A Deep Dive into Game Development:

Exploring the Intricacies of the Game Development Process

Jared Simonetti

Baldwin Honors Scholar at Drew University

Department of Computer Science, Drew University

Thesis Advisor: Alexander Rudniy

Abstract

In this paper, every step of the game development process is investigated. Five different game projects and prototypes are created to achieve this: including *Pong*, *Asteroids*, *Tetris*, a 2D platformer, and a 3D First Person Shooter (FPS). Each of these games are created using the *Godot* game engine and are programmed using its built-in scripting language: GDscript. This scripting language is used in conjunction with the innate classes of *Godot* to manufacture each game. Throughout this paper, the process behind the development and systems integrated into each game is described. Each game created provides its own unique insights into the evolution of games over time and their increase in complexity. With each game, new tools and methodologies are investigated and used in order to create the various components of each game, contributing to a rounded perspective of the process. Each game created shares common principles, but varies in the way features of gameplay are constructed. This paper is created with an audience of video game players, video game developers, and programmers in mind, making this background beneficial to the comprehension and enjoyment of this paper. The purpose of this paper is to provide understanding of every intricate detail that goes into the process of creating a video game.

Acknowledgements

I would like to thank all the people who helped me along the way, both in writing this thesis and for encouraging me to follow my passions. First and foremost, I would like to thank Professor Barry Burd. It was in his *Introduction to Logic* course, which I had taken in the first semester of my junior year, that I found a passion for programming and computer science. This passion led me to take on the challenge of learning how to code and develop games, later branching to passions in creating digital art through pixel art and 3D modeling. It was that one class that changed the trajectory of my academic career as well as aspects of my personal life.

Thank you to Aaran Robinson, a writing specialist at the University Writing Center at Drew University. In the final revisions of this paper, he had gone through the effort of thoroughly reviewing my paper, which was integral to my writing and revision process.

To my thesis committee, I thank you for your contributions. Professor Alexander Rudniy, who helped to guide my progress over the entire course of this thesis. With his interest in my work I was able to create a thesis on the topics that I love. Professor Barry Burd, for his insights on my paper from the lens of a computer scientist. Professor Adijat Mustapha, for her insights from the discipline of psychology.

To my family, I thank you for believing in me and providing your support for me in my academic pursuits. I could not have done it without you. You are the reason I have gotten to where I am today.

Table of Contents

0 Introduction

Video games have been an integral part of culture for the past half century, with their popularity only seeing an increase over time. They serve as a powerful social construct around which people build relationships with others through the games they share interest in (Skopljakovic, 2019). Popular video games become bastions of social networks in which people of like minded interests are able to find one another, grow friendships, and have healthy social interactions. The core of what draws people to video games is the primary purpose for their construction: entertainment. People play video games for the entertainment that they provide. This entertainment can be derived in many ways, provided the vast array of game genres that exist. A person might derive their pleasure from playing video games that provide increasingly difficult puzzles to complete (e.g. *Portal; Official Portal 2 Website*, n.d.). Alternatively, one might find enjoyment in building something in a 3D game space (e.g. *Minecraft; Official Minecraft Site*, n.d.). Some of the earliest games had a simpler goal, which is to attain the highest score possible, competing against either oneself or others (e.g. *Asteroids*). Many games, from their inception to present day, have continued to provide competition against others as a means of entertainment, pitting people against each other in friendly competition (e.g, *Pong*, *Halo*, *Fortnite*, etc.). Video games come in many forms, each one providing the player with a world to explore and a means to navigate that world. Behind each game is a vast array of systems created for the player to give them the experience of the game, many of which the player is completely unaware of. The goal of this paper is to take a look at how different games are developed from the ground up in order to shed light on parts of the game development process the average player might miss.

0.1 Introduction to Game Development

The process of game development draws from a diverse array of disciplines in order to produce a final product in the form of a video game. The steps needed to develop a full video game include the following: game design, narrative writing, art design/development, sound design/development, and programming. The game designing process provides direction to the game's construction. This process answers the questions of "what can the player do?", "what is the player's objective?", "how does the game end?", and other similar questions. It also provides clarity to the process of building the game.

Narrative writing has a role of variable size depending on the scale of the game in question, but shares a similar role to that of the game design process: instead of providing direction to the construction of the game, though, the narrative writing process adds depth and purpose to the various facets of the game for the player to observe.

Art and sound are two pillars in the development of the video game, with the two constituting everything the player sees and hears in the game. Art and sound are the medium through which the player perceives the game, making their design and development quintessential to the process.

The final step listed here is programming. This is the very thing that brings all other aspects of the game together to form one coherent whole. The programming of a game allows the narrative to be conveyed, sounds to be produced, and art to be presented. The programming of a video game works as instructions for all parts of the game to do as they are supposed to. The program allows the player to interact with the game, giving consequences to the player's actions in ways such as moving a character or firing a projectile. It is this interactivity that separates a video game from any other piece of media such as a movie or a song. Programming lies at the heart of the game development process and is the subject of focus for the topics of this paper.

0.2 Introduction to Game Engines and Frameworks

To create a video game, the most basic means of production is done by working directly with a programming language. You can, in theory, use any programming language to create a video game, however, some of the most popular choices include Python and C++ (*Welcome to Python.Org.*, 2025; *Cplusplus*, n.d.). There are circumstances where the usage of less popular languages have yielded impressive results. One particular example of this is *Rollercoaster Tycoon* (*RollerCoaster Tycoon*, n.d.), which was created entirely in the assembly programming language ("*RollerCoaster Tycoon* (Video Game)," 2024). The language that is used to create a game is highly dependent on the intention behind the game being made, whether it be size, scope, efficiency demands, or even personal experience; all are factors involved in choosing a language to make a game in. Python is a language that is particularly popular to work in due to its ease of use, however, it fails to meet the same level of efficiency as a language such as C++ in most use cases. Likewise, C++ is popular for its high level of efficiency, but has a steep learning curve and slower development process (*Is Python Faster and Lighter than C++?*, 2013). One of the main reasons that a game like *Rollercoaster Tycoon* was made in assembly in the first place is due to a great demand for efficiency with the number of calculations and processes occurring at a given time, something that would be far more difficult to achieve in a higher level language.

Beyond using a stand-alone language to create a video game, another choice is to go down the route of using a game framework or engine to take on some of the burden of the development process. The purpose of a framework is to provide a collection of functions and systems to be accessed across many different use cases. Some of the common functions present across many frameworks (e.g. *PyGame, Ren'Py, Love2D*; *Pygame Front Page — Pygame v2.6.0 Documentation*, n.d.; *The Ren'Py Visual Novel Engine*, n.d.; *LÖVE - Free 2D Game Engine*, n.d.) include a process function to run code for every frame the program is run, an input function for reading keyboard and mouse input, and a draw function to display drawable objects. Each framework has its own specialization, *Love2D* in particular being one that specializes in creating 2D video games. *Love2D*, alongside many other frameworks, benefit from being able to combine the usage of different languages for backend processes and program scripting. What *Love2D* does in particular is use Lua (*The Programming Language Lua*, n.d.), a simple to read and easy to understand scripting language, for game developers to write their code while running the processes of the game through their framework, which is entirely made in C++. This allows the framework to exhibit the benefit of a comprehensible scripting language and a highly efficient processing language.

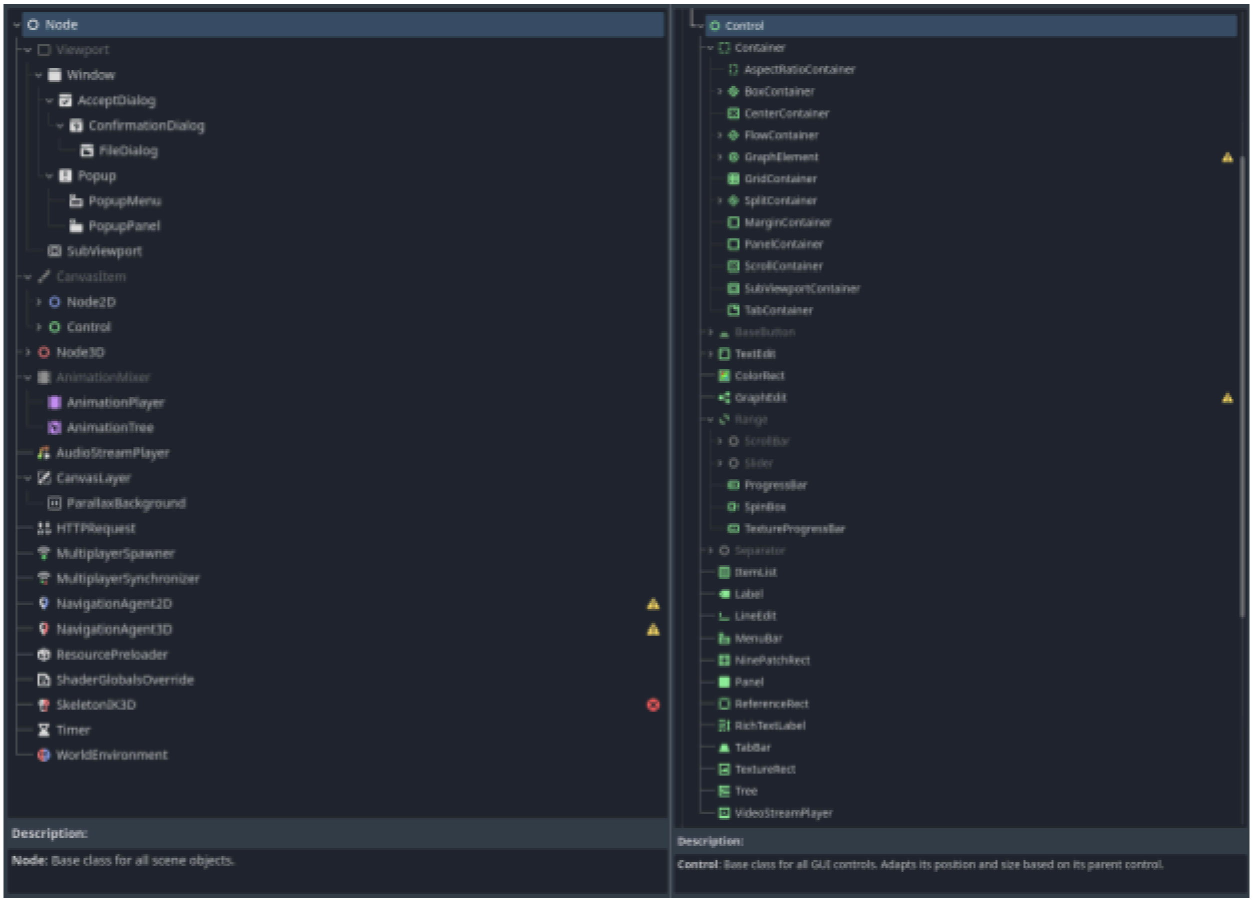
Game engines differ from frameworks due to their provision of an interface and tools that are integrated into the interface. With an engine it is possible for the program to visualize a particular object or game scene without running the project. Depending on the engine, you can also modify values belonging to a particular object and even manipulate its behavior without any written code. A consistency between some of the most popular game engines (e.g. *Unity*, *Unreal*, and *Godot*; *Unity Real-Time Development Platform*, n.d.; *Unreal Engine*, n.d.; *Godot Engine - Free and Open Source 2D and 3D Game Engine*, n.d.) is the usage of an object-oriented programming style. The different features of a game are encapsulated in "objects" that can be freely manipulated in the interface individually. Features of each object can be toggled and modified in the editor and correspond to their behavior when the game is run. This ability to affect game objects without running the game is a benefit that game frameworks do not have.

0.3 Introduction to Godot

Throughout this paper I will be re-creating various video games as well as original prototype games that reflect specific genres as a whole. To make these prototypes, I will be making use of the game engine *Godot* (*Godot Engine - Free and Open Source 2D and 3D Game Engine*, n.d.), an object oriented engine. For this project, the 4.2.2 stable version of *Godot* is used as it is one of the most recent stable versions of the engine to be released at the time of writing. The *Godot* engine is designed for a broad range of games, having an editor for both 2D and 3D game creation. The core building blocks of game creation in *Godot* are called “nodes”. These nodes exist in a parent-child hierarchy, with parent nodes dictating attributes and behavior of child nodes. These nodes come in four different types: base, control, 2D, and 3D (see figures 0.3.1 & 0.3.2).

All nodes inherit properties of the base node, allowing all nodes to interact with one another. It is from this base node that all *Godot* objects are made. Child nodes with branching functionality from the base node are the control, 2D, and 3D nodes. Among these node types, control and 2D nodes share a common parent that is called “CanvasItem”, due to the fact that they share many of the same qualities of a canvas. A canvas is a space in which items are drawn onto and in a game, a CanvasItem is an object that is “drawn” onto the screen. Such drawn objects have many properties such as opacity, color, and z-index. The opacity determines the degree of transparency of the object, the color determines the object’s capacity to contain the colors red, green, and blue in its pixels, and the z-index determines the order in which the object is drawn, with the last item being drawn appearing in front of all other items.

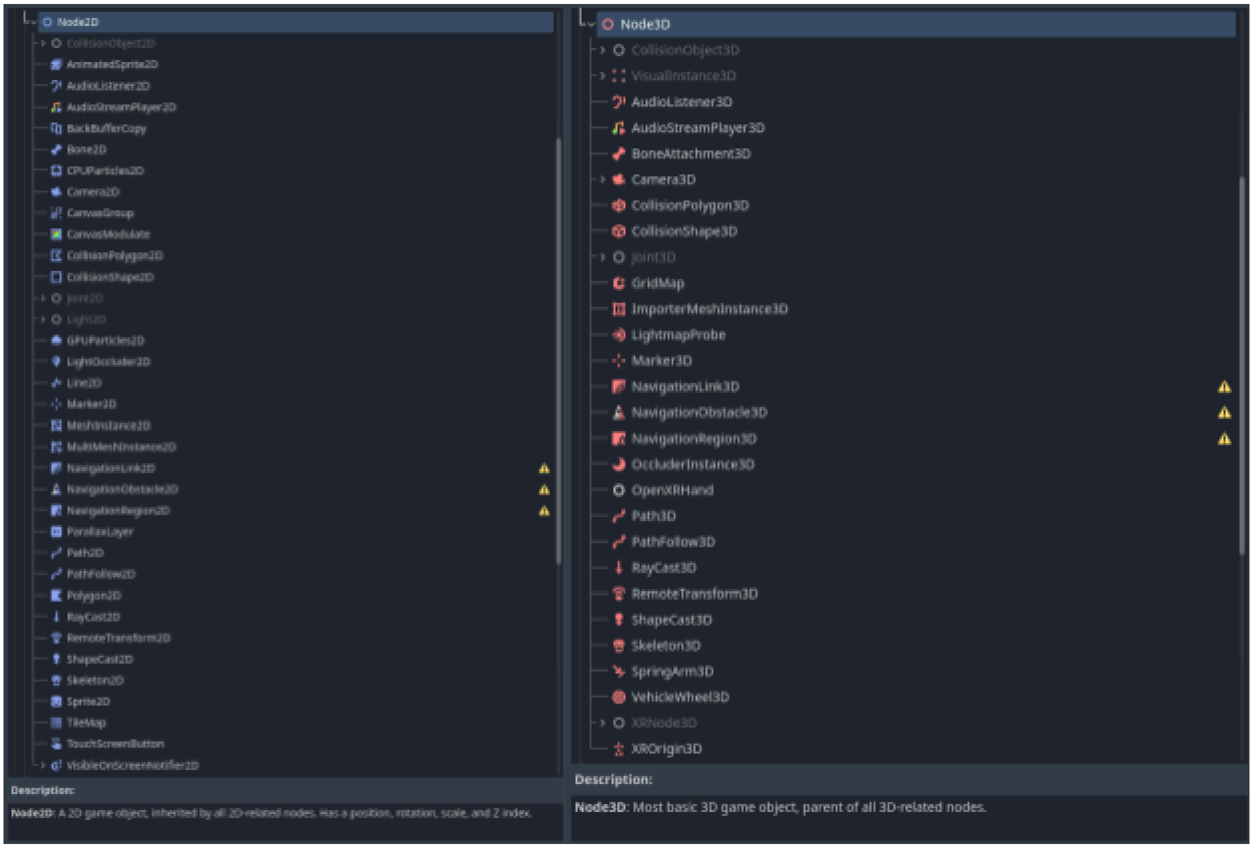
Figure 0.3.1: Base & Control Nodes



The base nodes of *Godot* can be seen on the left while the control nodes can be seen on the right.

For both control and 2D nodes (see figures 0.3.1 & 0.3.2), features belonging to canvas items are available to manipulate. However, despite these shared features, the two nodes and their children are distinct by a significant margin. Control nodes are designed for the purpose of creating a graphical user interface (GUI), which is presented to the player in the form of an icon, a button, text, and more. These nodes are used in menus, dialogues, and to represent player information such as inventory, health, currency, or points. This differs greatly from the expressed purpose of 2D nodes, which are used to compose the objects/elements of a 2D game scene. This includes the player, the background, the enemies, and the other various features of a 2D environment. Control nodes have properties and methods centered around alignment and anchoring, which are valuable in areas such as web page design in which such attributes are actively manipulated and refined to achieve quality user experience (UX). Likewise, 2D nodes are best used as game elements due to the great deal of properties and methods that allow for flexible placement and movement of objects along two axes. The last major type of node to consider is the 3D node. The 3D node breaks away from the control and 2D nodes due to its three dimensional nature. This brings a great deal more to consider, as a z-axis is added to what was formerly just an x-axis and a y-axis. With this said, however, if one were to compare the types of 3D nodes to the types of 2D nodes (see figure 0.3.2), it becomes apparent that the vast majority of 3D nodes have a 2D counterpart. This is because the purpose of 3D nodes is the same as 2D nodes—to compose the objects/elements of a 3D scene. The majority overlap is due to the common features that 2D and 3D games have, therefore necessitating many of the same types of nodes to create their respective scenes.

Figure 0.3.2: 2D & 3D Nodes



The 2D nodes of *Godot* can be seen on the left while the 3D nodes can be seen on the right.

Beyond the types of nodes that *Godot* provides, *Godot* also has two primary files that compose the game. One has already been mentioned on several accounts, which is the game “scene”. Game scenes are files which store one or more nodes in a parent-child hierarchy, with exactly one node at the top of the hierarchy. These scenes are stored in files with the “.tscn” extension; within their content the hierarchy of nodes and their respective attributes are stored. Among these attributes is the second primary type of file used to make a game, which is the script file. The scripting language of the *Godot* engine is called GDScript (*GDScript Reference*, n.d.), denoted with the “.gd” extension. This language is used to interface with the methods and properties of each node through the engine which is constructed in C++. Beyond using the built-in methods of the engine, the scripting language can be used to create custom properties, methods, and algorithms freely. Each node can have only one script attached to it, with each script capable of dictating the behavior of its node, child nodes, parent nodes, and sibling nodes (nodes with the same parent). In theory, each and every node has a means of accessing every other node in an active scene. With this said, nodes with the closest relationships (e.g. parent-child, sibling-sibling) have the easiest access to one another. This is because the fewest number of calls need to be made, with every parent-child pair being able to directly access one another with sibling-sibling pairs only needing to go through their shared parent. This gives merit to keep nodes that have information or functions valuable to another node close in the parent-child hierarchy.

In creating a script file, there is a great variety of means to customize code to achieve the desired results, however, there are a few tools built into the *Godot* engine that provide a foundation for programming a game. One of these tools that is commonly used is a combination of three functions, the _ready() function, the _process(delta) function and the _physics_process(delta) function (*Node*, n.d.). These functions are the cornerstones of GDScript and their purposes are in-line with their namesake. The _ready() function performs all instructions within at the very moment the scene is instantiated. In this function values can be initialized, timers can be started, and important starting operations can be performed. The _process(delta) and _physics_process(delta) functions are where the bulk of manipulatable behavior occurs during the runtime of the game. The difference, however, is that _process(delta) function handles all operations except those involving the built in physics engine, while the _physics_process(delta) includes operations involving the physics engine. Functionally, these two processes are one in the same in the manner in which they run code and are both addressed as the process step. Both of these functions are being called every frame and are often filled with miscellaneous check conditions for directing behavior. Each check condition, when fulfilled, runs a different portion of code created. If a particular operation needs to standardize behavior to be consistent over time, then the argument provided to the functions “delta” is used. This delta value captures the time passed between frames so that it can be used to standardize game operations, regardless of the frame rate the game is running at.

Another technique that can be implemented is the provision of various methods to a given scene. Methods are mainly responsible for checking, getting, and setting data of a particular object. If a scene is the child of another scripted scene, it would be more efficient to provide methods to the child scene for the parent scene to use in order to improve the efficiency of the game and modularize the code better.

One more technique that is often key to *Godot* game development is the usage of “signals”. The signal is one of the preferred methods of sending information upwards in the node hierarchy. Signals can either be custom made by setting the signal on ready and emitting the signal provided the right condition or by deriving it directly from certain nodes which have pre-made signals. The “timer” node is one good example of pre-built signals, as it comes with the “_on_timer_timeout” signal by default. As its name suggests, this signal is sent out when the corresponding timer reaches a time of zero. This signal can then be connected to a node higher up in the hierarchy which can use the signal to make certain decisions and perform appropriate operations.

There are nearly limitless possibilities on how a person may construct a game using the *Godot* engine, or any game engine for that matter, but these are some of the methods that see common use and will be used in the projects conducted in this paper (_ready, _process(delta), _physics_process(delta)).

1 *Pong*

Pong (*"Pong,"* 2025) is among the earliest video games to be produced and was a novel creation for its time. The premise of the game is very simple: the game is played with two players, one who controls a paddle on the left side of the screen and the other who controls a paddle on the right side of the screen. Each paddle moves at a set speed, up or down according to player input. The goal of the game for each player is to stop the ball that bounces around the screen from getting past the paddle that they control. Likewise, the player also tries to make it harder for the opponent to block the ball with their paddle. The way that the player is able to do this is by strategically blocking the ball with different parts of the paddle. The ball moves at the greatest amount of vertical speed when it bounces off of the part paddle furthest from its center. Upon colliding with the upper portion of the paddle, the ball is reflected upward and upon colliding with the lower portion of the paddle, the ball is reflected downward. Earlier in the game, it is possible to get from the top to the bottom of the screen with the paddle in the time it takes the ball to traverse the screen. Provided with the mechanics established, this would make for games that could technically go on forever. This is due to the lack of a changing variable that increases the difficulty of the game beyond the threshold of the players' capability. As a result, there is one more mechanic that comes into play, which is the acceleration of the ball. Upon colliding with the paddle, in addition to reflecting at different angles based upon the point of collision, the ball accelerates by a small margin. This margin is hardly noticeable upon the first few collisions, but the accumulation of collisions causes the difficulty to exponentially rise for both players until the round is won.

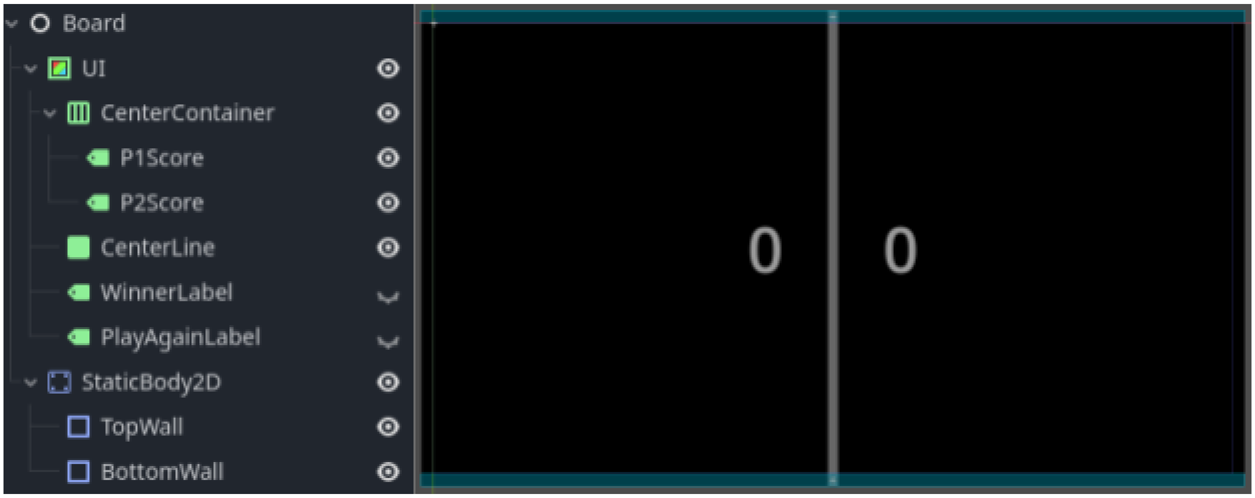
Upon starting the game, each player has a score of zero. Each round begins with a button being pressed to initiate the movement of the ball, or "serving". After serving, both players do their best to meet the ball with the paddle on their side of the board. Upon the ball crossing the opposite player's side of the screen, a point is awarded to the player. This repeats until one of the players reaches a score of 11, in which that player is announced the winner via printout to the screen.

When playing *Pong*, gameplay comes in the form of a player versus player centered design. You are given the expressed purpose of defeating your opponent and are given the means to do so through the usage of paddles in a table tennis-like environment. It is the skill of the player, the skill of their opponent, and the player's knowledge of the opponent that contribute heavily to gameplay.

1.1 Developing Pong

Pong has three main components: The board, the paddle, and the ball. First we have the board, which is the space in which gameplay is occurring. The board is a very simple object, but it contains pieces of data and visual representations valuable to gameplay. In the iteration of *Pong* created for this project, the board is constructed with a combination of several control and 2D nodes. The control nodes include two labels for the score of each player as well as two more labels to print out the winner at the end of a game (WinnerLabel) and a label to ask whether they would like to play again (PlayAgainLabel). This makes 4 labels total, with the visibility of the WinnerLabel and the PlayAgainLabel changing according to the state of the game. The 2D nodes that are part of the board scene are two rectangle shapes that are children to a "StaticBody2D" node (see figure 1.1.1). These two shapes can be seen below as the light blue bar at the top and bottom of the board. These shapes that constitute the static body would act as bumpers for the ball to bounce off of, and as a limiter for the paddle to be stopped by.

