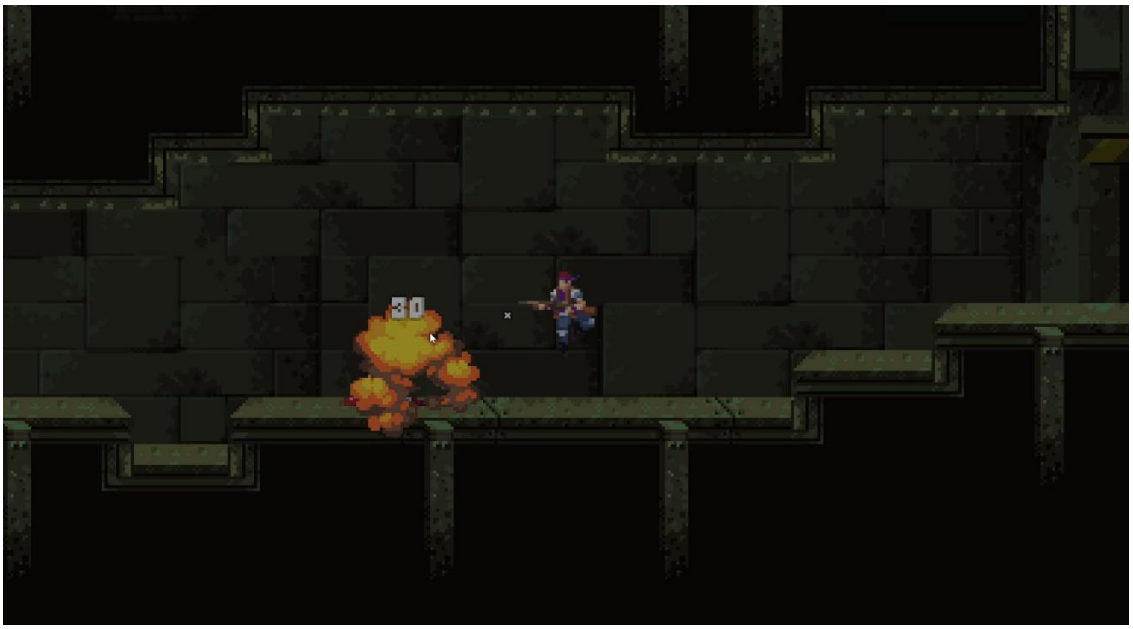


Fig. 5.29: Explosion Animation After the Player Hits the Soldier Using the Shotgun

The Drone entity in Fig. 5.30 is only affected by the Rocket Launcher weapon. It explodes when hit by a rocket.



Fig. 5.30: The Player Hits the Drone Using the Rocket Launcher

In Fig. 5.31, the Protagonist's health circle is reduced when it is hit by an enemy entity. A flash effect is produced to signify the Protagonist being hurt.

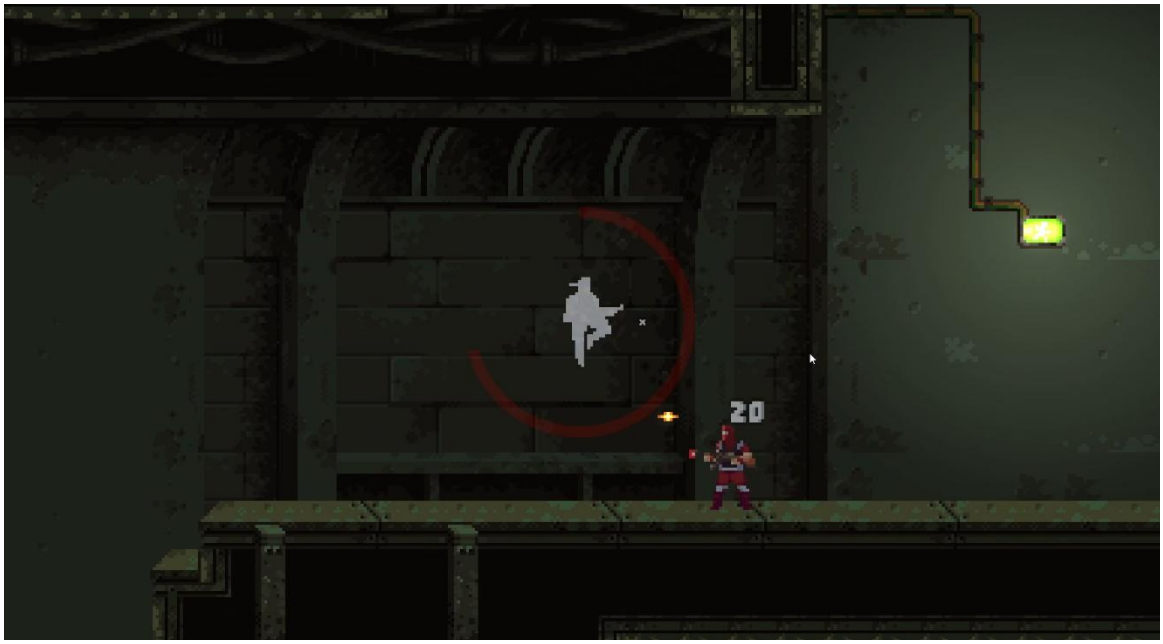


Fig. 5.31: Visualization of the Health Circle Reduction with Flash Effect After Player is Hit by the Soldier

The Saw entity colliding with the Protagonist results in lost health points, as shown in Fig. 5.32.