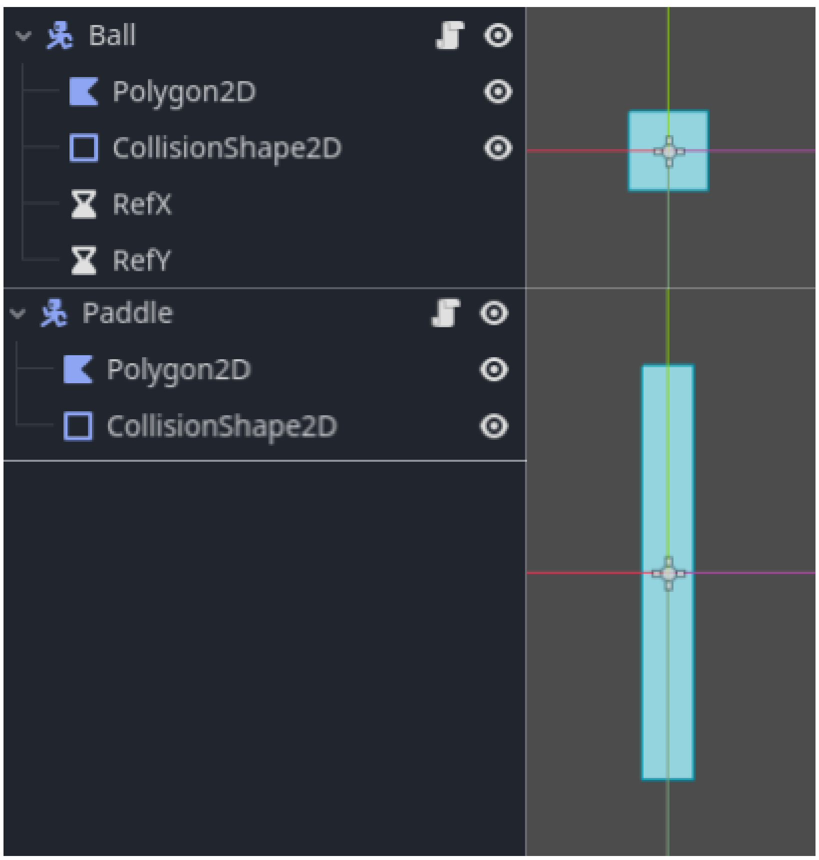
Figure 1.1.1: Board Scene



The board scene can be seen above on the right and the node hierarchy of the board scene can be seen on the left.

The ball and paddle scenes have smaller node structures than the board scene, however, their attached scripts provide each scene with more complex behavior. The ball scene is composed of 4 nodes, a white polygon in the shape of a square, a collider with the same shape and size, and two timers. The paddle scene is composed of 2 nodes, a white polygon in the shape of a tall rectangle and a collider with the same shape and size (see figure 1.1.2). What makes both of these scenes unique is the characteristics of their root node. At their root, both scenes have a "CharacterBody2D" node, a node which contains all of the characteristics a character might have in a game. Both the paddle scene and the ball scene are character body nodes, meanwhile only the paddle is a player controlled character. The reason for this choice is because the character body is in actuality a kinematic body, an object with manipulatable speed and direction that is affected by code and not the physics engine. The qualities of the character body made this node type suitable for both the ball and the paddle. There are two other types of physics bodies in *Godot*, the rigid body and the static body, but neither of them would be suitable for the purposes of these scenes. The rigid body relies on the physics engine to calculate its direction and movement, something that is wholly unnecessary for this version of *Pong*, given that the ball's movement is controlled via code (see figure B1.1). Likewise, the static body is also unsuitable since it is an immovable object through neither the physics engine nor code. This makes the static body perfect for the board scene, but not the ball or paddle scenes.

Figure 1.1.2: Ball & Paddle Scenes



Both the ball and paddle scenes can be seen here: The ball scene can be seen above on the top right and the node hierarchy of the ball scene can be seen on the top left. The paddle scene can be seen above on the bottom right and the node hierarchy of the paddle scene can be seen on the bottom left.

The ball and paddle scenes both have scripts attached to them which determine their behavior (see figures B1.1 & B1.2). Both scripts are centered around the primary goal of directing movement. For the ball script (see figure B1.1), the velocity of the ball is changed upon colliding with either the top or bottom of the screen and upon colliding with a paddle. If the ball collides with the top or bottom of the screen, the ball's vertical velocity is flipped, mimicking a perfect reflection/bounce. If the ball collides with a paddle, its vertical velocity will be changed in accordance with the distance between the ball and the center of the paddle and the horizontal velocity will be flipped and multiplied by a factor of 1.05 (increase by 5%). The paddle script is less complicated in terms of manipulating velocity compared to the ball script, rather it deals with control flow. The paddle script (see figure B1.2) has an unchanging set speed and is applied to the paddle's vertical velocity according to input. The paddle is provided with an ID number called "player_num" which is given to it based on which side of the board it is on. The paddle on the left is numbered 1 and the one on the right is numbered 2. The paddle assigned the numbered 1 moves up and down when the "W" or "S" keys are pressed, while the paddle assigned the number 2 moves up and down when the "UP" and "DOWN" keys are pressed. These are respectively read by *Godot* as the "up1"/ "down1" and "up2"/"down2" due to the configured input settings. Provided the appropriate input, the movement of the paddle is simple. The vertical velocity is set to -speed, +speed, or 0 according to the inputs provided.

The final detail to note in the construction of Pong is the main script (see figures B1.3 to 1.5B). In the construction of most games, a main script is utilized to keep track of important values and regulate behavior of objects in ways that those objects could not on their own. In this rendition of *Pong*, the main script is used to manipulate the speed and direction of the ball based on game state and player input. It is also used to store score values and update labels to their appropriate text and visibility. An important detail to note regarding the main script is that its behavior is based around a simple state machine revolving around the variable "state". This variable starts with the value "start" and changes to "play", "game_over", or back to "start" depending on the conditions met. If the game is in the "start" state, the ball sits in the center of the screen until a player presses "enter" or "space", which is set as "accept" through input settings. Once this is done, the game state is switched to "play" and the ball's movement is initiated. Upon initiating its movement, the ball is sent either left or right with an angle ranging from -45 degrees to 45 degrees with respect to the direction it is moving. The game switches back to the "start" state when two conditions are met: the ball crosses the edge of the screen on either the left or right hand side of the screen and neither player has a score that is greater than or equal to 11 (the game is not over). The game's state will flip between "start" and "play" until a point is gained by a player that brings them to 11 points. Once this occurs, the game will enter the "game_over" state, which displays the winner. From this state, a player can press "accept" to reset the game fully and clear the scores, returning to the "start" state (see figures B1.3 to B1.5 for code).

1.2 Intricacies Behind the Development of Pong

While the gameplay mechanics of *Pong* are simple in design, the steps involved in its development contribute to building a base of reference for games going forward. The game laid the groundwork for future game development with a simple translation of the game of ping pong (also known as table tennis) into a digital environment. It was from this translation of the game to a digital frontier that inspired other developers to branch out and create games of their own. *Pong* was among the first games to contribute to the growth of game development as a whole. The features of *Pong* that are significantly worth noting are how it implemented simulation of movement, collision detection, reflection logic, and its impact on the video game industry as a whole.

1.2.1 Simulation of Movement

One of the challenges in early game development was simulating the movement of an object and representing that movement graphically. In *Pong*, simulating the ball's movement requires calculation of the ball's position, speed, and direction. This calculation operates on the ball by judging where it was and where it is moving toward. Upon completion, the calculation is then reflected on the ball by graphically moving it to the new location. This process occurs for the movement of both the left and right paddles, with a more restricted ruleset which locks them to y-axis movement at a static speed. *Pong* lacks advanced graphical capabilities due to hardware limitations at the time of its creation, but still is able to represent movement by lighting select pixels that represent the ball and paddle objects.

1.2.2 Collision Detection and Reflection Logic

The way that the ball's movement is programmed and its interaction with the paddles is the key driving factor to the gameplay of *Pong*. To simulate the physics of bouncing off surfaces, the game requires a minimal detection system for the presence of other objects. When the ball collides with the top or bottom edges of the screen, the vertical velocity of the ball is inverted to simulate a bounce with no loss of momentum. This is implemented in a different manner compared to collisions with the paddles, the reason being that when the ball collides with a paddle, its horizontal velocity is inverted and incremented upward while the vertical velocity is set according to the distance from the center of the paddle that the ball collides with. This reflection is variable, unlike the consistent reflection with the top and bottom of the screen.

The early form of collision detection systems that *Pong* and similar games implemented goes by the name of "axis aligned bounding box" (AABB) collision. This form of collision detection reduces each object to a square with four vertices. If any one of these vertices overlaps, a collision is detected. This form of collision detection is not what was used for the version of *Pong* created in this paper, but can be manually implemented by hand or with the AABB class.

The reflection logic is inherently tied to the game's pacing. As the ball accelerates with each paddle collision, it becomes increasingly difficult for both players to react in time. This is especially so when the ball collides with the edges of the paddle, causing it to reflect at a steep angle. This logic is quintessential for the development of engaging gameplay for *Pong*.

1.2.3 The Impact on the Video Game Industry

Pong is an exceptionally simple game, consisting only of two player controlled paddle objects and a ball with simple movement logic. It can be argued that *Pong* is one of the trend setters for the many games that followed with the way it implemented the game loop, a cycle of states (such as "start", "play", "game over") for players to navigate. This structure has become fundamental in game design, allowing for comprehensible game progression. Games following the example of *Pong* have clear states for the player to interact with in a way that ensures a sense of progression. The use of states and score tracking also laid the groundwork for other game mechanics, such as achievements, leveling systems, and player progression found in modern gaming. It is the systems that were implemented into *Pong* that helped contribute the growth of the early game industry and the developers who supported it.

2 *Asteroids*

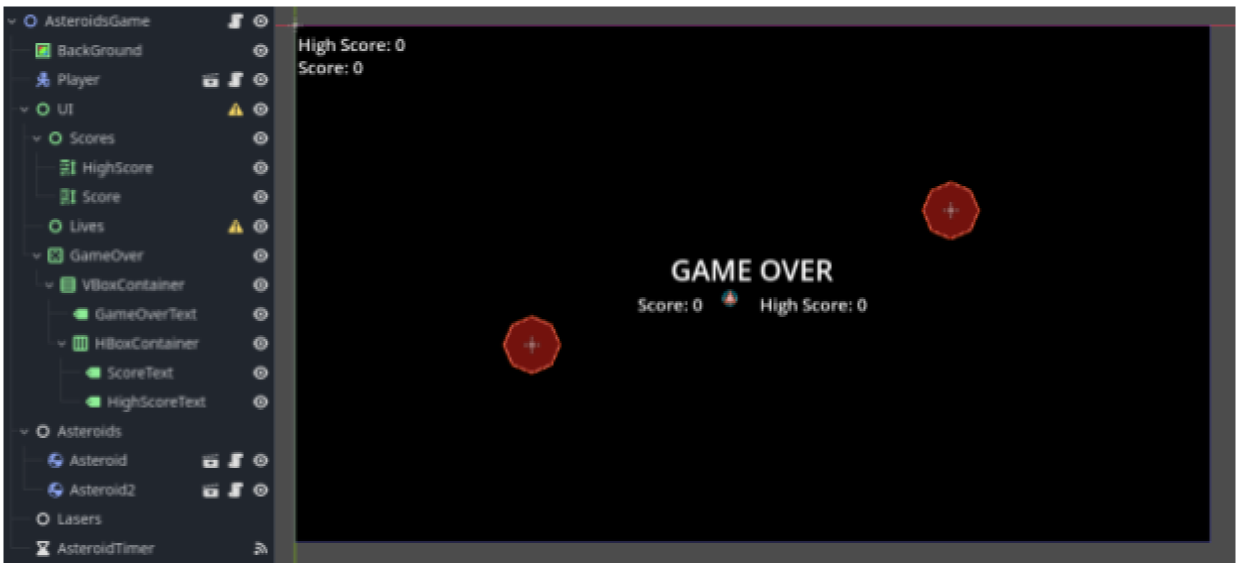
While *Pong* is among some of the first games to be widely released for two players, *Asteroids* (*"Asteroids (Video Game),"* 2025) is one of the earliest produced games for a single player. In *Asteroids*, the player controls a triangle which represents a spaceship. Around the spaceship an assortment of polygonal shapes of various sizes move around at variable speeds. While playing the game, the player is capable of two things, moving and shooting. In terms of movement, the player is capable of only accelerating in the direction the spaceship is facing, while also being able to change this direction by rotating left and right. As far as shooting goes, this action allows a player to "fire" a laser-like projectile in the direction the spaceship is facing. The player's goal is to gain points by shooting as many of the asteroids as they can without getting hit by them, as getting hit would result in the player losing lives. Upon losing all of their lives, the player loses the game and is presented with their score.

For *Asteroids*, the fun of the game comes from playing to achieve the highest score. This attribute of achieving the highest score, as it is documented on a leaderboard, creates gameplay that has social implications, with people playing the game to beat either their own personal best score or the highest score on the leaderboard. The game has functionally unlimited playability with the Player being limited by their level of skill at using the free moving ship they control.

2.1 Developing Asteroids

The game of *Asteroids*, despite its construction being based around single player functionality, is composed of somewhat more complex elements than that of the two player game of *Pong*. The version of *Asteroids* created for this paper is composed of several objects, including the user interface (UI), the player, the lasers that the player fires, and the asteroids. With these elements combined, the game of *Asteroids* is born. First we have the board: this is an object that exists in most games constructed around a single frame of view and is often paired quite closely with the main script (see Appendix A2 for *Asteroids* gameplay images). The "board" for the *Asteroids* game developed is captured by the "AsteroidsGame" scene. Inside this scene lies the background, the player, the UI, as well as containers to hold instantiated objects including the asteroids and the lasers (see figure 2.1.1).

Figure 2.1.1: Asteroid Game Scene

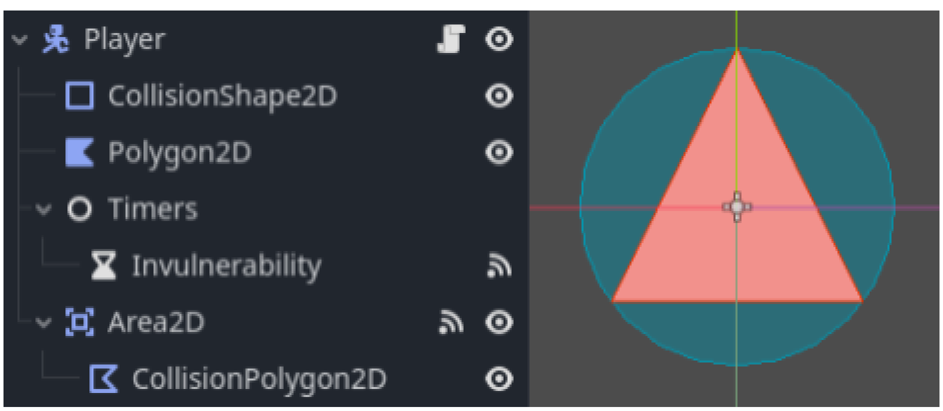


The asteroid game scene can be seen above on the right and the node hierarchy of the asteroid game scene can be seen on the left.

The main script (see figures B2.1 to B2.5) for the game is responsible for several core functionalities, all of which are centered around the operation of a state machine. The state machine for this game is very similar in terms of functionality and labeling to the *Pong* state machine. The states used for *Asteroids* include "start", "running", and "game_over". The game begins in the "start" state, in which the player is paused and invisible. No input will cause any response from the game except for pressing the "ENTER" key. Upon pressing "ENTER", the game shifts to the "running" state in which the player is now visible and interactable. The UI is also toggled to be invisible and the "AsteroidTimer" starts and begins to instantiate new asteroids. In this state, the game checks for the player's position every frame in order to determine whether their position falls outside of the board. If this occurs, the player's position is then set to the opposite side and maintains its velocity upon falling outside the board. The game enters the "game_over" state under one condition: the player's life falling below zero. When the player collides with an asteroid on the screen, they lose a life, which is kept track of in the main script. This is also represented graphically with duplicates of the player in the bottom left-hand corner of the screen (see figures A2.2 & A2.5). Once there is no player icon in the bottom left corner and the player is hit, the game enters the "game_over" state. In this state, the game operates in a very similar fashion to that of the "start" state, the major difference being what occurs when the player presses the "ENTER" key. When the player presses "ENTER", instead of initiating gameplay, the main scene is instead reset. By doing this, with the way the game is constructed, only the high score of the player persists to the next iteration of gameplay.

The main script is responsible for manipulating the behavior of all objects in the game with its focus primarily on the Player, Asteroid, and UI objects. The Player and Asteroid object (see figures 2.1.2 & 2.1.4) have their own script based behaviors and are both responsible for instantiating new objects of their own. To start, the behavior of the player scene and its composition can be described. The Player object has several core components, which include a polygonal shape, an area with a nested collision shape, a collision shape, and an invulnerability timer. Beyond these components, at the root of the scene is a kinematic body with the player script attached. Among the shapes that are children to the player, each serves a different purpose. The polygonal shape is the part of the player character that the person playing the game sees. It is a white triangle on top of the position of the player character. The area with a nested collider serves the purpose of detecting other objects that make contact with the Player. This area reads for collisions with asteroids in particular. The collision shape that is a direct child to the Player would have been used for this purpose, however, due to the lack of this built in functionality in this version of *Godot*, it is not. This collision shape solely exists for the purpose of satisfying the requirements of the "CharacterBody2D" node. This type of node necessitates a collision shape as a child node in order to perform the appropriate operations of managing object position and collision. In this version of asteroids, the Player is designed to not "collide" with any objects. Instead, it detects whether it is in contact with another object, takes damage, and has invulnerability for a short period of time as well as reduced acceleration. The collider that would otherwise bump into other objects, halting momentum, is set to not mask/read other objects.

Figure 2.1.2: Player Scene

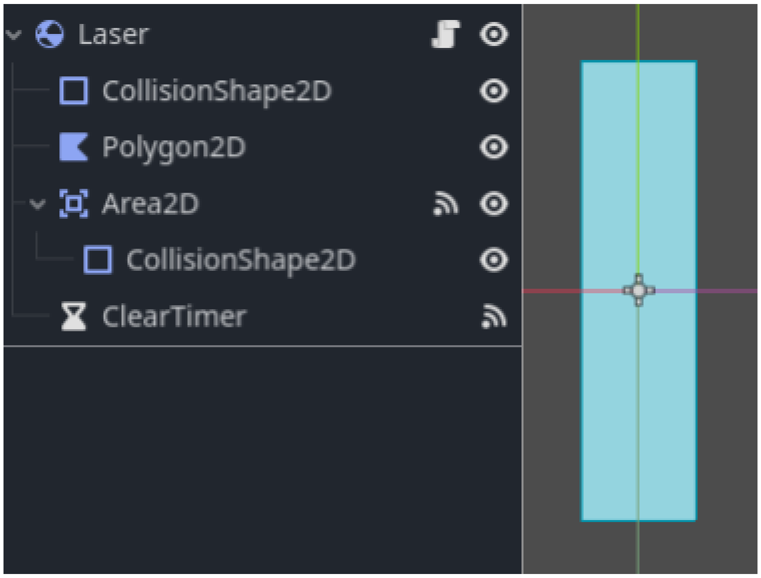


The player scene can be seen above on the right and the node hierarchy of the player scene can be seen on the left.

The script of the Player scene (see figures B2.6 to B2.8) serves many purposes, however, there are two core functionalities most relevant to playing the game. One such functionality is being able to manage the movement of the Player in accordance with user input. As established earlier, the Player has two means of movement: accelerating forward and rotating left and right. The Player can move forward with a fixed rate of acceleration and a fixed upper limit for their speed. By pressing the “W” or “UP” key the Player accelerates in the direction it is facing. The Player can rotate left and right at a fixed rate dictated by the Player script. By pressing the “A” or “LEFT” key, the Player can rotate counterclockwise and by pressing the “D” or “RIGHT” key, the Player can rotate clockwise. The second functionality of the Player script is the ability to shoot lasers. By pressing the “SPACE” key or “LEFT MOUSE BUTTON”, a laser scene is instantiated directly in front of the Player, moving in the direction that the Player is facing.

The laser scene is structured similarly to that of the Player scene with two major differences (see figure 2.1.3). First, rather than having a timer for dictating an invulnerability period, the laser has a timer node called “ClearTimer” which starts upon instantiation. When this timer ends, the scene is “freed” resulting in the laser being removed from the game. This is done to ensure that there are not too many lasers on the screen at the same time. Second, the root node of the laser scene is not a kinematic body, but rather is a rigid body instead. The reason for this is simple, as the laser scene does not require any complex manipulation to its movement behavior. The only thing the laser has to do in terms of movement is to move in a straight line. As a result, a rigid body is used in place of a character/kinematic body. The script attached to the laser scene root (see figure B2.11) is responsible for the initiation and maintenance of the laser’s movement as well as reading for collisions with asteroids in order to break them.

Figure 2.1.3: Laser Scene



The laser scene can be seen above on the right and the node hierarchy of the laser scene can be seen on the left.

The last object of note responsible for constructing the game of *Asteroids* is, of course, the asteroid. The Asteroid object scene is one that actually is far simpler in it's node structure than that of the player and laser objects. It is made with two polygons that are responsible for forming the black asteroid with a white border and one collision polygon in order for the Player and Laser objects to detect it. The Asteroid scene does not require an area node for detecting any collisions because the asteroid simply needs to exist for the player and laser objects to collide with it. Similar to that of the laser scene, the root node of the Asteroid object is a rigid body. The asteroid does not require any changes to its trajectory once it is instantiated, making this choice suitable. As for the script attached to the root of the Asteroid scene (see figures B2.9 & B2.10), the asteroid has two main functions: the initMovement() function, which is responsible for initiating movement and size and the breakAsteroid() function, which is responsible for destroying the asteroid.

In this version of asteroids, there are 5 asteroid sizes with every asteroid starting at the largest size, size 5. The size of the asteroid dictates the speed at which the asteroid moves and the scale by which the asteroid is multiplied by. The smaller the asteroid, the faster its speed and the lower its scale. When an asteroid is struck by either the player or a laser, the asteroid is destroyed by the breakAsteroid() function. While for most objects, a simple free() or queueFree() method is called to destroy the object, the asteroid object bears one unique quirk that is the ability to split. When an asteroid is instantiated by the main script (see figure B2.5), it is given an integer value between 1 and 5 that determines the number of times it will split. When an asteroid is hit, if it has splits remaining, it instantiates two new asteroids that are one size smaller and have one less split remaining. Upon instantiation, each instantiated asteroid is provided an initial speed at random according to their size and has their movement initialized by the initMovement() function. Over the duration of the game, the asteroids spawn with more splits, which consequently increases the difficulty of gameplay. Upon reaching zero splits and breaking, the asteroid is eliminated without creating any additional asteroids.

Figure 2.1.4: Asteroid Scene

The asteroid scene can be seen above on the right and the node hierarchy of the asteroid scene can be seen on the left.

2.2 Intricacies Behind the Development of Asteroids

The game *Asteroids* in its development required thoughtful consideration of the single player gameplay experience. As opposed to games such as *Pong* (see Chapter 1), in which two players play against one another with the goal of winning against the other, *Asteroids* derives a different goal—to score the highest number of points. It is a game that challenges the player to beat their own personal best and all others who have played the game, and is among some of the first games to establish the leaderboard, right after *Space Invaders* in 1978. Where *Pong* set the foundation for games with a player vs player focus, *Asteroids* further established and popularized the use of a leaderboard in which every player's score is recorded and the top scores are displayed for anyone to see.

Aside from the social and competitive introduction of the leaderboard and single player gameplay, the game of *Asteroids* contains game elements with greater complexity than its predecessors. *Asteroids* implements gameplay elements such as a free moving character, projectiles, and recursively instantiated objects. The game builds upon the core fundamentals of the gameplay loop and state management by adding greater game interactivity.

2.2.1 Game Objects

The core of *Asteroids* revolves around the interaction between the player, lasers created by the player, and asteroids. Each of these objects are designed with distinct characteristics based upon their node structures and their respective scripts.

2.2.1.1 Spaceship

The ship's movement mechanics in *Asteroids* are more nuanced compared to its predecessors like *Pong*. The player is able to rotate the ship and accelerate in the direction it is facing without any constraints to a particular axis. Furthermore, the ship is able to wrap around to the other side of the board upon crossing the threshold of the screen space, further increasing the liberty of player mobility.

It could be said that the ship in *Asteroids* is a far more involved player controlled object than that of the paddle in *Pong*. The ship in *Asteroids* is used in the same manner as the paddle is used in *Pong*, as a medium through which the player interacts with the game. With this said, the ship is directly linked to a greater scope of impact on the game. The ship is responsible for gaining points for the player, clearing asteroids in close proximity to the player, and is a risk for losing the game for the player. The ship is directly tied to the progression and ending of the game, meanwhile the paddle of *Pong* is adjacent, due to the paddle’s only job being to block and reflect the ball.

The laser object is created by the ship during gameplay due to the player’s actions and is a key feature of the game. The laser scene allows the player to break asteroids in pursuit of gaining points and preserving the life of the player. This scene adds complexity to the ship that the player controls, as it enables the player to exert their will outside of solely the ship they control.

2.2.1.2 Asteroid

In the game of *Pong*, difficulty is scaled over the course of each exchange between the players by means of the horizontal speed of the ball increasing until it crosses the threshold on either side of the screen. In the game of *Asteroids*, the difficulty is instead scaled with an increase in the number of times asteroids are able to split. At the beginning of the game asteroids are only able to split once and that is their limit. However, as the game progresses, the number of splits each asteroid is able to perform increases up to five times.

The asteroid also does something unique from a programming standpoint, as it implements the functionality of a recursive function, a function that calls itself, in the way it is designed. Each asteroid's splits variable is decremented and provided to new instantiated asteroids when it is broken. This creates a recursive creation of asteroids based around the initial amount of splits provided to the originally instantiated asteroid.

2.2.2 Object Instantiation

The game of *Asteroids* is among the first to introduce elements of object instantiation to games. In *Asteroids*, the game revolves around the instantiation of objects. This is to say that objects being created and objects being destroyed is core to the gameplay of *Asteroids* and the functioning of game objects. The ship controlled by the player necessitates the instantiation of laser scenes to interact with asteroids to gain points and the asteroids require the ability to create new asteroids upon splitting.

This concept of instantiating objects during gameplay was an important feature at the time of this game’s inception. This concept is also something that is implemented into the games created after *Asteroids* due to its addition of game complexity and flexibility. Instantiation is used in a variety of cases, from projectile creation to recursive objects.

3 *Tetris*

Tetris ("*Tetris*," 2025) is a game that can be described as a "speed-based puzzle". The game occurs on a grid pattern typically consisting of 10 squares horizontally and 20 squares vertically. The premise of the game is that you control an object made of four squares called a "tetromino". This tetromino's squares are positioned directly in-line with the grid space of the tetris board, the 10 by 20 grid behind the tetromino, and it progressively moves down the board at a set rate. The player has several means of moving the tetromino, being able to move it left, right, and even down to quicken its descent. The player also has the ability to rotate the tetromino clockwise and "drop" the tetromino in an instant bringing it to the lowest possible position in its current position and rotation. Upon reaching the bottom, the tetromino will "snap" to the grid. This snap is the act of the tetromino being removed as a player controlled object and being turned into data which is added to the board. In the place where the tetromino once was remains its squares, while a new tetromino is spawned at the top of the board. If another tetromino is to shift downward where the squares of a previous tetromino were deposited, it will display the same behavior as it would if it were to hit the bottom of the board. The player's goal in this game is to create full horizontal lines which provide the player with differing amounts of points. The player receives the fewest number of points when they complete a singular line while they receive the greatest amount of points when they complete four lines at the same time. When these lines are completed, all squares in the lines are removed and all squares above are lowered down by the number of lines cleared. As the player completes lines and scores points the level of the game increases. Based on the level of the game, the speed at which the tetromino controlled descends increases. As the game progresses, eventually the board is filled up due to the increasing difficulty of the game and mistakes made. If a new tetromino is made that cannot go anywhere, then the game ends and the player is presented with their score.

In *Tetris*, the goal of gameplay is similar to that of *Asteroids*. You score points in order to get the highest score possible. *Tetris* diverges from *Asteroids*, however, in its implementation of graphics and mechanics. Graphically, *Tetris* is a large leap from *Asteroids*, providing a variety of colors as opposed to solely black and white. *Tetris* also offers far more in terms of strategy-based compared to *Asteroids*, which emphasizes more on being able to shoot at asteroids effectively to gain points and dodge them to avoid losing lives. While *Tetris* also requires experience and skill to effectively control the active tetromino, the game has greater emphasis on planning out the next steps and adjusting a plan accordingly to changing conditions.

3.1 Developing Tetris

Compared to the previous two games developed, the game of *Tetris* separates itself in its complexity and implementation of different means of gameplay. In *Pong* the player controls a paddle and in *Asteroids*, the player controls a spaceship. These objects are represented with simple geometric shapes (a rectangle and a triangle, respectively) and share the quality of being manipulated through intervention with their velocity, setting it to a static value for the paddle and modifying it by a set acceleration and friction for the spaceship. In these cases, the physics engine built into *Godot* does a good deal of the heavy lifting by performing calculations on a basis of the amount of time passed from the last calculation performed. In the case of *Tetris*, due to the mathematical precision of the grid-based game, such calculations were not needed nor utilized. Instead, the game's "character," so to speak, is the tetromino which the player controls.

This tetromino (see figure A3.2) can be moved left, right, and down as well as rotated clockwise and “snapped” to the lowest possible position of the board. It does this not by its velocity, but rather by direct modifications to its position with restrictions to ensure it does not leave the confines of the board nor overlap with parts of the board that are filled. This is all done through communication between the main script attached to the “TetrisGame” scene and a child script attached to the “Board” scene.

The TetrisGame scene (see figure 3.1.1) holds all of the components of *Tetris* except for the tetromino object which is instantiated during gameplay. Going in order of the many components that make up the main scene, the first thing of note is the “Board” node. This object is unique to all other objects so far, as it only serves the purpose of running a script. *Godot* limits each node to only having up to one script attached, however, there are no limitations on the number of scripted objects that can be children to any given node. For the Board node in particular, its purpose is to hold the board script which contains a singular array of size 240 for storing every space on the grid as well as many methods for reading from and manipulating this array. The tetromino that moves about in the TetrisGame scene is able to interact with this array due to one very important utility function in the main script called “convertTetrominoToArray()”. This function is responsible for finding the active tetromino by name and converting its positional data by pixel coordinates directly into indices of a one dimensional array of size 240. These coordinates are then compared to those of the board array to check if a desired operation can be fulfilled. If the operation (e.g. slide to the side, rotate, shift down, etc.) can be fulfilled, then the tetromino performs the operation accordingly. Otherwise, the operation will not be performed.

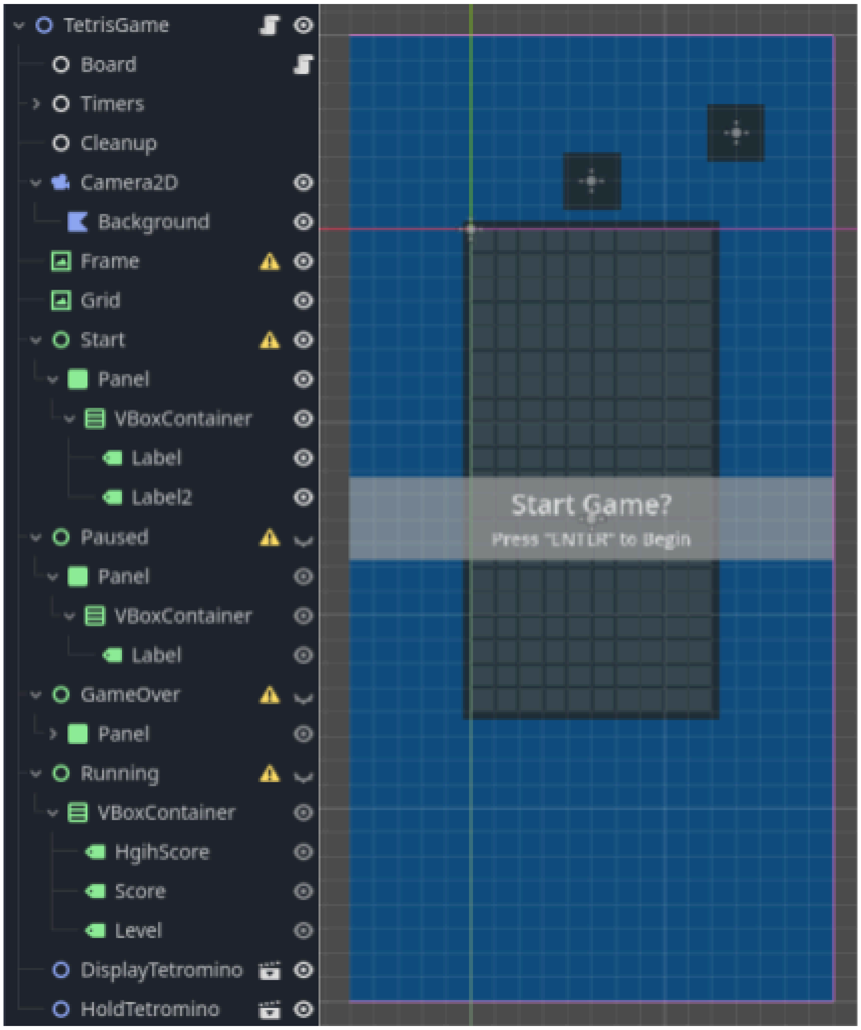
After the board node, there are three timers that are primarily used to influence gameplay: the “MovementTimer”, the “CleanupTimer”, and the “SlideTimer” (see figure B3.10). First, the MovementTimer is the timer responsible for shifting the tetromino down over the course of the game. The more points the player accrues, the faster this timer fires, resulting in the tetromino moving faster. Second, the CleanupTimer is responsible for dictating when child objects of the cleanup node are removed from the scene. When an object is no longer needed and has to be removed, it is parented by the node titled “Cleanup” to be removed when the cleanup timer fires. The primary purpose of this timer is to increase the efficiency of the cleanup process and to ensure no extra objects linger around. Lastly, the SlideTimer is primarily used as a convenience feature which cycles when the “LEFT/A” or “RIGHT/D” keys are pressed to move the tetromino left or right. This allows for smooth incremental movement when the keys are held instead of simply being pressed.

The following nodes are all present for the UI and background of the game. The background, frame, and grid persist for all states of the game, while the “Start”, “Paused”, “GameOver”, and “Running” scenes have their visibility altered on the basis of the state of the game. Following these are two scripted nodes which are responsible for two of the core mechanics of the game: the “hold” and “display” mechanics. The “HoldTetromino” object is in the top right corner of the scene and acts as a place for the player to hold precisely one tetromino at a time. If a player does not want to use the tetromino they have been given or want to use it later, they may press the “E” or “H” key to move it to the top right corner on top of the hold tetromino space. If there is no tetromino in that space, then a new tetromino is instantiated; if there is a tetromino in that space, then the two are swapped. The “DisplayTetromino” object can be found directly above the grid and is responsible for showing the next tetromino to be created.

When the active tetromino is moved to an empty hold space or locked to the grid, the tetromino seen in the display space becomes the new active tetromino.

The components that have been discussed so far make up the node architecture for the game of *Tetris*. However, there is a great deal that goes into the main script attached to the root of the TetrisGame scene (see Appendix B3). This version of *Tetris* has three distinct states programmed and four "visible" states. More specifically, one of the three states implemented into the game has two different modes. The three states used in this game are "Start", "GameOver", and "Running" (see figures A3.1, A3.5, A3.6, & A3.8). The Start and GameOver states exist only to preserve a clean transition from one state of the game to another. In the Start state, the player is presented with the start screen and only allowed to initiate gameplay by entering the Running state. In the GameOver state, the player is instead presented with the game over screen and is only allowed to return to the Start state. The Running state is where the game really begins, as the first tetromino is instantiated and begins to fall. In this state, the player is able to move the tetromino around and, provided the correct moves, the player can clear rows of squares to gain points. Within the Running state, the player is able to press the "P" key to pause the game. When the game is paused, a simple UI element is made visible and all action of the scene ceases. In this state, the tetromino cannot move by any action of the player or by any action of the movement timer. To unpause the game, the player can simply press the "P" key again and return to playing.

Figure 3.1.1: TetrisGame Scene

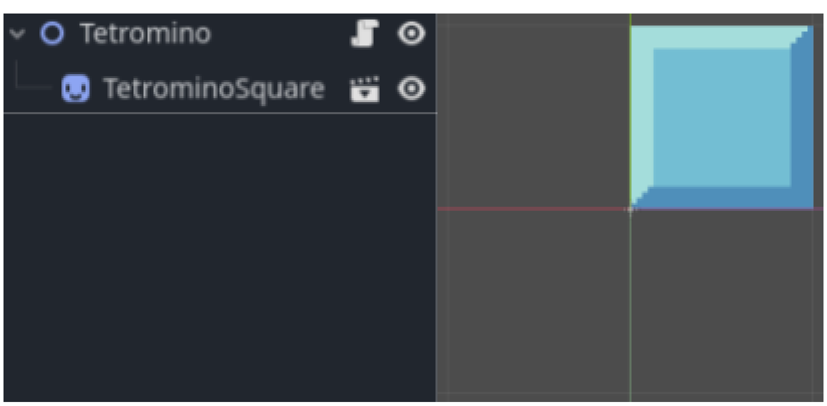


The tetris game scene can be seen above on the right and the node hierarchy of the tetris game scene can be seen on the left.

While playing the game, multiple operations occur in each frame, many of which are centered around the Tetromino object. These operations all occur within the main script (see figure B3.4 to B3.10), dictating the movement of the tetromino, the creation of a new tetromino, the conversion of the tetromino into an array, and the addition of a tetromino to the display and hold spaces. The Tetromino object itself (see figure 3.1.2) does not have much content in terms of its construction and programming. The node architecture is simply composed of a singular 2D sprite titled “TetrominoSquare” and a 2D root node with the tetromino script attached. This is the first usage of a sprite thus far, that being a 2D image used to represent an object or environment. The reason for the way the tetromino is structured lies in how the tetromino is coded (see figures B3.14 & B3.15). Every tetromino, by their nature, must be constructed with four squares. The square that each tetromino begins with is the square at the origin point of the object and is mathematically represented as “Vector2(0,0)” or “Vector2.ZERO”. The tetromino script is responsible for instantiating three additional squares to complete the tetromino. It does this by using a dictionary called “cells” in the “Globals” script.

Globals is a script that serves the sole purpose of storing important values and has no code to directly run. Among these values is the list “cells” which contains each value of the tetromino enumeration as a key. This key is attached to a list of four vectors that designate the positions of each square in the tetromino. This relative positioning is in units of a full square width, consequently each square’s position is multiplied in order to be placed at the appropriate location in units of pixels. After all of the four squares have been created, their color is set by changing the frame of the spritesheet they inherit their texture from.

Figure 3.1.2: Tetromino Scene



The tetromino scene can be seen above on the right and the node hierarchy of the tetromino scene can be seen on the left.

3.2 Intricacies Behind the Development of Tetris

The development of *Tetris* introduces a unique set of challenges that are distinct from previous games such as *Pong* and *Asteroids*. Unlike the focus of prior games being focused around vector-centered game objects, as seen with the ball of *Pong* and the laster/asteroids of *Asteroids*, *Tetris* is a puzzle game that deals with the management of player controlled pieces in a grid-based environment. While *Pong* and *Asteroids* work with setting the speed of objects and the direction they are moving, *Tetris* works with the timing and validation of movements of a tetromino while considering squares fixed to a limited board space. The game is centered around an active tetromino being controlled by the player as their means of interacting with the game and progresses with their completion of full rows.

3.2.1 Grid-Based Movement and Tetromino Manipulation

At the heart of *Tetris* is its grid system, a fixed board where tetrominoes fall and interact with previously placed tetrominoes. The grid occupies 10 spaces of width and 20 spaces of height, where the grid is initiated empty and is filled as tetrominoes are placed. Each tetromino is made of four squares, and each square has a position within the grid. The tetromino is represented as a set of these squares, which are initialized in one of seven predetermined shapes (I, J, L, O, S, T, and Z). Unlike vector-based movement where speed and direction are the values of interest, *Tetris* operates in the movement of a tetromino to the sides, downwards, and rotated clockwise.

Provided that there is grid space available for all of the squares of a tetromino to occupy, the tetromino can move all of its squares horizontally by one space with player input. If there is no grid space available, the tetromino remains stationary. Upon being shifted down, the same condition is checked as is done for horizontal movement. If the tetromino passes the check, the tetromino is shifted down. If the tetromino encounters conflicts with the grid, it is then attached to the grid and the next tetromino is instantiated. The remaining movement the tetromino can perform is a clockwise rotation, which has slightly more nuance than previous movement validations. If the rotation would move the tetromino outside of the grid space, the tetromino is moved horizontally to keep it within the border of the grid. Additionally, if a rotation would move a square of the tetromino into an occupied part of the grid, the tetromino is then unable to perform any rotation.

These movements are implemented through direct changes to the tetromino's position on the grid. This system requires precise manipulation of the falling tetrominoes and careful planning of where to place tetrominos to maximize the number of points gained through gameplay.

3.2.2 Board Management

In *Tetris*, it is vital to keep track of the state of each square in the grid. To do this, a one-dimensional array of size 240 is used, where each index corresponds to a specific grid square. This array is manipulated to track which spaces are filled and which type of square fills each grid space. When any movement operation is attempted to be performed on the active tetromino, its pixel-based coordinates are converted into array indices of the grid array corresponding to the current position of the four squares of the tetromino. If there is an overlap with existing squares on the board or a boundary breach, the movement operation is prevented.

When the active tetromino fully descends down the board to a valid location for it to become part of the grid, several important steps take place. The indices which the tetromino occupies in the grid array are filled with a character to represent the type of tetromino being attached to the board (‘i’, ‘j’, ‘l’, ‘o’, ‘s’, ‘t’, or ‘z’). These values added to the array are used to determine whether those spaces are filled and with what color of square they should be filled with. Based on which spaces of the board are containing squares or missing squares dissonant with the array due to the addition of a tetromino or the clearing of a row, squares are added to the board, removed from the board, or have their positions changed.

3.2.3 Timers and Speed Management

While there is no “speed” that the tetromino is moving at, there is a rate at which the tetromino descends by one space at a time. The primary factor influencing the increase in difficulty of gameplay for the player is the time it takes for the tetromino to descend. The tetromino descends by one space at a set interval of time designated by the MovementTimer. As the player accumulates points and the game increases in level, the time designated by the MovementTimer decreases, therefore increasing the speed of the tetromino’s descent. The faster the tetromino becomes, the harder it can be to control where it falls and plan where it could best be placed.

In *Asteroids*, timers were utilized to clear the lasers created by the player as well as manage the invulnerability period of the player. With this said, the usage of timers for the game of *Tetris* is integral to the core mechanics of the game. Additionally, another timer used for the purpose of more engaging gameplay is the SlideTimer. This timer has the expressed purpose of providing the player with the ability to smoothly move the tetromino to the sides and downwards by holding down the corresponding keys rather than tapping the keys for each increment of movement.

3.2.4 Informational Mechanics

There are two mechanics introduced in *Tetris* that contribute greatly to strategy building during gameplay: the hold and display mechanics. Above the *Tetris* board are two spaces designated to hold and display tetromino. The held tetromino is a tetromino that is being held by the player to be used at a later time while the displayed tetromino is the next tetromino to descend. These provide vital information to the player and the opportunity to control more of the game then if tetrominoes were to descend at complete random with no information regarding which tetromino is going to descend.

Information is valuable when playing any strategy-based game and contributes to more entertaining gameplay by giving the player the ability to think their way through the game provided their limited resources. The player has to make thoughtful decisions about their moves as they influence future gameplay, with a mistake being able to come back at a later time to end a run of the game early.

4 Platformer

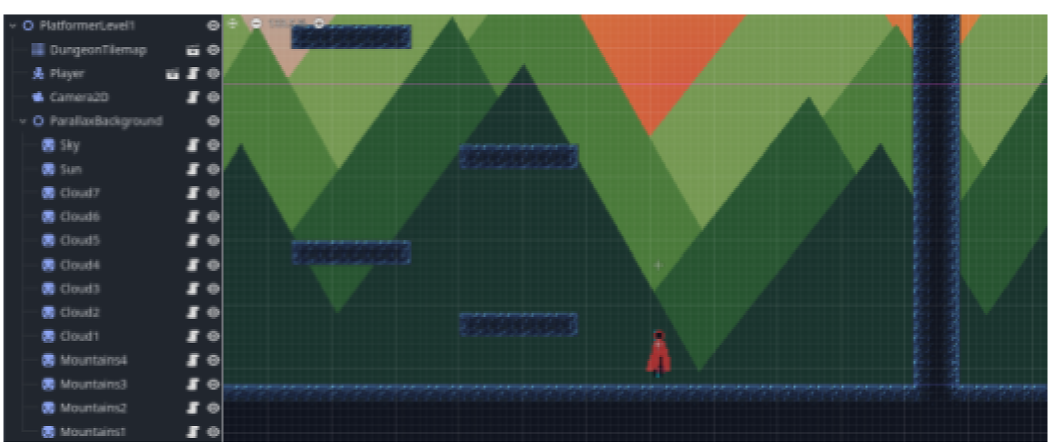
A platformer (“Platformer,” 2025) is a game designed around the set purpose of getting from one place to another. In a typical 2D platformer, the player typically comes in the form of a 2D animated sprite which the viewport of the game follows as they move about the screen. The most conventional controls for this type of game include the player’s ability to move the player character left and right, as well as the ability to make the character jump. This allows the player character to move from platform to platform to get to their desired goal, hence the name “platformer”. Some games provide additional capabilities such as limited flight or even restrict some movement such as limiting the player to one direction of movement or to only being able to jump (e.g. *Jump King*; *Jump King*, n.d.). This formula also extends into 3D games, with some modifications due to the addition of navigating a 3 dimensional space. By convention, in a 3D platformer the player has the ability to move forwards, backwards, and side-to-side. The camera also follows a few standard orientations, viewing the character from the back or side with a fixed or free camera, or viewing the world from a first-person perspective in the player character's viewpoint.

The direction of platformer games can vary greatly, with some games emphasizing the completion of puzzles (e.g. Portal 2; *Official Portal 2 Website*, n.d.) and others emphasizing platforming challenges of increasing difficulty (e.g. Celeste; *Celeste*, n.d.). What platformers provide in spades compared to the games created prior (*Pong*, *Asteroids*, and *Tetris*), is the freedom to traverse the game environment. In a game like *Pong* and *Tetris*, the game is centered around a fixed viewpoint of the pong board and tetris grid respectively. Meanwhile, in a platformer, the game’s viewpoint is fixed upon the player character and follows them wherever they go. This freedom allows for a vast array of options through which gameplay can be constructed, leading the genre to branch into many different routes. The intent and purpose behind these games at their core is traversal, moving to a designated location with purpose.

4.1 Developing a Platformer

In developing a platformer, two primary requirements must be met. A player character that can move about its environment and an environment for the player character to move around in. In this particular platformer prototype, three primary objects are created which include a tilemap to create platforms and walls for the player character to make contact with, a parallax background, and the player character itself (see figures 4.1.1 to 4.1.3). Unlike the previous games developed so far, platformers often necessitate the development of multiple scenes to switch to on the basis of progression. Consequently, there is far more emphasis on modular objects than before, so much so that the "PlatformerLevel1" scene doesn't need to have a script attached to it (see figure 4.1.1). This scene contains multiple elements which function independent of their parent scene, making them capable of being used to construct multiple levels of different designs and difficulties.

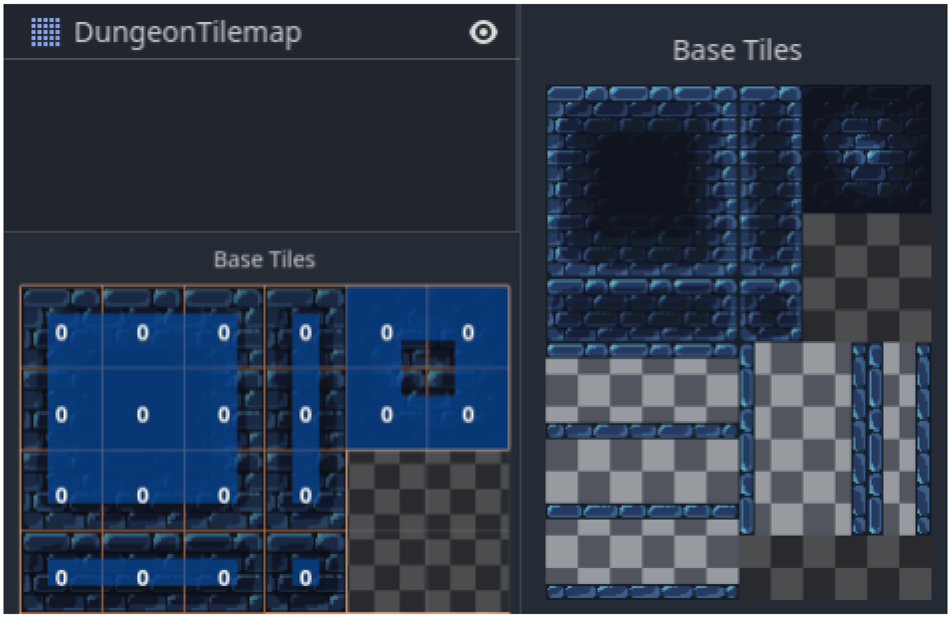
Figure 4.1.1: Platformer Level Scene



The platformer level scene can be seen above on the right and the node hierarchy of the platformer level scene can be seen on the left.

The first object in the platformer scene is the “DungeonTilemap” node, which is the tileset themed with blue tiles (see figure 4.1.2). This tilemap combines three primary features: the ability to place a tileset onto a 2D environment, add collision to a tileset, and automatically configure tiles drawn onto a 2D environment. The first feature allows the developer to draw a tileset into a 2D environment. A tileset is a collection of drawings with even spacing that allows them to be drawn recursively into an environment. The tilemap sections off each individual drawing of a tileset for this expressed purpose. The second feature allows the tileset to interact with other physics bodies, allowing the player to stand on platforms and collide with walls. The third feature allows the developer to “autotile”, an incredibly resourceful tool when creating levels. To begin autotiling, each tile is split into a 3 by 3 grid which can be filled according to where each tile connects to adjacent tiles (see figure 4.1.2). If this is done for a complete tileset that can represent all configurations within a 2D space, the tiles can simply be drawn into the environment in which they are desired while *Godot* is able to handle which tiles are most appropriate to use and fills in the spaces drawn in using the tileset. In the creation of a 2D platformer, tilemaps are typically some of the most versatile tools for level building.

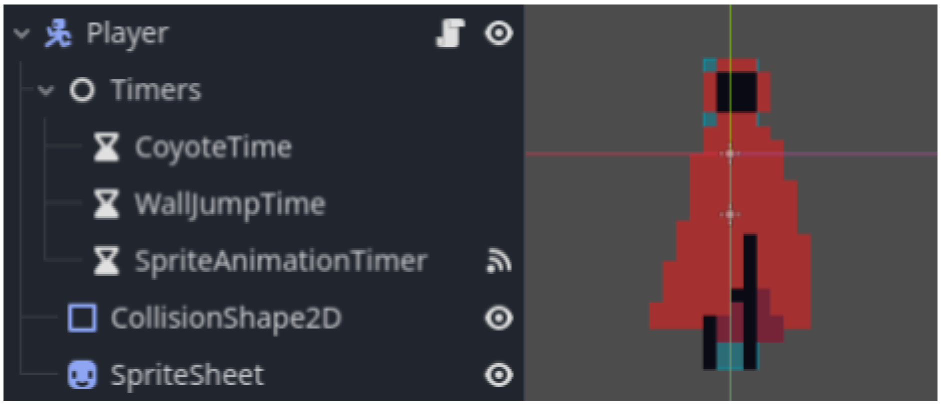
Figure 4.1.2: Tilemap Scene



The tilemap scene can be seen above on the right and bottom left and the node hierarchy of the tilemap scene can be seen on the left.