Fig. 5.32: Visualization of the Player Losing Health After Getting Hit by the Saw

In Fig. 5.33, the Monster entity can’t move beyond a certain platform. It gives the Player an opportunity to retreat to a safe spot when health is running low.

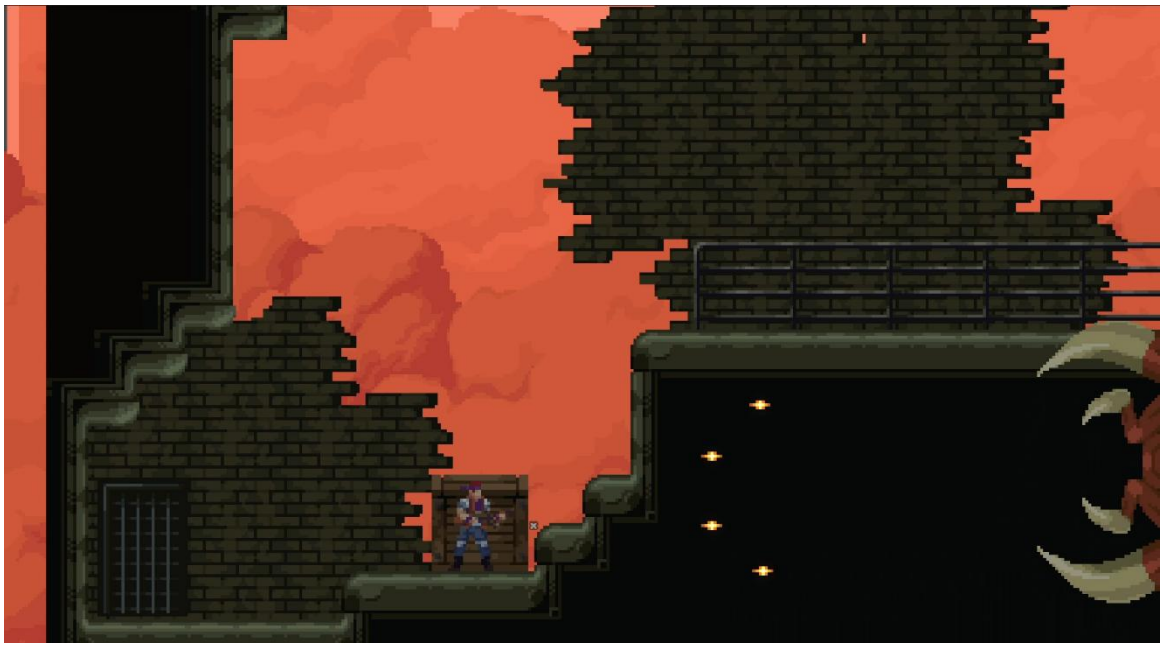


Fig. 5.33: Player Retreats to the Safe Zone When Fighting the Monster Entity

When the Monster entity loses all its health points, the death animation for that entity is shown in Fig. 5.34.



Fig. 5.34: Death Animation of the Monster Entity is Shown After Being Killed by the Player

CHAPTER 6

CONCLUSION AND FUTURE WORKS

6.1 Limitations

This section outlines the key challenges and various limitations discovered during the implementation of the project, including simplified visuals, limited gameplay mechanics, and restrictive player movement.

6.1.1 Development Constraints

Due to the time constraint and resource limitations, the game offers limited features.

6.1.2 Visual Fidelity

2D games are not as detailed graphically. They offer less realistic lighting and visuals.

6.1.3 Simplified Visualization

The game offers simplified visuals since most of the assets are pre-made and gathered from different sources, which limits their customizability.

6.1.4 Restrictive Player Movement

The movement of the Player characters is restricted to two directions in 2D games, along the horizontal and vertical axes. Thus, the game doesn't offer the 3D movement style of the real world.

6.1.5 Limited Playtime

A smaller gameplay area and having a few levels result in the playtime being limited.

6.1.6 Lack of Advanced Features

Due to time constraints, advanced features such as save systems, level progression, or narrative elements like dialogue were not implemented.

6.2 Conclusion

This project successfully demonstrated the development of a 2D Action Platformer using the Godot engine and free assets. Key mechanics such as dashing, ducking, weapon switching, and enemy AI were implemented using Godot's node system and GDScript. Visual effects like flash shaders and health circle shaders enhanced player feedback. The

project highlights the feasibility of indie game development with accessible tools and emphasizes the importance of structured design and iterative testing.