After the tilemap node, there are three more objects of note that compose the platformer: the Player object, the Camera object, and the Parallax Background object. The Player object (see figure 4.1.3) is composed of a set of three timers, a collider, and a sprite sheet. The timers “CoyoteTime”, “WallJumpTime”, and “SpriteAnimationTimer” all have expressed purposes based on how the player’s behavior is programmed (see figures B4.3 & B4.4). CoyoteTime is responsible for giving the player a brief period of time in which they can jump after leaving a ledge. This is implemented for the sake of providing forgiveness for small and common mistakes by the player when jumping off of a ledge. This same principle carries over into WallJumpTime which creates a refractory period for when the player leaves a wall. The last timer, SpriteAnimationTime, is a utility timer with the expressed purpose of creating fluid animation. The player has multiple states that dictate which frame of the sprite sheet (see figure A4.4) is used at a given moment. Two of these states, “walk” and “idle”, use multiple frames. While in these states, SpriteAnimationTime is responsible for cycling between the appropriate sprite frames for these animations.

Figure 4.1.3: Player Scene



The player scene can be seen above on the right and the node hierarchy of the player scene can be seen on the left.

As would be expected, the root node of the player scene is a character body similar to that of the paddle of *Pong*, the spaceship of *Asteroids*, or the tetromino of *Tetris*. Compared to those characters, though, the player character of this platformer has the most complex movement and animation behavior seen thus far in the analysis (see figures B4.1 to B4.4). Nearly all processes of the player script occur each frame, as they are run in _physics_process(delta). Each frame, the player's movement, friction, jump, timers, and animation are handled in a sequential fashion (see figure B4.1). In terms of movement, the player character has access to several means of platforming. The player has the base capabilities of being able to move left and right using the "A/LEFT" and "D/RIGHT" keys respectively and the ability to jump using the "SPACE" key. These allow for the simple traversal of platforms. Alongside these capabilities, the player may additionally jump a second time by pressing the "SPACE" key while mid-air and wall jump by pressing the "SPACE" key while in contact with a wall. Beyond these actions the player may also slow their descent and place themself into the “float” state by holding the “W/UP” key while falling.

Throughout the management processes of the players’ movement, the player’s state is changed according to user input and the player character’s position relative to the world surrounding it. The player’s state is then accounted for during the animation step, which manipulates the graphics of the player to match their current action (e.g. walking, jumping, standing, etc.).

4.2 Intricacies Behind the Development of a 2D Platformer

Developing a 2D platformer is a process that involves the management of physics processes, animation handling, and level designing. In a platformer game, the purpose of all objects is to exist for the player to interact with and observe as a means of gameplay. The level is designed for the player to traverse, meanwhile collectibles are designed for the player to acquire, enemies are designed to challenge the player, and goals are designed to give direction to the player’s traversal of the game. In creating a 2D platformer, the player character lies at the core of development, as it is through the character that the player is able to experience the game.

4.2.1 Player Movement

Out of the games seen thus far in this piece of work (*Pong*, *Asteroids*, *Tetris*), the 2D platformer provides the most fast array of options for the player to interact with the game. The player can walk, jump, double jump, and wall jump to traverse all parts of the game, and provided the camera that follows the player around the scene, levels can be constructed to provide any amount of the world for the player to traverse. All interactions with the game occur through the player's movement. As the player character navigates the scene, the player is able to observe more of the game world and make decisions on where to go accordingly.

4.2.1.1 Jump Mechanics

Platformers necessitate one particular quality to be a platforming gaming, which is the ability to jump. While this may seem like a trivial mechanic in which the player must be launched upwards before being brought back to the ground, there is a great deal of thought that goes into this process. The jump designed for a platformer character can vary, some games implement a custom jump that manipulates the velocity progressively over the course of the jump. In this prototype, all jumps (jump, wall jump, and double jump) are implemented through simply setting the vertical velocity of the player to the set value. Upon double jumping, this value is slightly minimized and upon wall jumping, some horizontal velocity is applied to the player to move them away from the wall. These create a varied array of options for the player to traverse the world. The player is then brought back to the ground by applying gravity to the player while they are in the air.

4.2.1.2 Flexible Movement Mechanics

Something that maximizes the quality of controlling the player character's movement is the usage of two timer's to support movement: the CoyoteTime timer and the WallJumpTime timer. The CoyoteTime timer is implemented to create forgiveness for the player making small mistakes and accounting for cases in which the player performs a jump at the very edge of a ledge, but the game fails to identify that the player is still in contact with the floor. People are not perfect, consequently mistakes in which the player just barely misses a ledge is inevitable. Such moments are unsatisfying during gameplay, resulting in the implementation of the leeway that CoyoteTime provides.

This same thing is implemented using WallJumpTime for the player's wall jump. This implementation also varies from the CoyoteTime slightly in its purpose. While both timers work to reduce the dissonance between the actions of the player and the responsiveness of the player character, the WallJumpTime timer is integral to maximizing the functionality of the wall jump as a whole. When jumping from a wall, based on the construction of the environment around the player, the player might encounter a platform or another wall that they need to access. This would naturally necessitate the player to move towards their goal directly after jumping from the wall. If the player attempts to move away from the wall just before the player performs a wall jump, the WallJumpTime will be able to catch the player's attempt at a wall jump and allow the action despite the player not being in contact with the wall.

These timers are so brief, that they often never even go noticed, yet they provide an overall improvement to the feel of gameplay for the player. Work like this often goes unnoticed in many games, but it does not take away from the value they provide.

4.2.2 Animation Systems and State Management

While handling every input event from the player, the state of the player is changed. This same process occurs even when there is no player input on the basis of the objects the player is in contact with and the speed at which the player is moving. The state of the player reflects all of the various conditions that surround the player. For example, the player is walking if the player is moving to the side and they are in contact with the floor. Alternatively, if the player is moving upwards and in the air, they are jumping. Provided that the player is in the air, moving downwards, and the "W/UP" key is pressed, they are floating. These states of the player are kept track in order to appropriately animate the player in accordance with their actions.

During the animation step, the player is provided one of five animations congruent with their state. Three of these animations are static while two cycle between multiple frames. To account for this limited animation cycle, while the player character has a single frame animation active, their sprite is simply set to that state. During the jump state, the jump frame is set and during the float state, the float frame is set. Alternatively, during the multi-frame animations, the SpriteAnimationTimer is started and the player character's animation is initialized at its first frame. Every time the timer completes a cycle, the frame increments by one. Provided overflow past the frames available for the animation, the frame is set back to the initial frame of the animation.

4.2.3 Level Creation

Levels in a 2D platformer constitute the world through which the player travels. Through each level the player might have an objective to fulfill, quota to meet, or a destination to arrive at. There are two particular features of a level to give depth to the scene and provide traversability to the scene: the background and the foreground.

The background that was used in this platformer prototype is a parallaxing background. It adjusts the position of individual background elements relative to the player's movement in order to give depth to the scene despite its 2D nature. These background elements exist solely to add aesthetic value and depth to the scene. What is needed to create parts of the level which the player can interact with directly is a foreground. In this game prototype, a tilemap is used to place squares of varying text into the foreground. These tiles are provided the capacity to interact with the player through the addition of colliders to each tile, giving the player the ability to walk along them and use them to traverse the scene.

Functionally anything can be used to represent the level, even simple shapes and colors. With this said, implementing mechanics such as parallaxing and tile mapping can create games with far greater depth and with larger level scope.

5 First Person Shooter

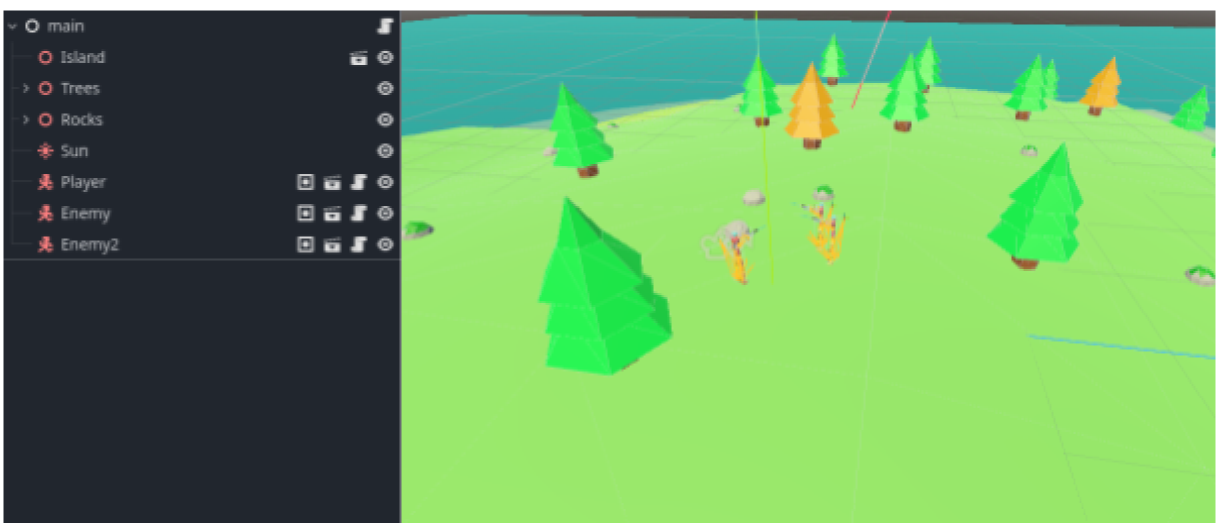
A first person shooter (FPS) is a type of game developed in a 3D or 3D-like environment, centered around the mechanics of a free moving character body and shooting projectiles in some manner. Such games have a vast variety of objectives and goals, some having the set purpose of fighting computer controlled enemies, while others have player characters which you fight against ("First-Person Shooter," 2025). As elaborated briefly in regards to a platformer game in a 3D environment, an FPS typically gives the player the liberty of moving forward, backward, left, and right. Often, an FPS also will integrate the ability to jump, crouch, and sprint. Some even go a step further by giving the player unique mobility options such as diving and crawling. Beyond such options of mobility, the core mechanics of shooting often comes from items which the player equips, often being in the form of guns. All FPS games, by their nature, give the user a view of gameplay from the first person, but not all of such shooters are limited to one such perspective. Some FPS games allow for the player to view gameplay from different perspectives such as the third person view (e.g. Fortnite, Valorant; *Fortnite*, n.d.; *VALORANT*, 2025).

For an FPS, the range of games that exist has the very same variety as that of platformers. In the very same way as a platformer, an FPS follows the player, giving liberty to the player to navigate their environment as they please. Some of the first games of this genre provide the player with a maze to navigate through with enemies to defeat along the way (e.g. Doom; "*Doom* (1993 Video Game)," 2025). Some of the newer games take a different approach, producing gameplay in which the players are pitted against each other in battle (e.g. Fortnite, Valorant; *Fortnite*, n.d.; *VALORANT*, 2025). These games are constructed around the premise of defeating all opponents, whether that is by yourself or on a team, facing computer controlled characters or other players.

5.1 Developing a First Person Shooter

The FPS game is the only project in this paper that utilizes 3D nodes. These 3D nodes occupy the same use cases as 2D nodes, but instead bear characteristics of a 3D object and exist in a 3D space. The vast majority of 2D nodes have direct counterparts in 3D to account for a third dimension to methods and data values. For example, the player character for this project (see figure 5.1.3) has a root node "CharacterBody3D" which is the 3D counterpart to "CharacterBody2D". Instead of position, velocity, and acceleration data being stored in a Vector2, a vector with x and y values, such data is stored in a Vector3, a vector with x, y, and z values. In this particular version of an FPS, the elements in the scene are kept relatively minimal, similar to the development of the platformer prototype.

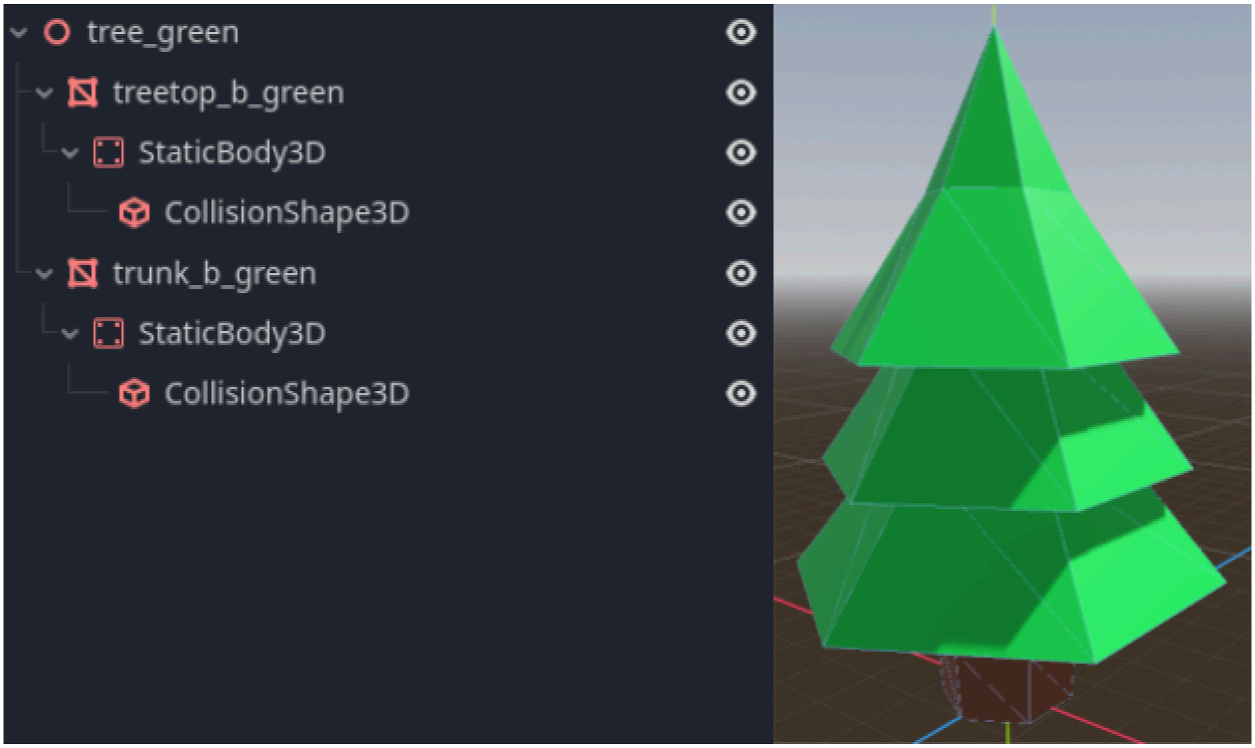
Figure 5.1.1: Main Scene



The main scene can be seen above on the right and the node hierarchy of the main scene can be seen on the left.

In the test level scene (see figure 5.1.1) there are different objects that are responsible for the composition of the environment. The island acts as a base or floor, on top of which a selection of rocks and trees populate. This island as well as the rocks and trees are all constituted by a mixture of two primary nodes: A “MeshInstance3D” and a “StaticBody3D” (see figure 5.1.2). The mesh instance for these objects is a 2D sprite or color array resource, this resource being an array of color values and points of positional data to represent the object in a 3D space. The “mesh” terminology used here refers to any object presented in a 3D environment. The static body allows these objects to interact with objects also containing physics bodies such as the player. For these objects, the static body shares the same positional data points as each object’s mesh instance. One object that has not been seen until now is the node titled “Sun” which is a “DirectionalLight3D” node. The purpose of this node is to illuminate the scene from a single point, making it crucial for a 3D environment in which shadows are vital for representing depth graphically. This node has a 2D counterpart that has not been used so far due to its purpose being more out of aesthetics rather than need. None of these objects are scripted, as the sole purpose they need to fulfill is to take part in helping to compose the world environment.

Figure 5.1.2: Tree Scene

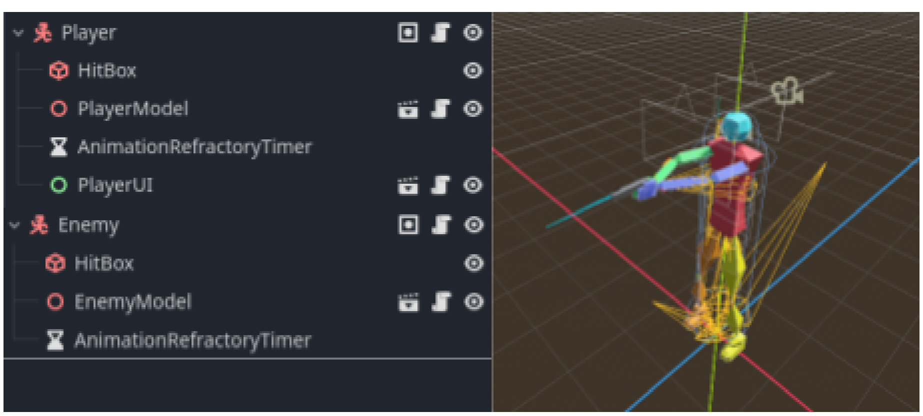


The tree scene can be seen above on the right and the node hierarchy of the tree scene can be seen on the left.

What remains in this game's level design is a player and two enemies (see figure 5.1.3). The Player and Enemy objects are formed of nearly an identical node structure with two notable exceptions. The first being the presence of a UI for the player, given that they need for a graphical representation of their data and the enemy does not. The second exception of note is that the Player object has two different cameras, one for their first person perspective and another for their third person perspective (see figure 5.1.4). The first person camera is simply attached to the head of the character (see figure 5.1.5), while the third person camera is more complex in its construction. It is built around a set of two nodes called "Path3D" and "PathFollow3D" which allow the camera to move fluidly between two points on a set path. The purpose of this is to allow the third person camera to adjust its position based on the presence of objects that enter the range of the camera. The camera detects such objects through the usage of two raycasts, which work to detect objects in a straight line in front of and behind the camera's view. This ensures that this camera does not get stuck in objects and provides that player with a consistent third person view of the game.

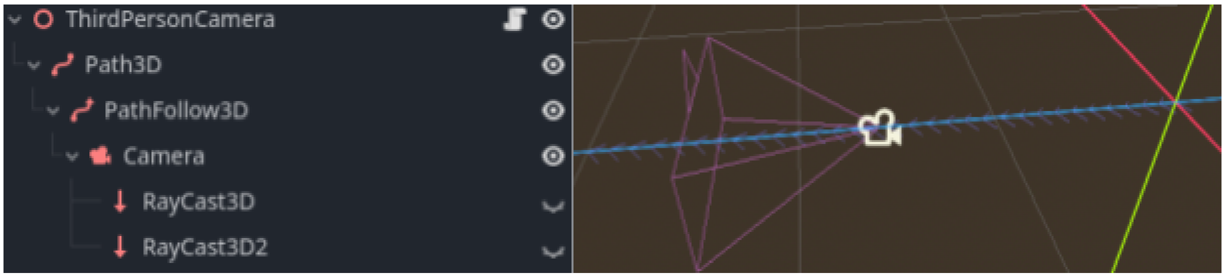
What does remain consistent between the player scene and the enemy scene are the hitboxes used and the model used. Both scenes (see figure 5.1.4) utilize what is called a capsule shape for their collision detection. While in previous 2D projects a simple square or circles sufficed for all intents and purposes, 3D shapes need to account for there being three dimensionality to their influence on the environment. The capsule shape in particular is a common option for characters in 3D, as it has a rounded top and bottom which makes it harder to get stuck on objects, while also having a cylindrical body that maintains other objects at an exact distance from the character in all directions. Aside from the capsule collider, another thing the player and the enemy both share is a character model, which bears significantly more complexity in its design..

Figure 5.1.3: Player Scene



The player scene can be seen above on the right and the node hierarchy of the player scene can be seen on the left.

Figure 5.1.4: Third Person Camera Scene



The third person camera scene can be seen above on the right and the node hierarchy of the third person camera scene can be seen on the left.

This character model (see figure 5.1.5) is a demonstration of the interaction between 3D modeling software and its compatibility with the *Godot* engine. The model itself is imported from Blender (Blender.org - Home of the Blender Project, n.d.), a community-developed and free to use software used for the development of 3D models. The model here is composed of precisely 14 different meshes. Among these meshes are two feet, two hands, one head, two lower arms, two lower legs, one torso, two upper arms, and two upper legs. All of these body parts, as themselves, would not be able to properly constitute a player model, as there is nothing that allows for the model to move or be animated in any fashion. In theory, each body part can be moved and rotated on an individual basis in order to create the appropriate animations for the character such as walking and jumping. However, such methods are highly inefficient due to the amount of time it would take to produce animations. The solution to this problem is to create a skeleton for the model. A skeleton is an invisible collection of objects called "bones" which are responsible for manipulating 3D models based upon their position, rotation, and size. The skeleton of a 3D model is similar in purpose to that of the skeleton of a person: it is a non-visible part of the body that constrains movement by the nature of its construction. For a 3D model, a skeleton is responsible for connecting the various individual parts together in order to create one cohesive object.

The primary means by which the skeleton works to tie the different parts of the model together is through hierarchies, using the very same parent-child relationships as the *Godot* engine. Conventionally, the parent's behavior directly influences the child's behavior. This is the default behavior of the hierarchy in Blender, with positional and rotational changes made to a bone influencing its children. Such behavior is referred to as "forward kinematics", as all changes are applied forward in the hierarchy. However, such methods are not ideal when animating certain movements, such as movements of the hand which often necessitate the hand moving to set positions. Such movements would be achieved easiest through the usage of another form of kinematics, "inverse kinematics". This technique is applied to the hands and arms of the model and results in changes to the child bone, in this case the hand, affecting its parent nodes, the lower arm and upper arm. While this method reverses the way in which the bones interact with one another, it still abides by and utilizes the convention of the parent-child relationships present.

Outside of the usages of the hierarchical nature of the skeleton, one more method was used to improve the animation process, which is the usage of "targets". A target is a bone that is removed from the hierarchy and serves mainly the purpose of acting as a pointer for other bones to aim towards. The target bone is a child of only the main bone of the model, meaning it does not follow the convention of being a child of another bone of the character's body parts. For this model, there are four pointers, two for the arms and two for the legs, which serve the purpose of keeping the elbows of the arms back and the knees of the legs forward.

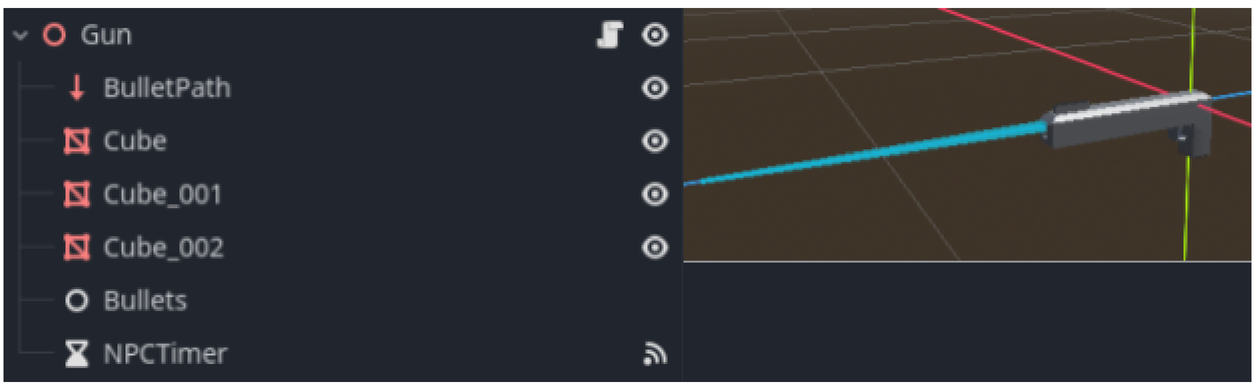
In the character model (see figure 5.1.5), each of the 14 meshes has a corresponding "BoneAttatchment3D" node. This particular type of node allows for additional items to be added to the skeleton. Every bone, at minimum, has a static body attached to it with the same positional data as the mesh corresponding to that particular bone. The only exception to this is the torso which is split between three bones: the hip, the chest, and the spine, which are all responsible for manipulating the torso mesh and are all given their own cuboid shape that follows the dimensions of the torso. Despite not being exactly one-to-one with the model, the effect is negligible during gameplay. Beyond the static bodies which are added to the player model, there are a few more notable attachments: attached to the head is a first person camera, to the main bone a third person camera, and to the right hand a gun.

Figure 5.1.5: Player Model Scene

The player model scene can be seen above on the top right and the node hierarchy of the player model scene can be seen on the left across two columns.

For the purposes of this prototype, a simple gun (see figure 5.1.6) and bullet (see figure 5.7) are used. The gun scene is of a relatively simple construction designed around the set purpose of instantiating bullet scenes (see figure B5.12). The first node in this scene is a raycast, a 3D vector with a set length, titled “BulletPath”, which is responsible for giving bullets a point to start from and a set direction to move towards. After the raycast is a set of three meshes that constitute the gun object. The scene also contains a node to contain instantiated bullet scenes and a timer that is implemented for the purpose of demonstrating simple enemy behavior, as bullets are produced by timeouts of the timer rather than player input. As far as player input is relevant to the gun scene, upon pressing "LEFT MOUSE BUTTON" a new bullet is created and is informed by the raycast of where to be positioned and at what angle. After that, all behavior of the bullet is carried out by the bullet's script.

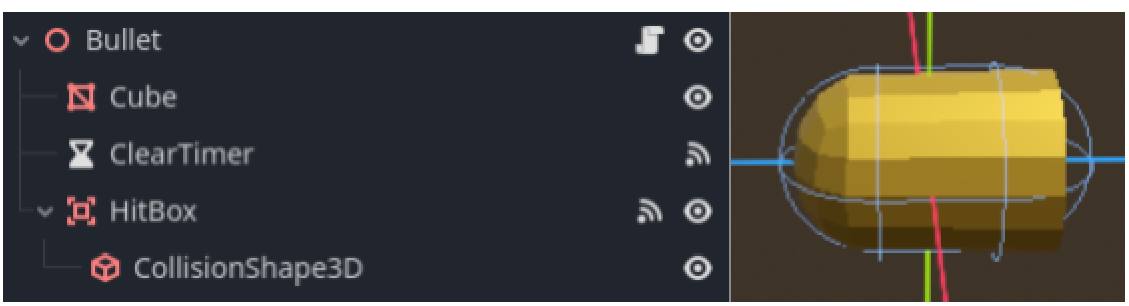
Figure 5.1.6: Gun Scene



The gun scene can be seen above on the right and the node hierarchy of the gun scene can be seen on the left.

The bullet scene (see figure 5.1.7) is also light in construction, possessing a single mesh for its body, one timer to free the bullet after a set amount of time, and a simple hitbox for detecting objects. What makes the gun scene different from projectile objects seen thus far, such as the ball in *Pong* or the laser in *Asteroids*, is that it is not a physical object nor does it use the physics engine. This is made apparent first by the fact that the root node of the bullet scene is a simple 3D node, but also in the script of the bullet scene (see figure B5.13). The bullet script has a few primary purposes: move the bullet, emit damage signals, and destroy the bullet. The bullet object is moved by multiplying the speed of the bullet by the amount of time passed and adding that value to the position of the bullet in the direction it is moving. This is the reason while this bullet is so different from previous projectiles, as instead of manipulating the velocity of the projectile, the speed of the bullet is directly applied to the position of the bullet over time (therefore achieving the same effect without work of the game engine). The bullet runs a function upon colliding with an object, which checks for the layer the colliding object is on and emits a signal of varying damage on the basis of which layer the object is on. This is done so that different parts of a character, player or enemy, take damage according to the severity of a hit. By the logic of the bullet script and the player scene, the player's head receives the largest amount of damage and the arms and legs receive the least amount of damage, with the torso receiving the average of the two. Upon colliding with the object, the bullet also is destroyed, irrelevant of what object it collides with. If the bullet does not collide with anything within 10 seconds it will also be destroyed from a timeout call from the "ClearTimer" node.

Figure 5.1.7: Bullet Scene



The bullet scene can be seen above on the right and the node hierarchy of the bullet scene can be seen on the left.

5.2 Intricacies Behind the Development of a First Person Shooter

Developing an FPS involves a similar level of focus to the player character as is required for developing a platformer game. At the center of gameplay is the player and their capacity to interact with and influence the environment. In this FPS prototype, a 3D environment and 3D player character are created, with the player character having the capacity to walk and jump around their environment much like that of a platformer character. The FPS character, however, separates itself from a standard platformer character greatly in the focus of its construction. The place where gameplay has its greatest focus is the implementation of mechanics to shoot projectiles. Additionally, actions are implemented for the purpose of preserving the player's capacity to not be hit or increase their capacity to hit targets of their own. The ability to sprint, crouch, and view the player character from the third person perspective revolve around this virtue.

5.2.1 Player Movement

Movement for the 3D FPS game differs greatly from any of the movement in game seen thus far. Due to the implementation of a third dimension, the player has access to an x-axis, y-axis, and z-axis. The player has the ability to traverse the x and z-axis by walking forwards backwards, left, and right. They are also able to access the y-axis by jumping, falling, or walking up and down inclined surfaces. The degree of freedom that 3D movement provides is incomparable to that of 2D games, as it provides many means through which the player can manipulate their position. This is the evolution of player freedom seen throughout the projects presented thus far. *Pong* gives one axis of movement. *Asteroids*, *Tetris*, and the 2D platformer give the player two axes of movement. The FPS gives the player a full three axes of movement.

5.2.2 Camera Management and Perspective

In creating an FPS, there is a great deal of attention on how the camera is managed. In the prototype FPS there are two cameras that are created for the player to observe the game world. The primary camera is the first person camera, being located where the head of the character is. This camera moves wherever the player's head is and shows the world as the player character would "see" the world. The secondary camera, which the player can switch to from the primary camera, is the third person character. This camera is located behind the player and follows them wherever they go. This camera shows the player character as they exist in the world around them. This camera is also constructed with additional features to improve gameplay. This third person camera has the problem of potentially being inside of or obscured by objects behind the player that are too close. This problem is mitigated by the third person camera by dynamically changing its position if it is too close to an object. This results in an overall improvement to gameplay for the player and accounts for known pitfalls of the third person camera.

5.2.3 Handling Projectiles and Combat

Something else the FPS introduces unique to other projects is the implementation of multiple objects with the capacity to affect one another through a damage manager. Since this is a FPS game, the aim is to shoot opponents to damage them until their health reaches zero. In this prototype, the player character and enemy characters are able to inflict damage by firing bullets at opponents are receive damage upon being hit. Through the usage of a damage manager, the game registers where a character has been hit, by which character they were hit by, and is able to calculate the appropriate amount of damage in accordance with this information.

Each individual projectile is instantiated by a gun scene held by its respective character. This gun is given an id according to the character that is holding it and each instantiated bullet is given this same id. This information is valuable for when the damage manager need to find which character shot the bullet. These features constitute the core of gameplay, setting a groundwork for customization of player and enemy characters.

5.2.5 3D Models and Animation Systems

Among the most complicated parts of designing an FPS is a creation of a fully animated 3D character. As far as differences go between the FPS prototype and prior games, this process has by far the greatest leap in complexity. The player character is constituted of a rigged and animated 3D model created in Blender. This means that not only is it a culmination of different 3D body parts, but each of these parts are assembled together through the use of a skeleton and are provided animation with adjustments to the values of that skeleton. Compared to the frame by frame animation used in the 2D platformer, this is both a different means of animation and a different medium of art as a whole. 2D games have also been seen using skeleton-based animation systems (e.g. Rain World), but such a system was unnecessary for the purposes of the 2D plaformer created. For 3D games such as this FPS, this animation style is the default due to the nature of the 3D environment.

6 Conclusion

Game development is a process that delves into the most minute details of constructing game objects and scripts that work together to create one coherent whole. It is during the process of creating a game that a developer must consider how each component contributes to the final product. While the developer is able to see the entire picture at all times, what they have built and how it works, the player is exposed to a limited scope of that work at a given time in the form of gameplay. From the moment a person picks up a game, they enter an intricate logical web created by the game developer that dictates the actions the player can take. A game developer's goal is to create a game that guides the player through the game in a way that is intuitive to the player and flows naturally, allowing them to appreciate gameplay to its fullest. The player comes first in the creation of a videogame, as it is their enjoyment that the game is created for.

With all that is done for the player, there is one question that comes to mind from the developer's perspective: what does the player know about the construction of a game? This thesis serves to provide insight into this question by showing what goes on behind the scenes of developing a video game. Each object of a game is intricately crafted in a manner that fits into the game it is part of and contributes to the gameplay experience of the player.

6.1 Future Work

There is a vast list of details involved in the game development process that were not touched in this work. A thorough breakdown of a more complicated game project than the games displayed in this paper would be able to cover a variety of concepts unexplored. Other directions for work outside of programming, such as art and sound design, would also be incredibly valuable for the purpose of showing more about game development from the ground up. A particular area of focus touched upon briefly in this paper is the 2D and 3D art animation process. There are many methods of creating an animated game character and implementing that character into a game that are worth investigating further.

Study into game development practices that make games more or less successful is another area that could also be valuable in future research. This could help contribute to the growing body of research into refining best practices in the game development process. Research can take a psychological approach that delves into the elements of game development and how each element of the game is observed by the player. Alternatively, a sociological approach can be taken to investigate players' reviews of games across the internet with consideration to each of those games and how they were constructed.

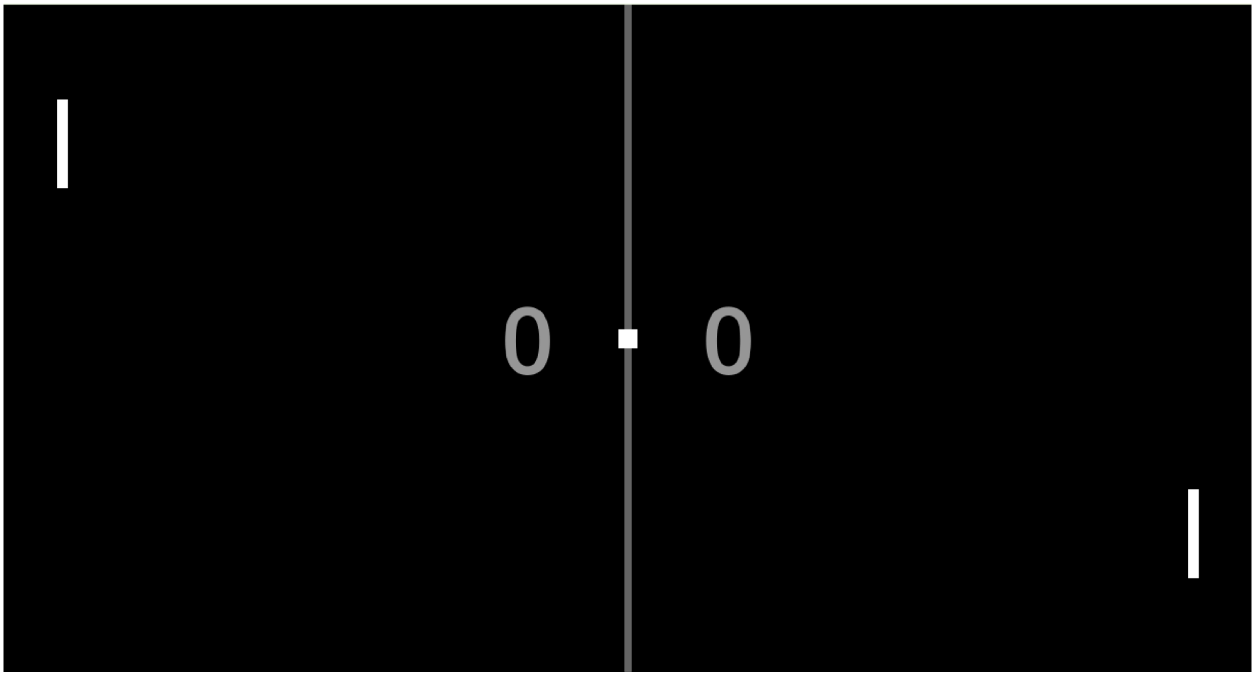
6.2 Final Remarks

I am fortunate to have had the opportunity to write this thesis in a subject matter that I love. Through this thesis, I was given the freedom to delve into new and creative projects in game development, a passion of mine. I hope that anyone who reads this paper can share some of the love I have for game development and that anyone reading can learn something new irregardless of technical background.

Appendix A - Screenshots

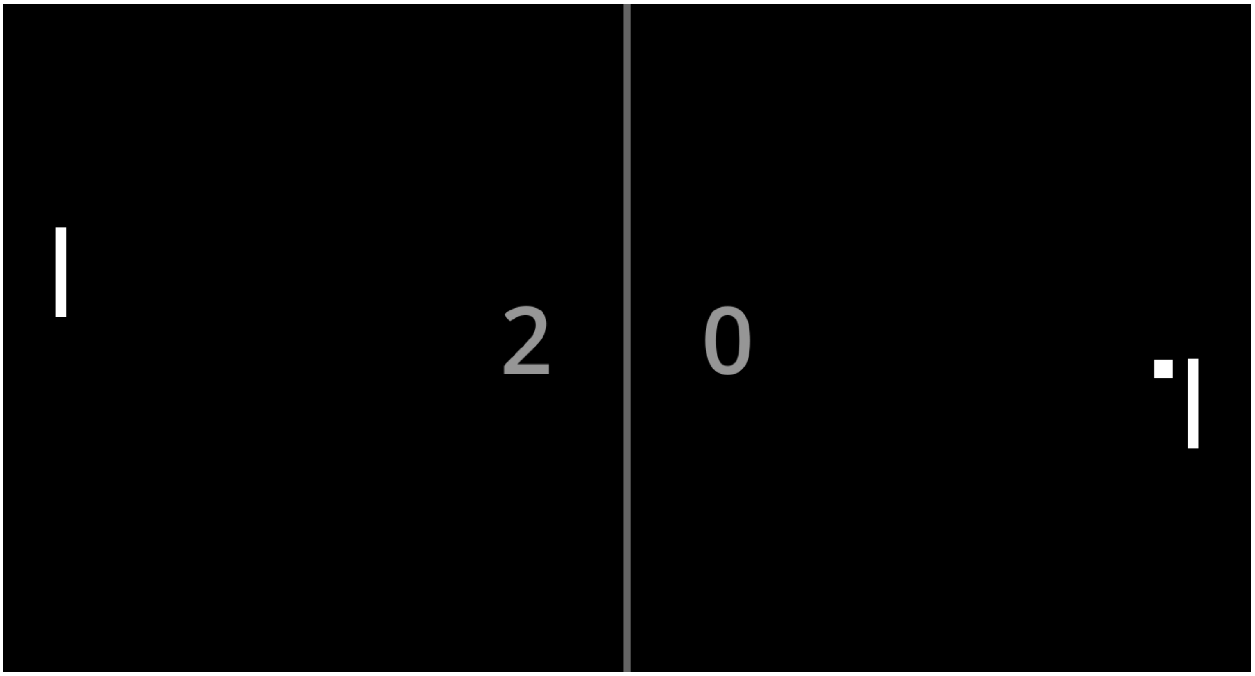
A1 Pong

Figure A1.1: Pong Game Initialization



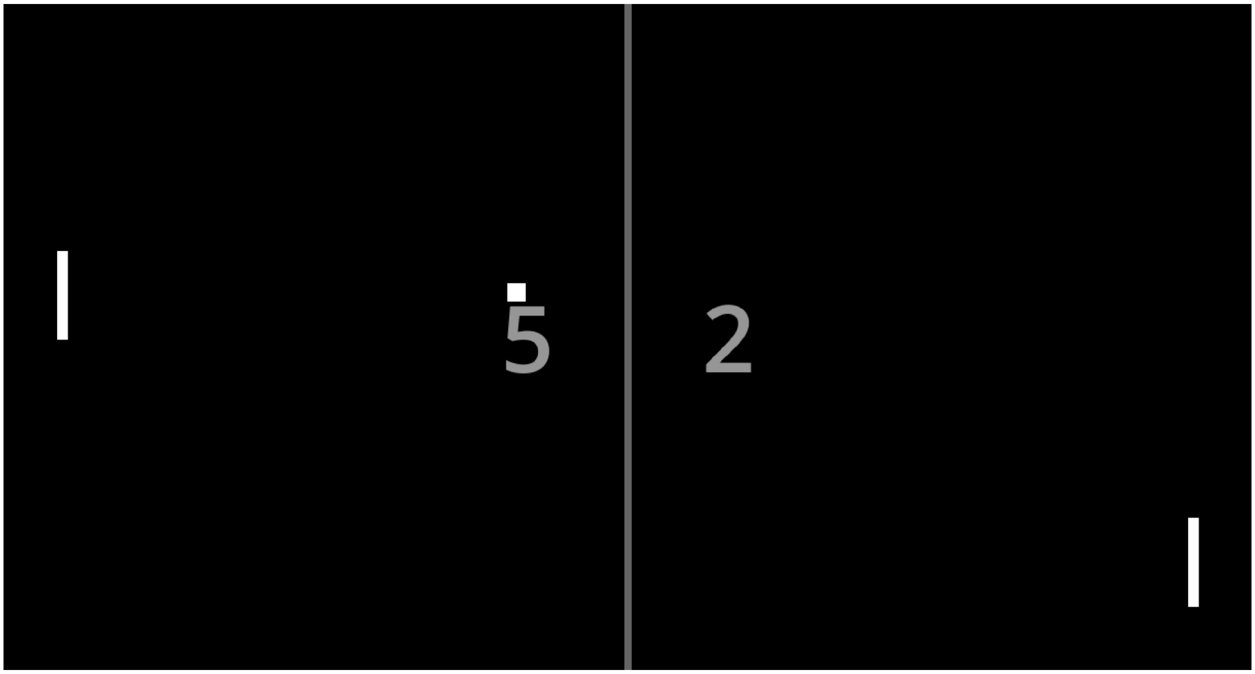
Initial game state. Player 1 controls the paddle on the left using the “W” and “S” keys on a keyboard. Player 2 controls the paddle on the right using the “UP” and “DOWN” keys.

Figure A1.2: Pong Gameplay 1



The game moves from the initial game state called the "start" state to the "running" state when the "SPACE" key or "ENTER" key is pressed. Upon entering the "running" state, the ball is propelled left or right at a random angle up or down with a maximum of 45 degrees. Upon starting the game, the ball moves right on the first turn and switches direction on every consecutive turn. The ball maintains its velocity as it moves across the screen, reflecting with a fully elastic collision upon colliding with the top or bottom of the screen. Upon colliding with the paddle on either side (right paddle in the image above), the ball reflects in the direction furthest from the center of the paddle collided with. The ball moves at the greatest vertical speed when colliding with the very edges of the paddle. In the image above, the ball strikes the upper portion of the right paddle, moving up and to the left as a consequence.

Figure A1.3: Pong Gameplay 2



As the game progresses, the players will score points when the ball crosses the edge of the screen on their opponents' side. In the image above, the player on the left has scored 5 points, while the player on the right has scored 2 points.

Figure A1.4: Pong Player Wins

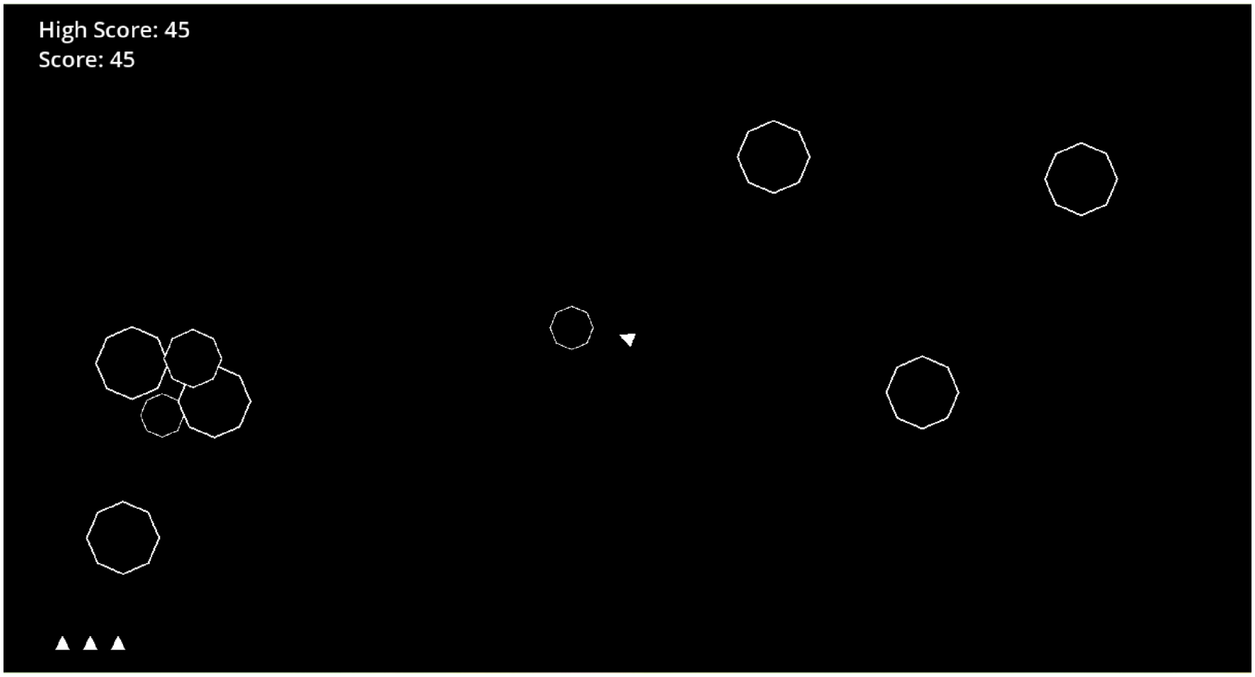
Once either of the players reach a score of 11, the game enters the “game_over” state. In this state, similar to the “start” state, the ball is positioned in the center of the screen and its momentum is arrested. It differs, however, in the fact that the text “PLAYER 1 WINS!” and “PLAY AGAIN” are displayed on the screen. Upon pressing the “SPACE” key or “ENTER” key the game is reset to the “start” state and the player scores are reset to 0.

A2 Asteroids

Figure A2.1: Asteroids Game Initialization

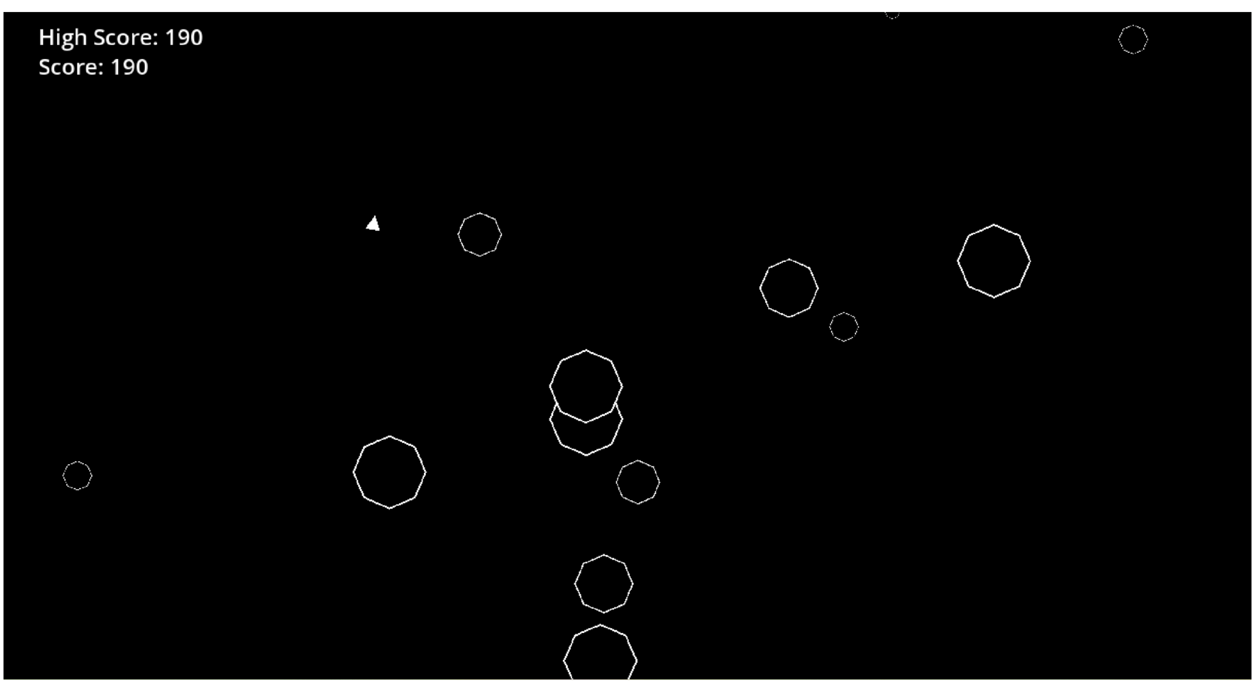
Upon starting the game, the player is presented with the screen above. There is the text "PLAY AGAIN?" with both the score and the high score displayed beneath it.

Figure A2.2: Asteroids Gameplay 1



Upon pressing the "SPACE" key or "ENTER" key during the start screen, the player will enter a brief period of invulnerability in which the ship flashes between visible and invisible.

Figure A2.3: Asteroids Gameplay 2

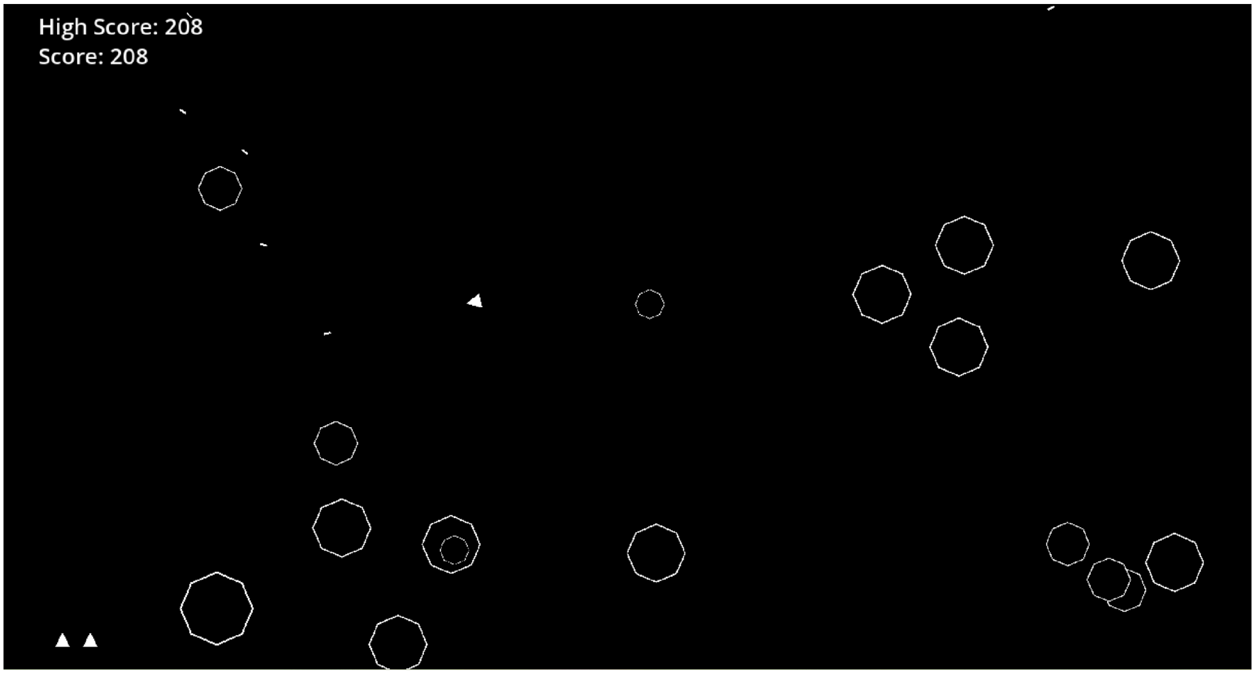


Over the course of the game, the player will end up colliding with the asteroids that spawn in over time. When the player collides with an asteroid, they enter a period of invulnerability, flashing between visible and invisible, and they lose a life. Consequently, one of the lives in the bottom left hand corner of the screen disappears.

Figure A2.4: Asteroids Game Over

When the player has no lives remaining in the bottom left corner of the screen and they collide with an asteroid, the game enters the “game_over” state. This state appears similar to that of the “start” state with some simple adjustments to the text shown.