6.3 Future Works

This section discusses potential improvements to the project that can be implemented in the future. Adding these changes will make the game more polished, playful, and engaging for Players and address the current limitations.

6.3.1 Story-driven Approach

Having a story-driven narrative will allow players to connect to the characters and the game more deeply, thus making it more immersive.

6.3.2 Quest Systems

Quests can be designed to allow players to collect resources or get extra rewards. Side quests tend to offer new characters or unveil important details about existing ones, which makes the story deeper. Additionally, some side quests can be designed to be more complex, which incentivizes players looking for difficult challenges [19].

6.3.3 Power-ups

Random power-ups throughout the game can be given to players after killing a random number of soldiers, such as getting loot points or rejuvenating the health of the player partially or fully. It can also be used to increase bullet speed, gun reload time, or gun damage capability.

6.3.4 Multiple Character Implementation

Multiple characters can be unlocked when the default character completes a certain number of quests.

6.3.5 More Content

Additional levels, enemy types, and weapons can be introduced to expand the game.

6.3.6 Particle Effects

Enhancing explosions and environmental effects can make the game more immersive.

6.3.7 Sound Design

A more dynamic soundtrack and contextual sound effects will upgrade the Player's experience.

6.3.8 Enhanced Features

Make the scenarios look like 2.5D. 2.5D is a style of gameplay or movement in a video game or virtual reality environment that is constructed using a 2D plane with virtually no access to a third dimension, although it exists in a space that resembles three dimensions visually. 2.5D is often simulated and rendered in a 3D environment digitally [20].

6.3.9 Visual Improvements

Focus on making the art and animation more visually appealing to offer a more immersive experience.

6.3.10 Cross-Platform Compatibility

Make sure the game is compatible with different platforms, or operating systems such as Android, iOS, macOS, and Linux.

REFERENCES

[1] Magic Media Studio, "Understanding the Technical Differences Between 2D Games and 3D Games", October 1, 2024, online: https://magicmedia.studio/news-insights/understanding-the-technical-differences-between-2d-games-and-3d-games/

[2] Ross Bramble, "The Ultimate Guide to 2D Game Genres", March 14, 2023, online: https://gamemaker.io/en/blog/2d-game-genres

[3] SplicedOnline Team, "Why 2D games are better?", November 28, 2024, online: https://splicedonline.com/why-2d-games-are-better/

[4] Vandana Abrol, "Choosing Between 2D and 3D Games: Pros and Cons", April 1, 2025, online: https://digittrix.com/blogs/choosing-between-2d-and-3d-games-pros-and-cons

[5] Mainleaf, "2D Game Development: The Ultimate Guide to Building Engaging Adventures, February 1, 2025, online: https://mainleaf.com/2d-game-development-the-ultimate-guide-to-building-engaging-adventures/

[6] Glitch Guru, "Game Engines: A Comparative Analysis", January 21, 2024, online: https://medium.com/@GlitchGuru/game-engines-a-comparative-analysis-ef9af01f125e

[7] Wikipedia, "Platformer", July 13, 2025, online: https://en.wikipedia.org/wiki/Platformer

[8] Lucas Kelleher, "Defining the Platformer genre", March 04, 2024, online: https://www.kelleherbros.com/blog/2024/3/4/defining-the-platformer-genre

[9] Godot, "Overview of Godot's Key Concepts", n.d., online: https://docs.godotengine.org/en/latest/getting_started/introduction/key_concepts_overview.html

[10] Godot, "Node – Godot Engine (stable) documentation in English", n.d., online: https://docs.godotengine.org/en/latest/classes/index.html#nodes

[11] Godot, "GDScript Reference – Godot Engine", n.d., online: https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/gdscript_basics.html

[12] Lucidchart, "What is a Flowchart – Lucidchart", n.d., online: https://www.lucidchart.com/pages/what-is-a-flowchart-tutorial

[13] Lucidchart, "What is a Data Flow Diagram", n.d., online: https://www.lucidchart.com/pages/data-flow-diagram

[14] Lucidchart, "UML Use Case Diagram Tutorial", n.d., online: https://www.lucidchart.com/pages/uml-use-case-diagram

[15] Katia, "Understanding Tilesets: Benefits, Types, and Planning for Game Environments", August 15, 2025, online: https://www.animationguides.com/tilesets-in-game-design/

[16] Godot, "FastNoiseLite – Godot Engine (4.4) documentation in English", n.d., online: https://docs.godotengine.org/en/stable/classes/class_fastnoiselite.html

[17] Godot, "Tween – Godot Engine (stable) documentation in English", n.d., online: https://docs.godotengine.org/en/stable/classes/class_tween.html

[18] BBC, "Test tables – Testing and documentation – National 5", n.d., online: https://www.bbc.co.uk/bitesize/guides/zx9vxfr/revision/2

[19] Thespherebog, "What is a Sidequest", October 6, 2024, online: https://thespherebog.com/sidequest/

[20] Wikipedia, "2.5D", March 28, 2025, online: https://en.wikipedia.org/wiki/2.5D