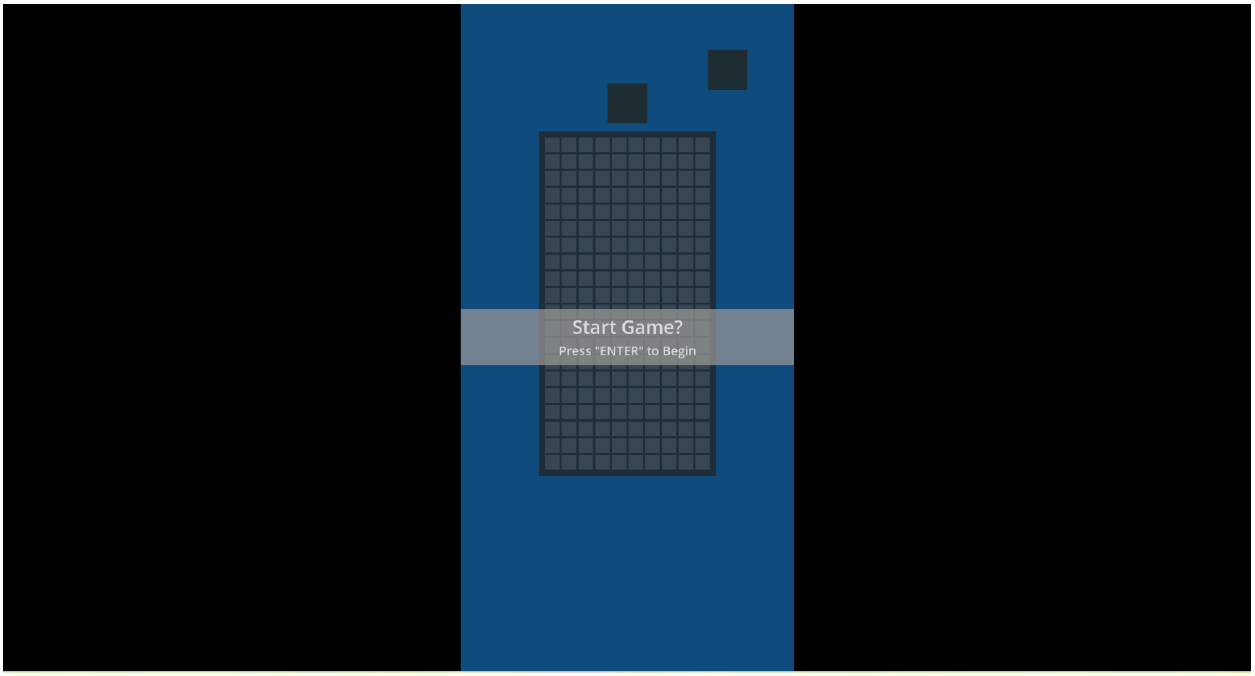
Figure A2.5: Asteroids Gameplay 3



Here the player is shooting by pressing either the “SPACE” key or “LEFT MOUSE BUTTON”. The lasers produced from this action are what break the asteroids and as a result lead to the score increasing. The high score shown is saved between plays of the game, while the high score persists across multiple playthroughs.

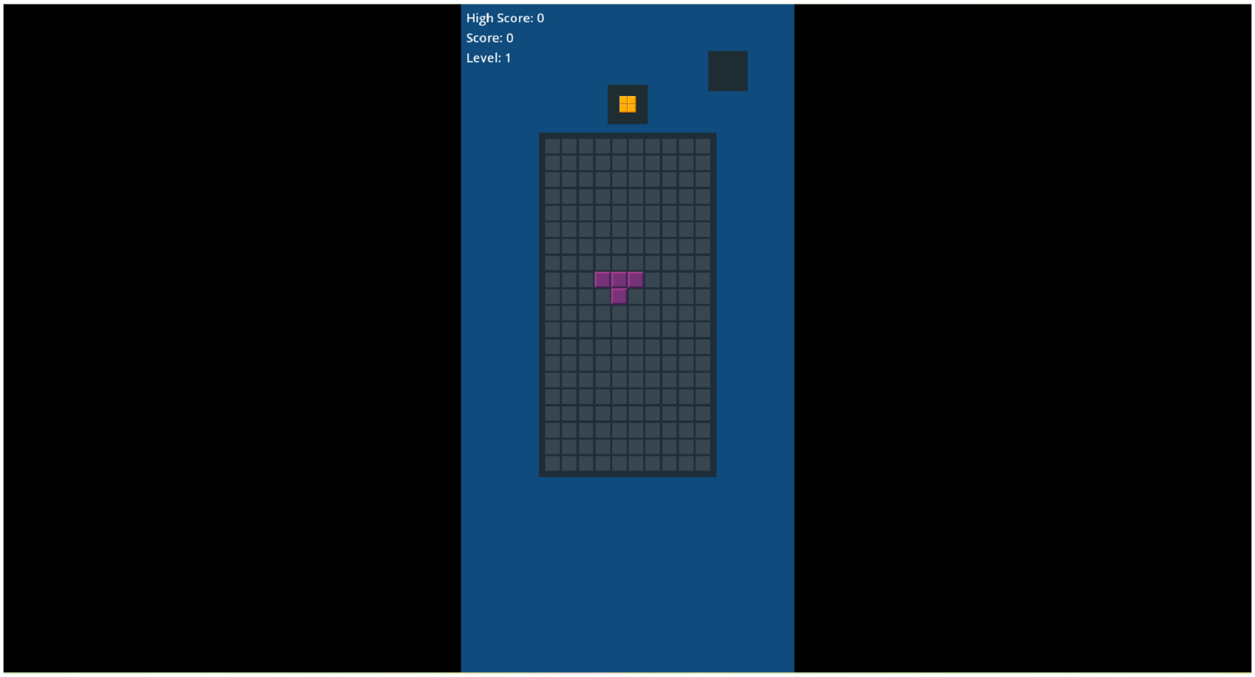
A3 Tetris

Figure A3.1: Tetris Game Initialization



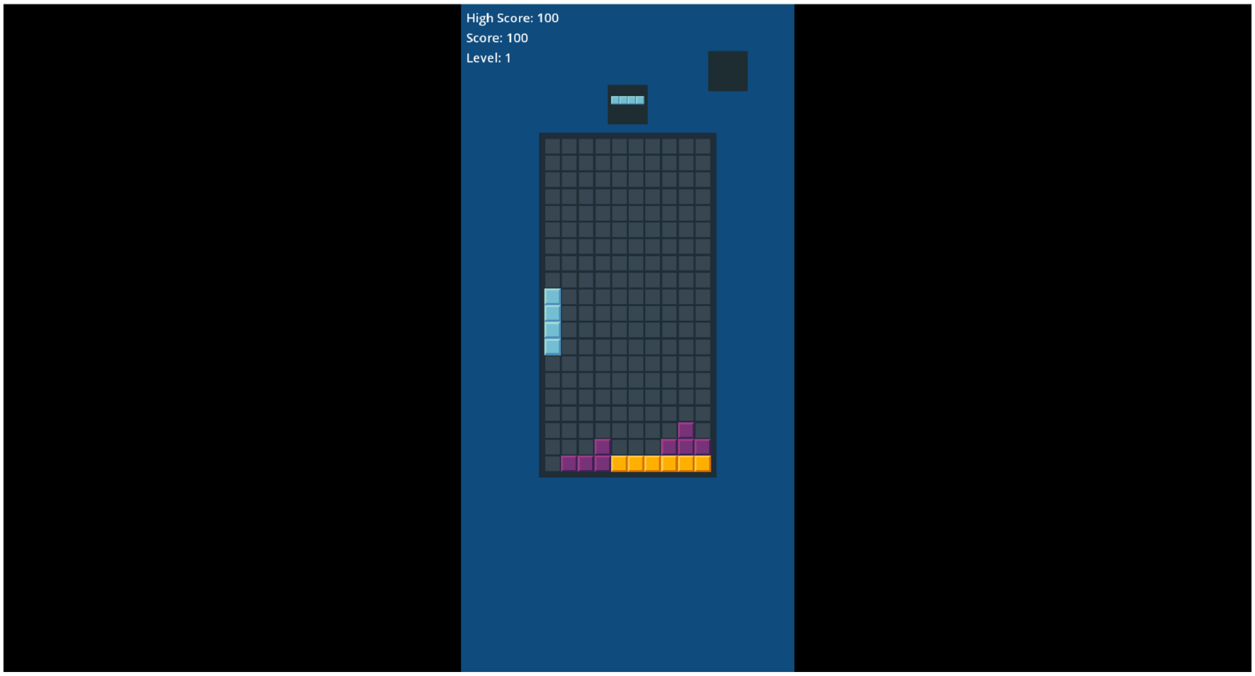
In the “Start” state, the game can only be moved to the “Running” state. There is also a banner across the screen that prompts the player to start the game this way.

Figure A3.2: Tetris Gameplay 1



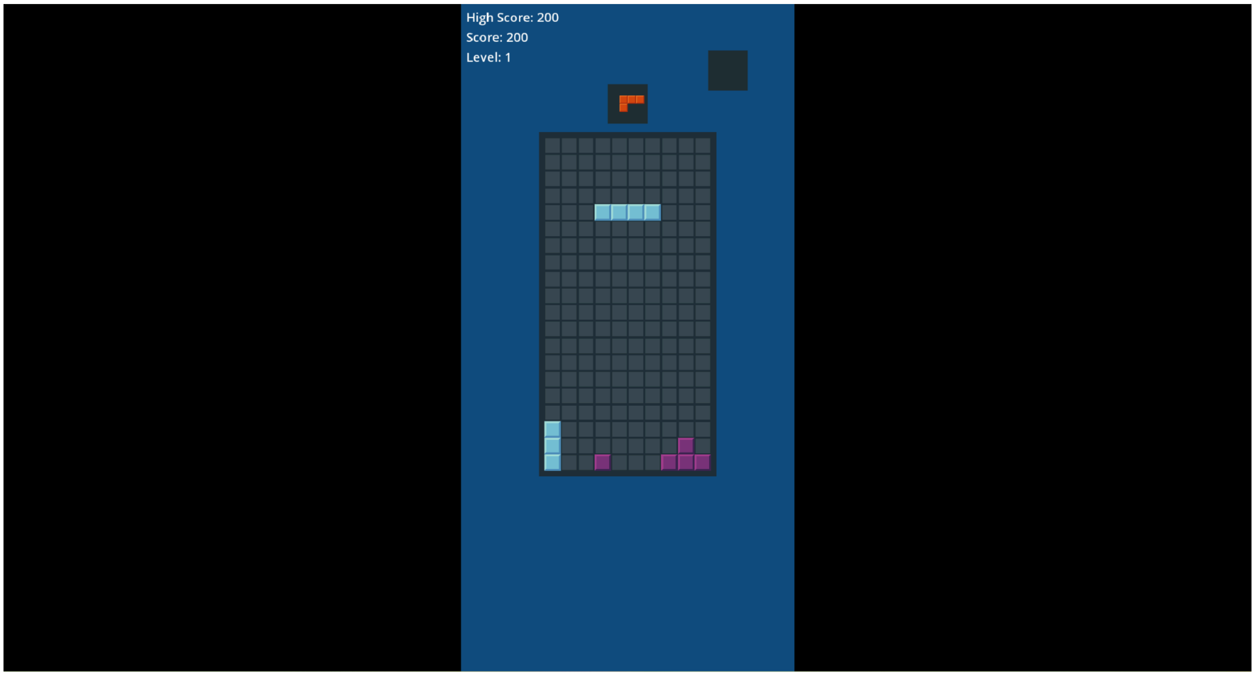
Upon starting the game and entering the "Running" state, one tetromino will be created and begin to descend while another tetromino is created and stored in the "DisplayTetromino" scene at the top of the board. These tetrominoes are randomly generated on the basis of a randomized seed based on runtime and can be of any shape.

Figure A3.3: Tetris Gameplay 2



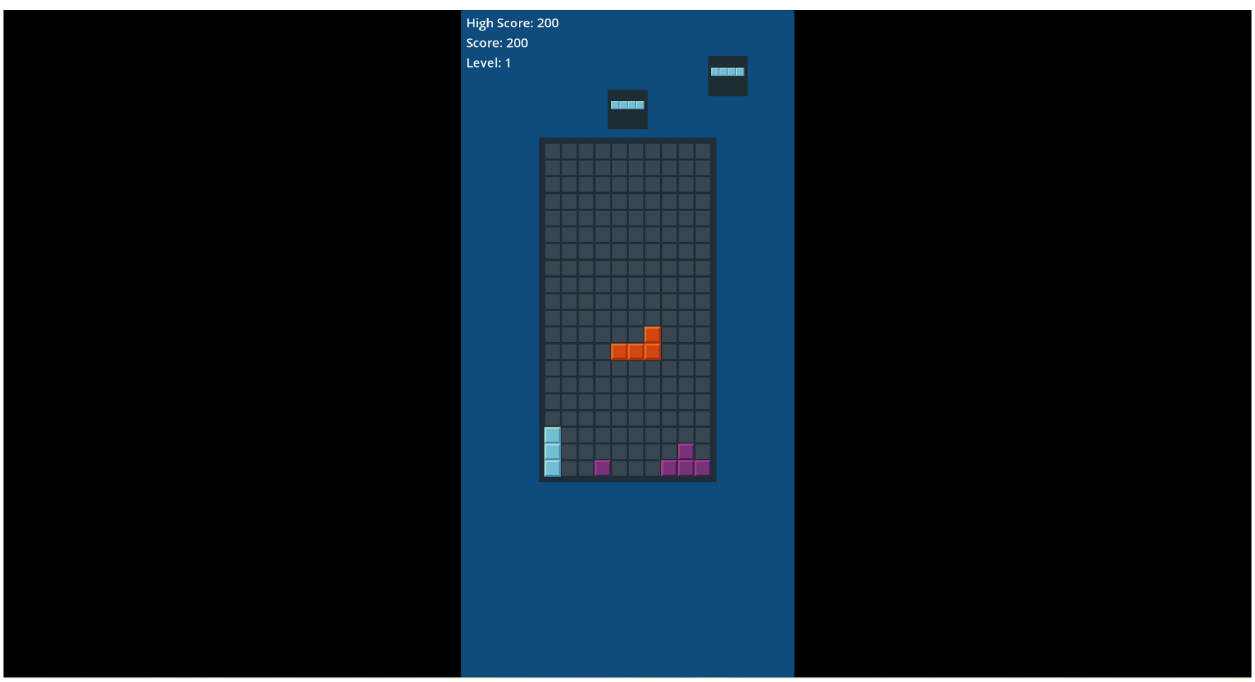
In this particular image, it can be seen that the player has gained 100 points to their score. For clearing a single row at a given point, the player is awarded 100 points multiplied by the level of the game at the point of scoring. In this case, the player has completed one line at level one, resulting in a score of 100.

Figure A3.4: Tetris Gameplay 3



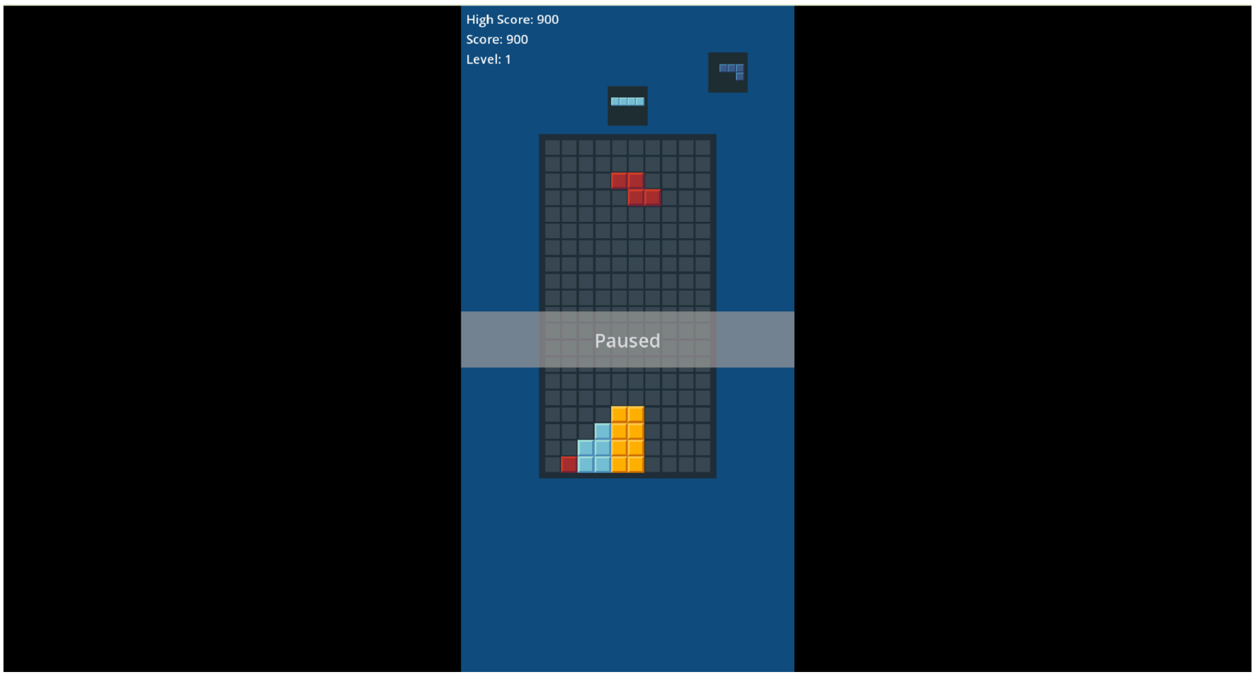
This figure is a direct continuation of figure 3.3A showing the player completing an additional line, bringing the score from 100 to 200.

Figure A3.5: Tetris Hold Tetromino



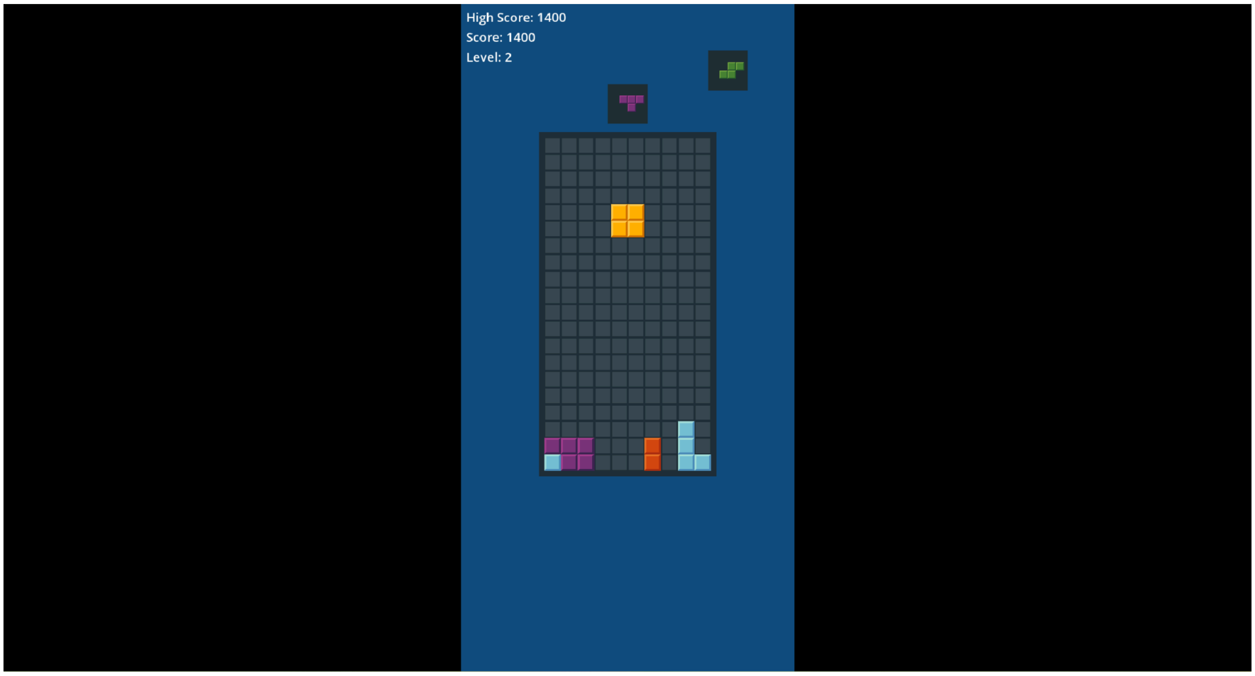
By pressing the “E” or “H” key, the player is able to use the “hold” mechanic. This moves the active tetromino to the space in the top right of the board. In this instance, there is no tetromino in the hold space which results in the active tetromino to be moved to the hold space and the display tetromino being made into the new active tetromino. If there is already a tetromino in the hold space, that tetromino and the active tetromino will simply be switched.

Figure A3.6: Tetris Pause Screen



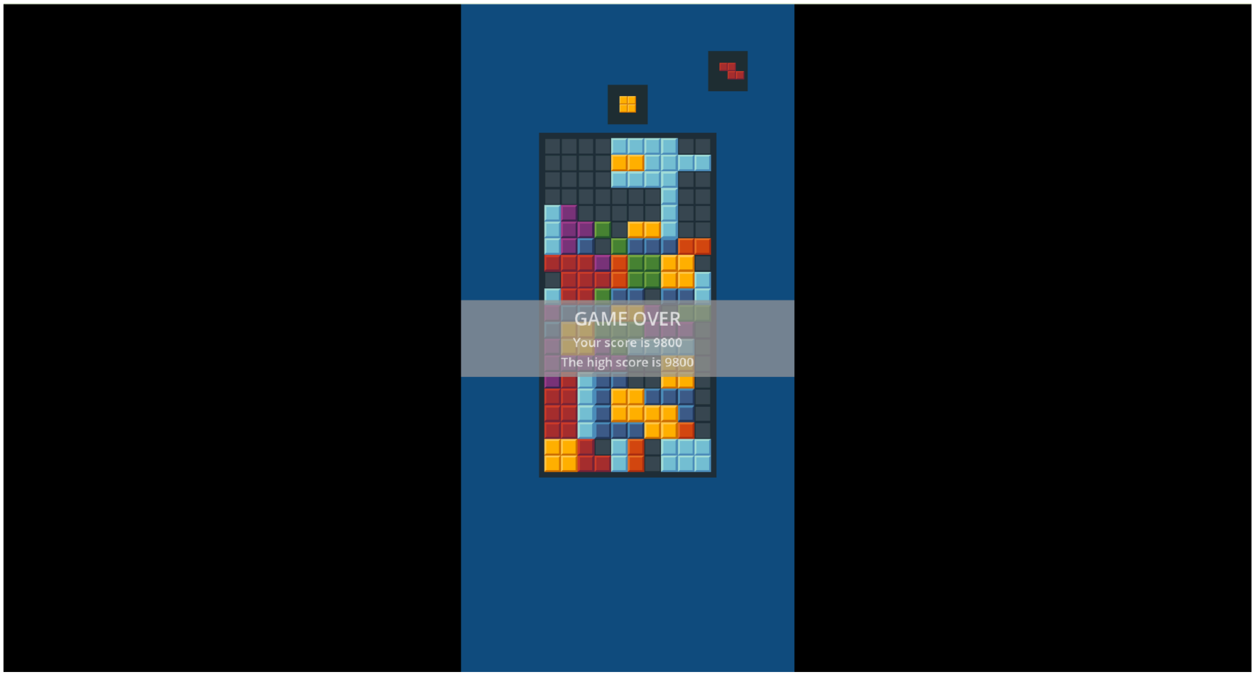
Presented here is a pseudo-state of the “Running” state of the game, the “Paused” state. During this state, there is no change to the state value in the main script, but rather the property of the main scene “paused” is set to true. This can only be done in the running state by pressing the “P” key. The game returns to normal operations upon pressing the “P” key again.

Figure A3.7: Tetris Gameplay 4



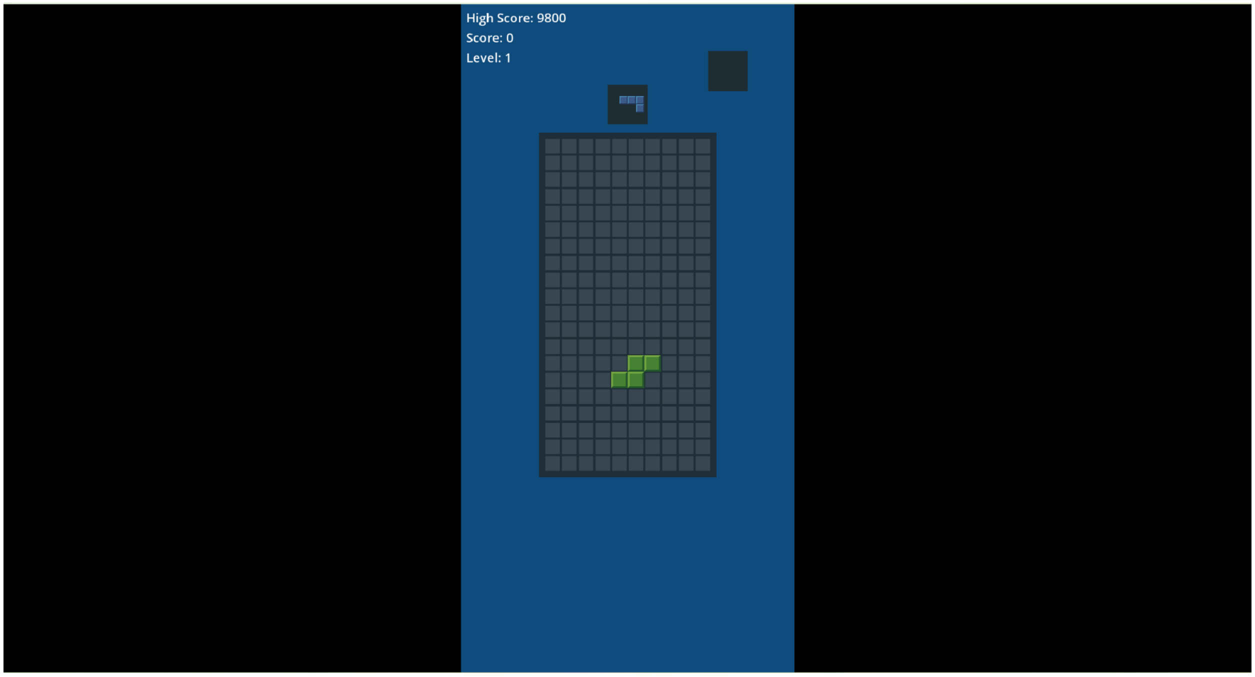
Upon crossing over certain point thresholds, the level of the game increases. With the increase in level, there are two major changes to gameplay: The first notable change is the increase in the speed by which the active tetromino falls. The period of time it takes for the tetromino to fall one space decreases with the accruement of levels up to a finite point. The second notable change is the increase in points earned from completing new lines. The points earned for the completion of one or more lines is multiplied by the level during calculation.

Figure A3.8: Tetris Game Over



As the difficulty level increases, the player will inevitably miss spaces, resulting in incomplete lines. When enough of these stack up and the tetromino cannot move down when it is at the very top of the board, the game enters the “GameOver” state. In this state, the player is presented with the score and high score achieved during their session of gameplay. To exit this state and return to the “Start” state, the player can press the “ENTER” key.

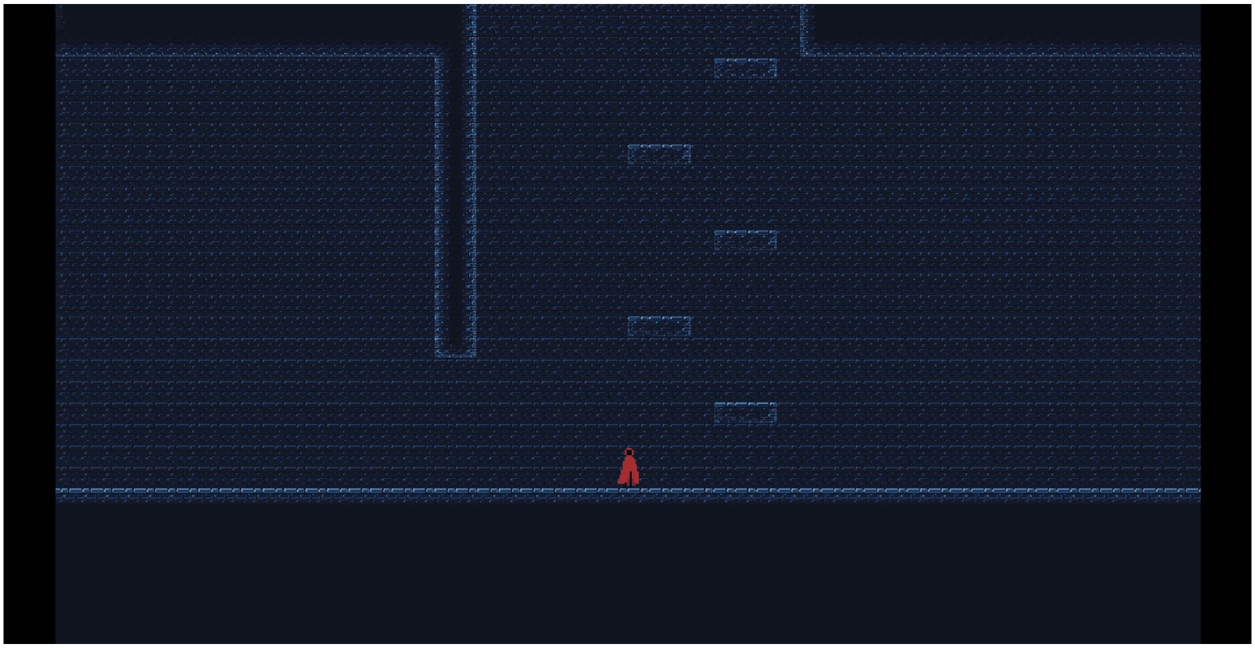
Figure A3.9: Tetris New Game



Upon starting a new game after a previous playthrough, the high score of previous games persists. This figure is a continuation of gameplay from figure 3.8A, in which the score achieved was 9800. This value persisted and is shown in the top left corner of the screen as a consequence. If a score exceeds 9800 during gameplay, that score will become the new high score.

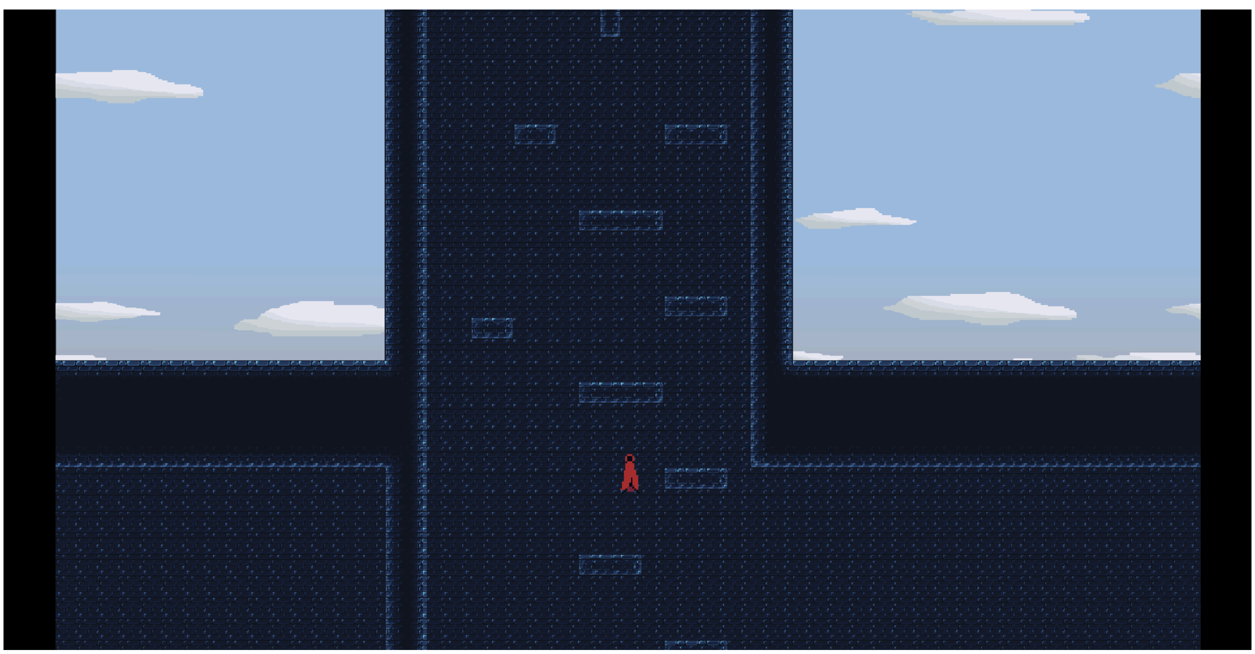
A4 Platformer

Figure A4.1: Platformer Game Initialization



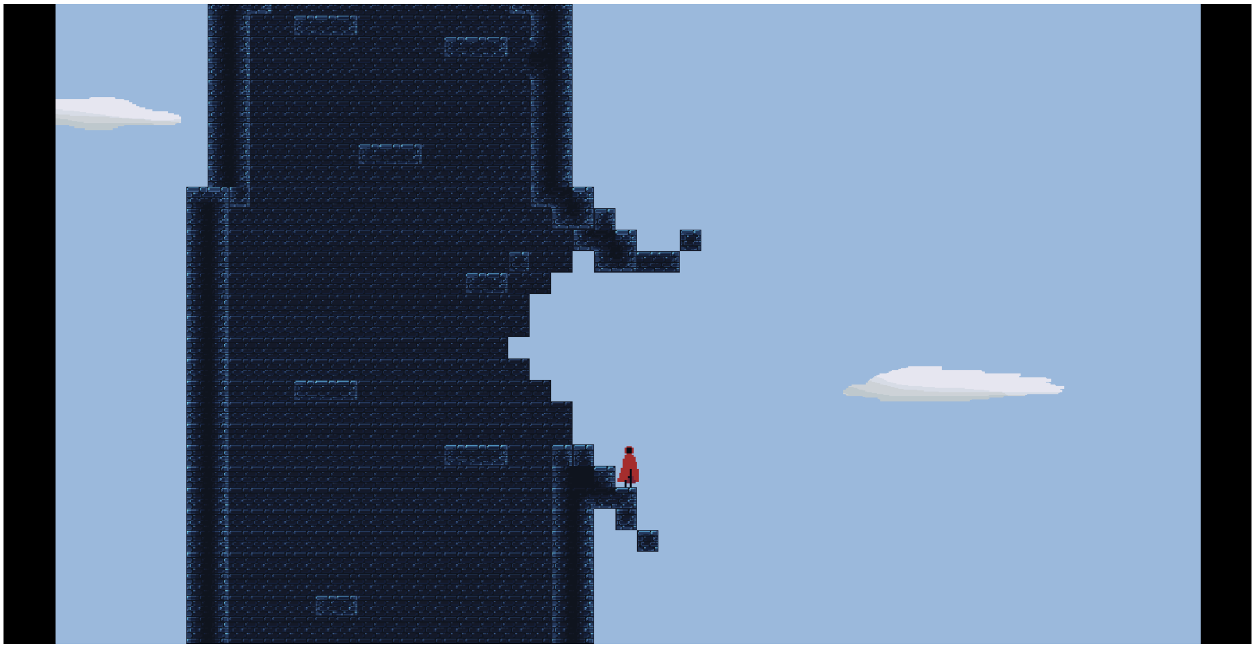
This is what the player sees upon starting the game. Here the player character is seen at the bottom of the screen in the center.

Figure A4.2: Jumping



In this image the player character can be seen in the “jump” state while jumping from one platform to the next. To perform this movement, the player presses the “SPACE” key to jump upwards and the “D/RIGHT” key to move to the right.

Figure A4.3: Platformer Gameplay 1

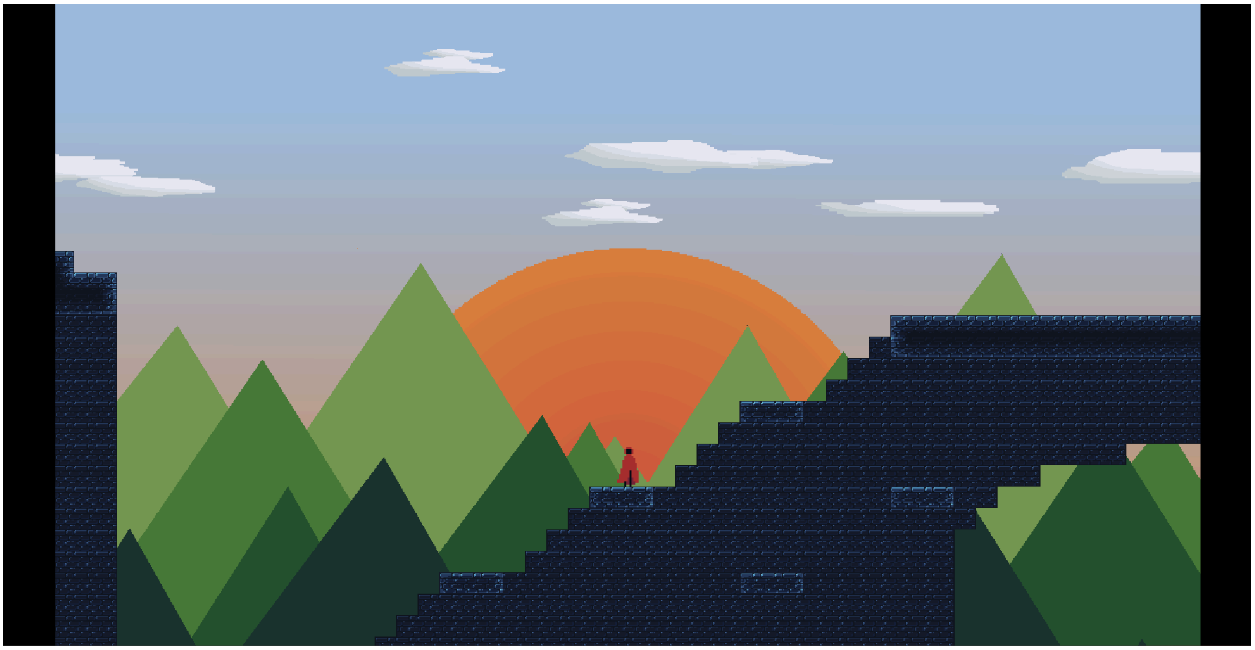


The player character can be seen here at the top of the tower. As the player stands here, the clouds in the background move across the screen to simulate an active and real environment.

Figure A4.4: Floating

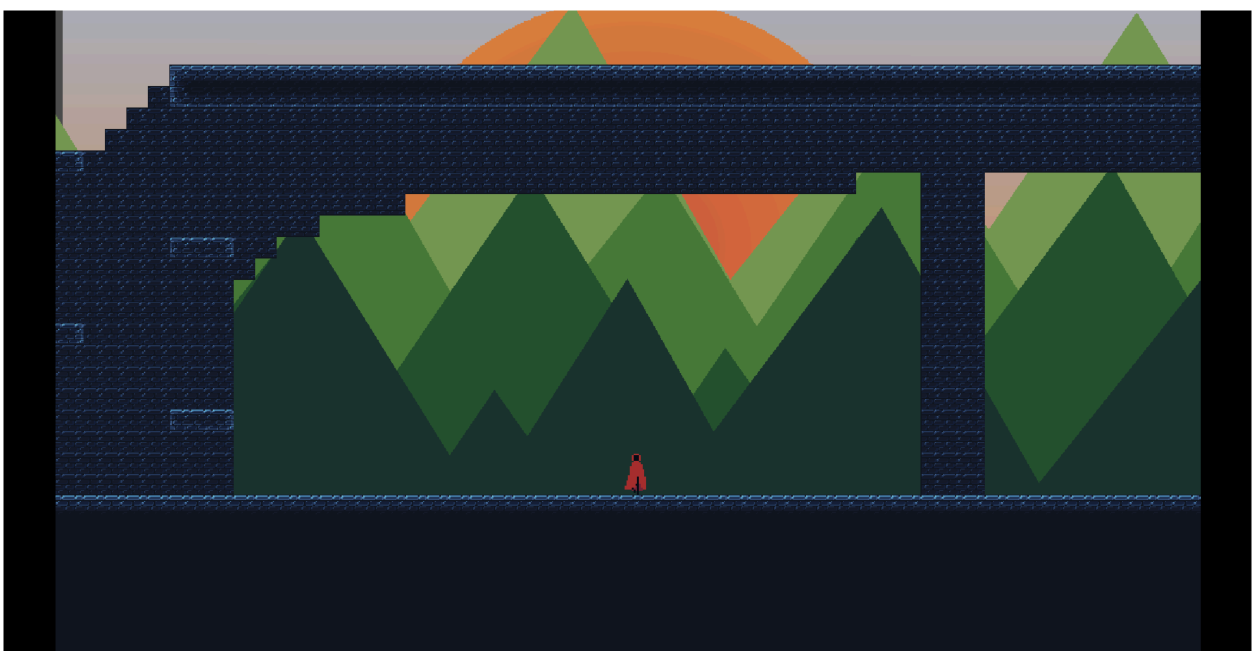
Shown here is the player in the “float” state, in which they descend through the air at a fixed rate. The player character enters this state when the player is holding the “W/UP” key in addition to the player character moving downward through the air at or above the speed of descent for the “float” state.

Figure A4.5: Platformer Gameplay 2



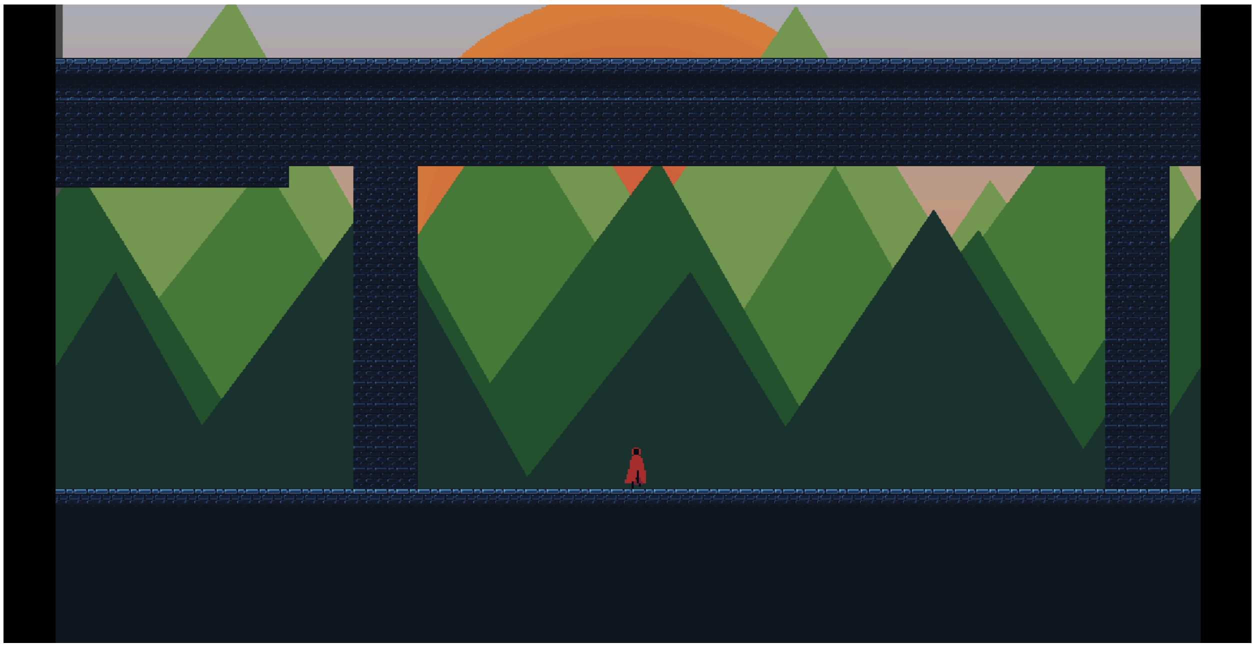
The player character can be seen here with a full display of all components of the parallax background. These components include the following: the sun, the sky, the mountains and the clouds.

Figure A4.6: Parallax 1



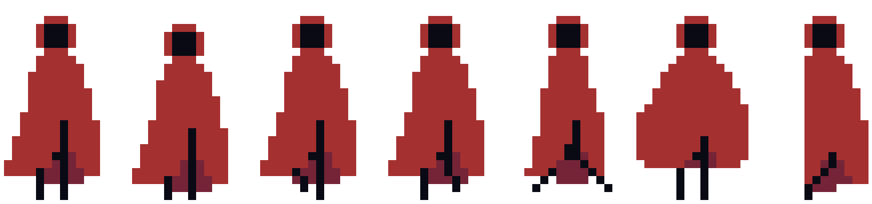
The background behind the player is displayed here and provides a reference point for the dynamic movement of the background in accordance with parallaxing (see figure A4.7).

Figure A4.7: Parallax 2



A shift in all elements of the parallax background can be seen here, with reference to prior positioning of the player character (see figure A4.6). Elements that are meant to be closest to the player (e.g. darkest mountains) move the most relative to the player, while elements meant to be furthest away from the player move the least relative to the player (e.g. lightest mountain).

Figure A4.8: Platformer Character



Here is the sprite-sheet that is used to make the player character's animations. In this sprite sheet, there are five animations of note, two of which are two frames and three of which are one frame each. The first and second frames are responsible for creating the idle animation in which the player bobs up and down. This animation plays when the player is stationary and making contact with the ground. The third frame and fourth frame make up the walking animation. This animation plays when the player is moving while touching the ground. The fifth frame makes up the jump/falling animation which plays while the player is in the air. The sixth frame makes up the float animation which plays while the player is in the air falling down and holding the "W/UP" key causing their descent to be slowed. The seventh frame makes up the slide animation which plays while the player is moving while pressed against a wall.

A5 First Person Shooter

Figure A5.1: First Person Shooter Game Initialization

This is the "first person view" of the first person shooter in which the player is viewing the environment through the player character's eyes. In this view, the arms of the player and their gun are visible, alongside features of the environment and the player's UI, which is composed of a simple health bar in the bottom left corner. From this perspective, features of the environment include scattered trees and the ground.

Figure A5.2: First Person Shooter Enemies

From this perspective, two simple enemies can be seen shooting in front of themselves. These are exceptionally simple enemies who shoot in a straight line in front of them on a timer. We can see some additional features of the environment including stones and a horizon from here.

Figure A5.3: First Person Shooter Third Person

Despite having the title of "first person shooter" such games are not always resolved to solely a first person view. Seen here is the player in their "third person view" in which the player is viewed from behind.

Figure A5.4: First Person Shooter Jump Animation

The player is capable of the “jump” action which provides them vertical force concurrent with the jump animation seen here.

Figure A5.5: First Person Shooter Walk Animation

Like the previous figure 5.4A demonstrates, the player can perform the “walk” action which plays the walking animation simultaneously.

Figure A5.6: First Person Shooter Player Damage

Upon being struck with a damaging projectile, in this case a bullet, the player is damaged. At the instance of impact, the player receives damage which updates their health total which begins at 100. This health reduces by the amount of damage caused.

Appendix B - Code

B1 Pong

Figure B1.1: ball.gd

```
extends CharacterBody2D

@export var initSpeed = 300
@onready var speed = Vector2.ZERO

func _physics_process(_delta):
	runMovementBehavior()
	move_and_slide()

#Regulates the movement of the ball scene
func runMovementBehavior():
	velocity = speed
	
	#Upon collision with a surface, run this behavior
	if is_on_wall():
		var normal = get_wall_normal()
		
		#If ball collides with paddle, run this behavior
		if normal.x and $RefX.is_stopped():
			speed.x = -speed.x
			speed.x *= 1.05
			
			#Based on distance from paddle center, increase or decrease vertical speed
			if position.x > 640:
				speed.y = (position.y - Globals.paddle2_position.y)/50 * abs(speed.x)
			else:
				speed.y = (position.y - Globals.paddle1_position.y)/50 * abs(speed.x)
			$RefX.start()
		
		#If ball collided with wall, run this behavior
		if normal.y and $RefY.is_stopped():
			speed.y = -speed.y
			$RefY.start()
```

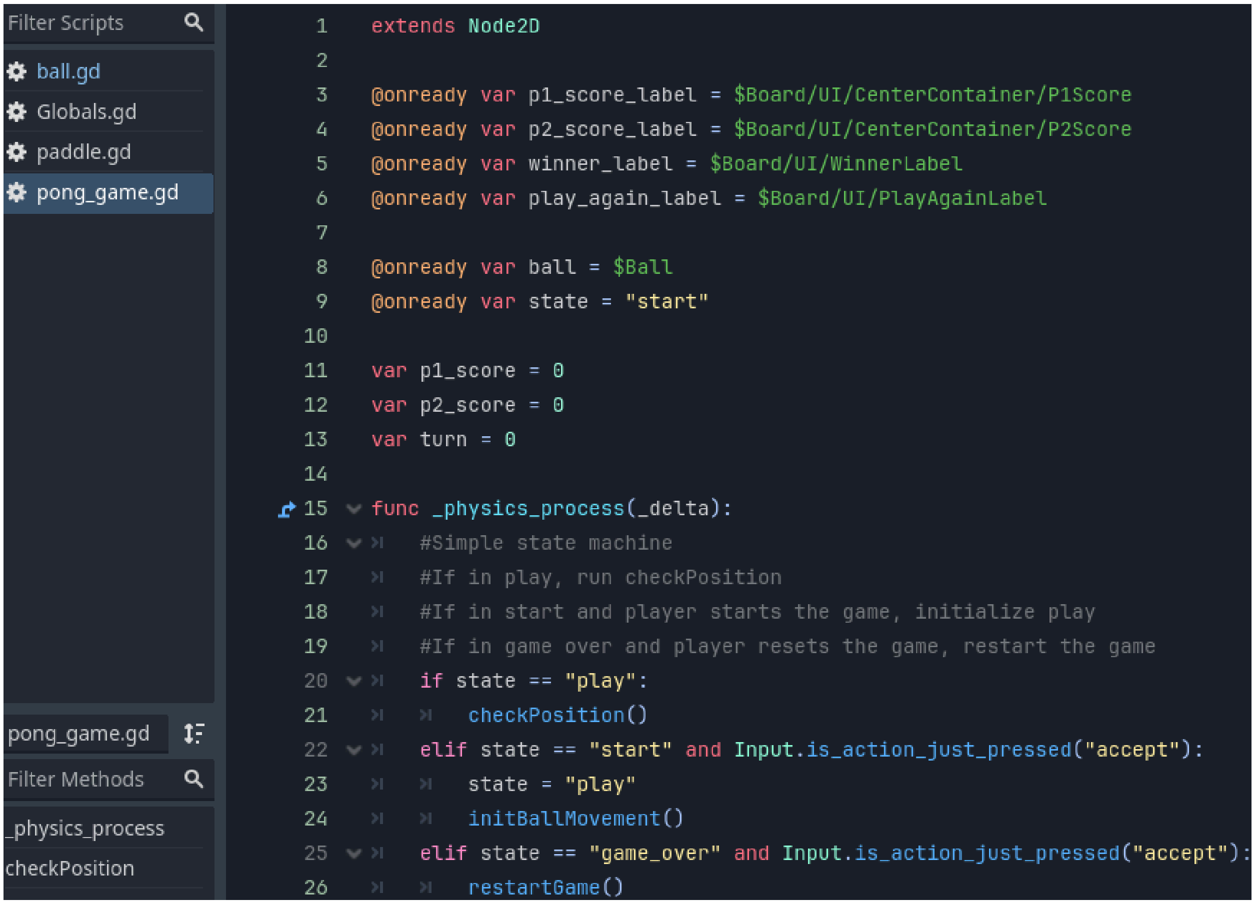
This script has the purpose of managing the movement behavior of the ball scene by running the runMovementBehavior() function every frame. In this function, the velocity of the ball is set to the ball's speed, and the speed of the ball being set according to logical processes that follow the setting of the velocity. To clarify, the velocity of the ball is a property of the object that is used in the move_and_slide() function which works to move the object in accordance with its velocity. Meanwhile, the speed of the ball is a variable the script modifies to set its velocity.

Figure B1.2: paddle.gd

```
extends CharacterBody2D

@export var player_num : int
@export var speed = 300

func _physics_process(_delta):
	var direction : float
	var up_is_pressed : bool
	var down_is_pressed : bool

	#Paddle movement is dictated by player 1 OR player 2
	#Player 1 uses "WASD"
	#Player 2 uses arrow keys
	if player_num == 1:
		Globals.paddle1_position = position
		direction = Input.get_axis("up1","down1")
		up_is_pressed = Input.is_action_pressed("up1")
		down_is_pressed = Input.is_action_pressed("down1")
	elif player_num == 2:
		Globals.paddle2_position = position
		direction = Input.get_axis("up2","down2")
		up_is_pressed = Input.is_action_pressed("up2")
		down_is_pressed = Input.is_action_pressed("down2")

	#Move if one direction is pressed
	#Don't move if no direction is pressed or both directions are pressed
	if up_is_pressed and down_is_pressed:
		velocity.y = 0
	elif direction:
		velocity.y = speed * direction
	else:
		velocity.y = 0
	move_and_slide()
```

This script regulates the movement behavior of the paddle script on the basis of player input. Based on a player_num that is provided to the script to designate player one and two, the paddle will move up or down at the set speed according to either the “W/S” or “UP/DOWN” key input.

Figure B1.3: pong_game.gd 1



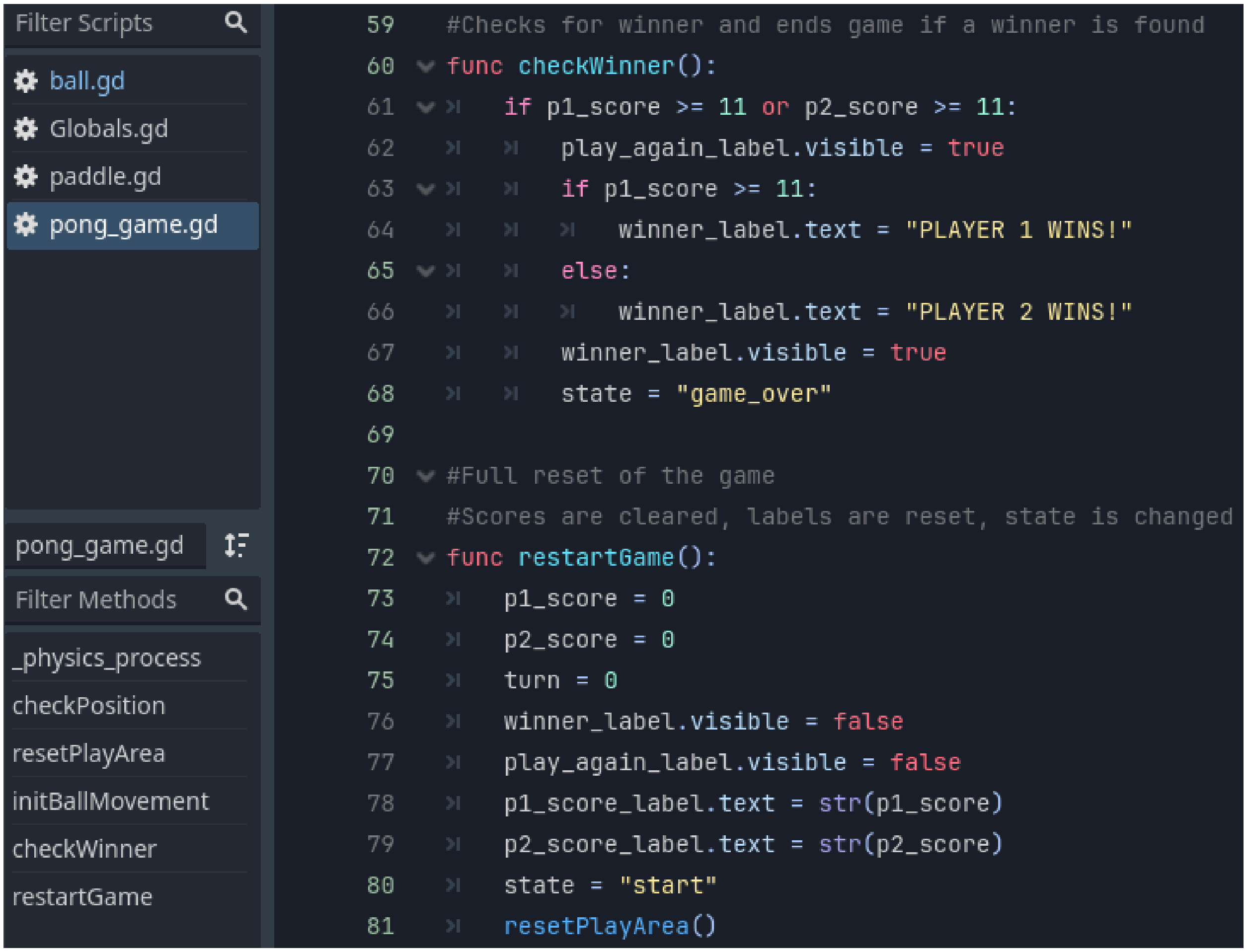

```
extends Node2D

@onready var p1_score_label = $Board/UI/CenterContainer/P1Score
@onready var p2_score_label = $Board/UI/CenterContainer/P2Score
@onready var winner_label = $Board/UI/WinnerLabel
@onready var play_again_label = $Board/UI/PlayAgainLabel

@onready var ball = $Ball
@onready var state = "start"

var p1_score = 0
var p2_score = 0
var turn = 0

func _physics_process(_delta):
	#Simple state machine
	#If in play, run checkPosition
	#If in start and player starts the game, initialize play
	#If in game over and player resets the game, restart the game
	if state == "play":
		checkPosition()
	elif state == "start" and Input.is_action_just_pressed("accept"):
		state = "play"
		initBallMovement()
	elif state == "game_over" and Input.is_action_just_pressed("accept"):
		restartGame()
```

This is the main script of the game Pong. In this code, functions are run each frame according to a simple state machine in _physics_process(delta). If the state is “play”, the checkPosition() function is called. If the state is “start” and the “ENTER/SPACE/LEFT MOUSE BUTTON” key is pressed, the state changes to “play” and the initBallMovement() function (see figure B1.4) is called. If the state is “game_over” and the “ENTER/SPACE/LEFT MOUSE BUTTON” key is pressed, the restartGame() function (see figure B1.5) is called.

Figure B1.4: pong_game.gd 2

Continuation of the pong_game.gd script. Here are the checkPosition(), resetPlayArea(), and initBallMovement() functions. The checkPosition() function searches for the position of the ball and assigns a point to a player if its X coordinates exceed the boundaries of the play area followed by updating the labels accordingly and calling resetPlayArea(). The resetPlayArea() function sets the ball's position to the center of the board, its velocity to zero, and sets the game state to "start" so that the next round of play can be initiated. Lastly, the initBallMovement() function is responsible for initiating the ball's movement, switching which direction the ball is moving each turn and sending the ball at a randomized angle of up to 45 degrees of magnitude.

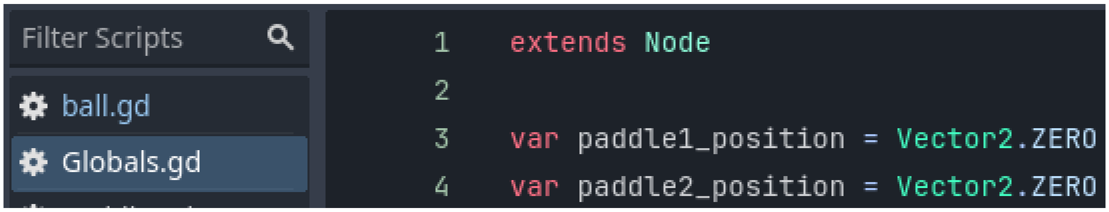
Figure B1.5: pong_game.gd 3




```
#Checks for winner and ends game if a winner is found
func checkWinner():
	if p1_score >= 11 or p2_score >= 11:
		play_again_label.visible = true
		if p1_score >= 11:
			winner_label.text = "PLAYER 1 WINS!"
		else:
			winner_label.text = "PLAYER 2 WINS!"
		winner_label.visible = true
		state = "game_over"

#Full reset of the game
#Scores are cleared, labels are reset, state is changed
func restartGame():
	p1_score = 0
	p2_score = 0
	turn = 0
	winner_label.visible = false
	play_again_label.visible = false
	p1_score_label.text = str(p1_score)
	p2_score_label.text = str(p2_score)
	state = "start"
	resetPlayArea()
```

Continuation of the pong_game.gd script. Here is what remains of the main script, containing the checkWinner() and restartGame() functions. The checkWinner() function checks whether either players’ score exceeds or is equal to 11 and shows text on the screen to congratulate that player. The game state is then switched to “game_over”, from which the player can run the restartGame() function. The restartGame() function acts to reset all values that could be modified during gameplay to their original state followed by a run of resetPlayArea().

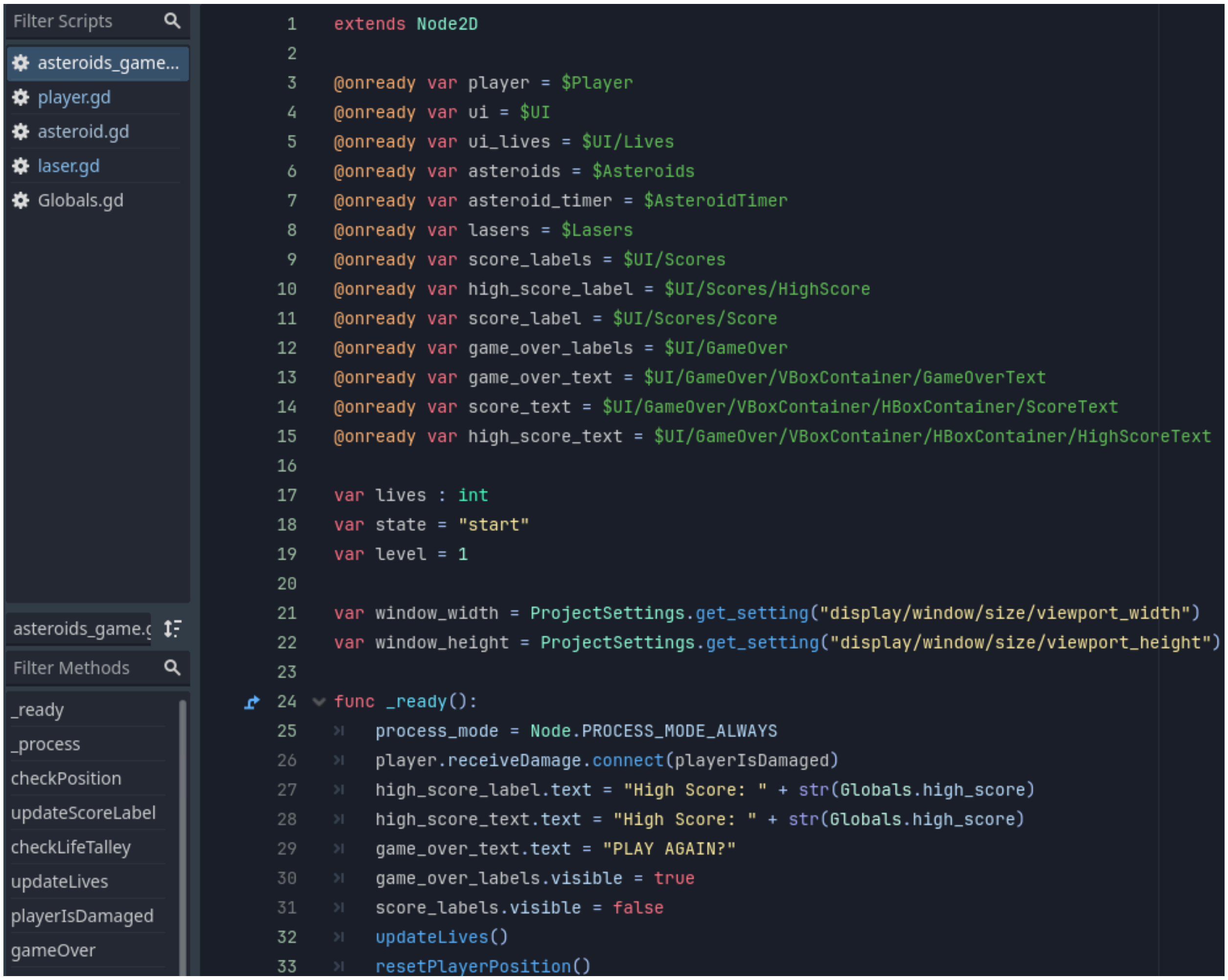
Figure B1.6: Globals.gd



These are two global values which are set by the paddle.gd script (see figure B1.2) and utilized by the ball.gd script (see figure B1.1). Various means could be used to communicate this information from one scene to the other, but this method is the most simple.

B2 Asteroids

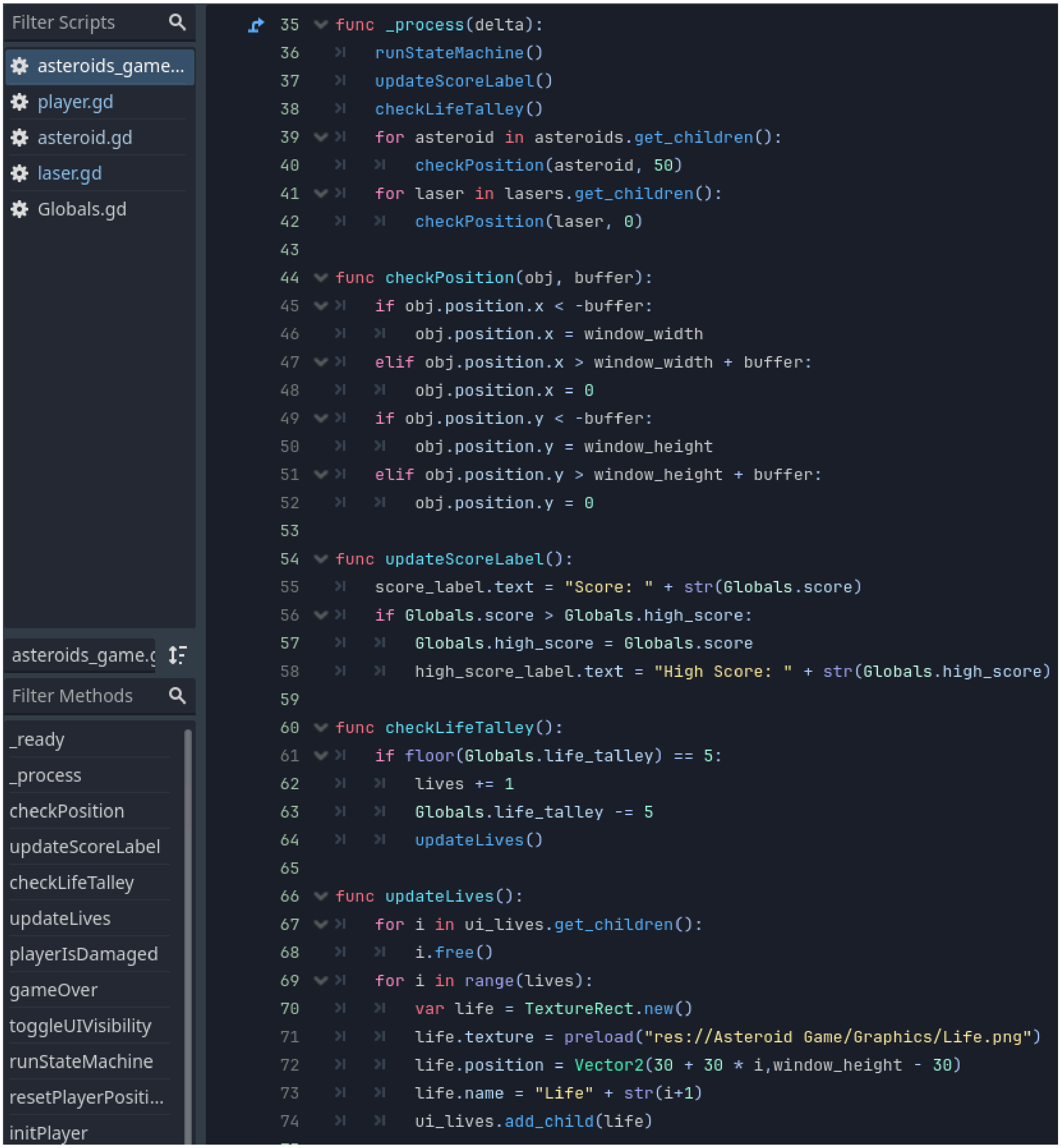
Figure B2.1: asteroids_game.gd 1




```
extends Node2D

@onready var player = $Player
@onready var ui = $UI
@onready var ui_lives = $UI/Lives
@onready var asteroids = $Asteroids
@onready var asteroid_timer = $AsteroidTimer
@onready var lasers = $Lasers
@onready var score_labels = $UI/Scores
@onready var high_score_label = $UI/Scores/HighScore
@onready var score_label = $UI/Scores/Score
@onready var game_over_labels = $UI/GameOver
@onready var game_over_text = $UI/GameOver/VBoxContainer/GameOverText
@onready var score_text = $UI/GameOver/VBoxContainer/HBoxContainer/ScoreText
@onready var high_score_text = $UI/GameOver/VBoxContainer/HBoxContainer/HighScoreText

var lives : int
var state = "start"
var level = 1

var window_width = ProjectSettings.get_setting("display/window/size/viewport_width")
var window_height = ProjectSettings.get_setting("display/window/size/viewport_height")

func _ready():
	process_mode = Node.PROCESS_MODE_ALWAYS
	player.receiveDamage.connect(playerIsDamaged)
	high_score_label.text = "High Score: " + str(Globals.high_score)
	high_score_text.text = "High Score: " + str(Globals.high_score)
	game_over_text.text = "PLAY AGAIN?"
	game_over_labels.visible = true
	score_labels.visible = false
	updateLives()
	resetPlayerPosition()
```

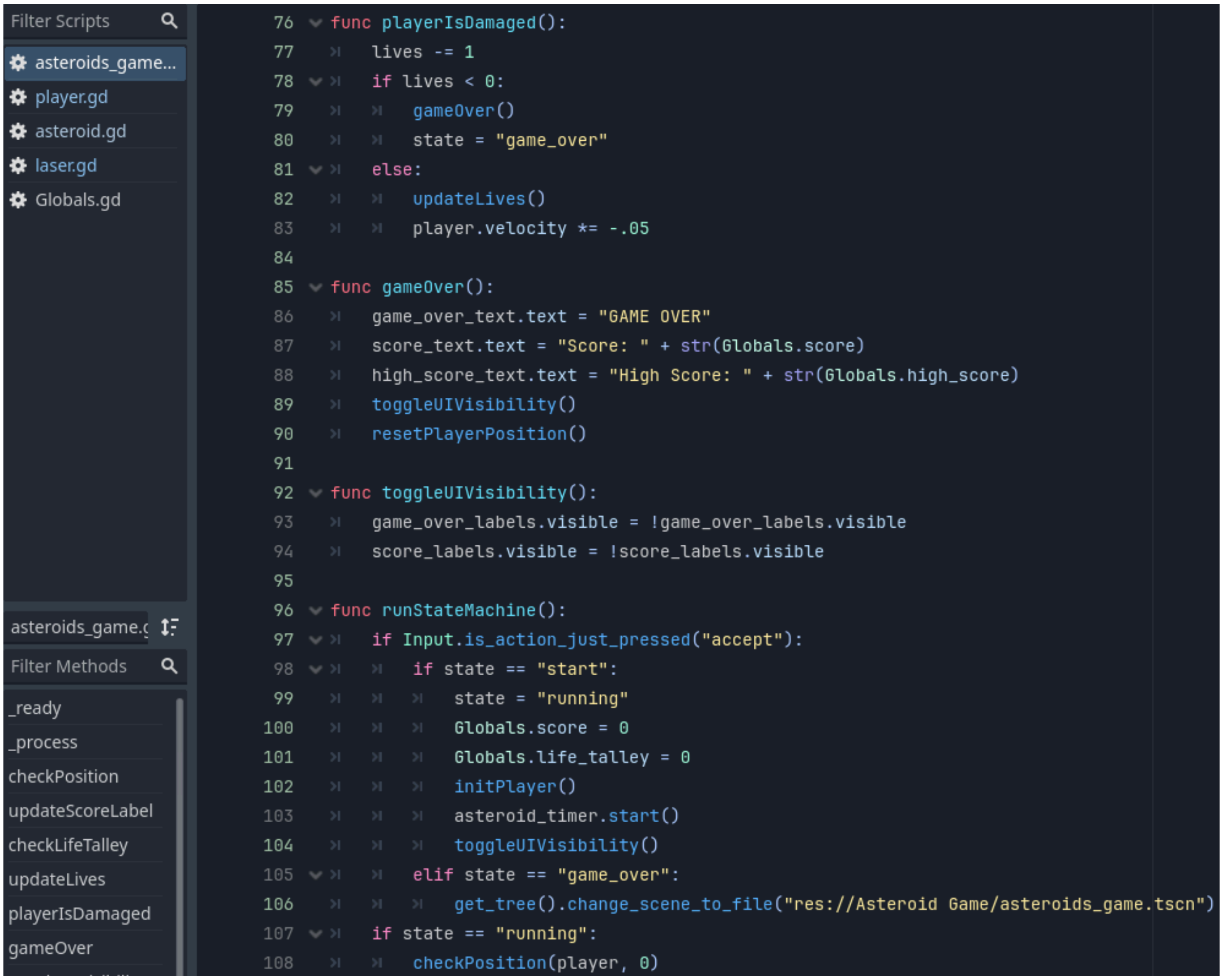
This is the beginning of the asteroids_game.gd script, the main script of the game *Asteroids*. In this portion of the script only the initialized values and processes can be seen. Upon instantiation, the main script connects the "playerIsDamaged" signal from the player scene to the receiveDamage() function, values are changed in accordance with global values, labels are made invisible, and the updateLives() function followed by the resetPlayerPosition() function are called.

Figure B2.2: asteroids_game.gd 2



Continuation of the asteroids_game.gd script. Here we see the logic run each frame in _process(delta) and three functions of the main script: checkPosition(), updateScoreLabel(), checkLifeTalley() and updateLives(). Each frame the runStateMachine() function is run (see figure B2.3), followed by updateScoreLabel() and checkLifeTalley(), followed by the checkPosition() function, which takes every asteroid and laser as an argument. The checkPosition() function takes in an object and a buffer as arguments, the buffer being different for asteroids and lasers. This function then determines whether the object exceeds the boundaries of the board with the provided buffer. If the object exceeds that range, its position is flipped to the opposite side of the board where it has surpassed that limit. The updateScoreLabel() function changes the text of the score labels to match the scores they are meant to reflect. It sets the high score to be equal to the score if the new score exceeds the former high score. The checkLifeTalley() function is responsible for reading whether the "life_talley" global exceeds a certain threshold and responding to such an occurrence by resetting the talley, adding one life, and calling updateLives(). The updateLives() function is responsible for updating the visual representation of the player's remaining lives. It does this through the usage of a loop, with the range of the number of lives, to generate a sprite of the player evenly spaced apart in the bottom left corner of the screen.

Figure B2.3: asteroids_game.gd 3



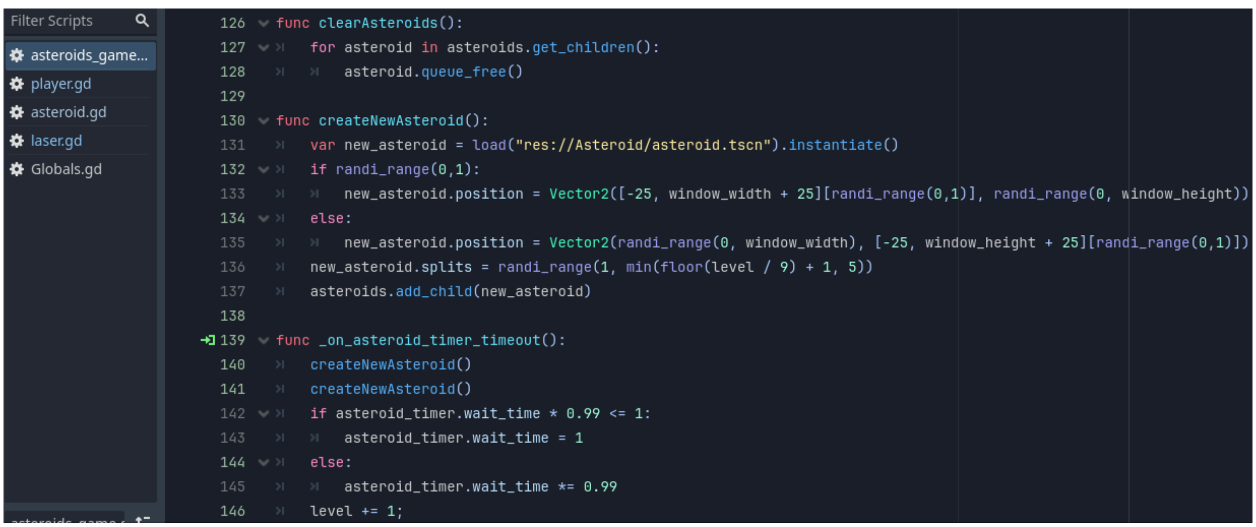

```
func playerIsDamaged():
	lives -= 1
	if lives < 0:
		gameOver()
		state = "game_over"
	else:
		updateLives()
		player.velocity *= -.05

func gameOver():
	game_over_text.text = "GAME OVER"
	score_text.text = "Score: " + str(Globals.score)
	high_score_text.text = "High Score: " + str(Globals.high_score)
	toggleUIVisibility()
	resetPlayerPosition()

func toggleUIVisibility():
	game_over_labels.visible = !game_over_labels.visible
	score_labels.visible = !score_labels.visible

func runStateMachine():
	if Input.is_action_just_pressed("accept"):
		if state == "start":
			state = "running"
			Globals.score = 0
			Globals.life_talley = 0
			initPlayer()
			asteroid_timer.start()
			toggleUIVisibility()
		elif state == "game_over":
			get_tree().change_scene_to_file("res://Asteroid Game/asteroids_game.tscn")
	if state == "running":
		checkPosition(player, 0)
```

Continuation of the asteroids_game.gd script. In this section of the main script, there are four functions to be seen: playerIsDamaged(), gameOver(), toggleUIVisibility(), and runStateMachine(). The playerIsDamaged() function is run each time the player collides with an asteroid. In this function the health of the player, which is stored in the main script as a variable, is decremented by one, followed by a logical expression that ends the game using the gameOver() function if the player's lives falls below zero and calls updateLives() (see figure B2.2) followed by a halt in the player's velocity. The gameOver() function ceases gameplay by modifying the text of multiple labels, calling toggleUIVisibility(), and calling resetPlayerPosition(). This shows the text "GAME OVER" to the player and displays the score and high score. The toggleUIVisibility() function is a simple flip for the visibility of UI elements by changing their visibility to the opposite of what they were previously. The runStateMachine() function displays two different flows of logic: one occurs when the player has pressed the "ENTER" key and another which is run unconditionally. Provided the appropriate player input, the "start" state will switch to the "running" state, followed by a reset of globals, a call to initPlayer() (see figure B2.4), a timer initialization to produce asteroids, and a call to toggleUIVisibility(). Providing the same input in the "game_over" state results in a full reset of the game scene, with the only carry-over being the high score value stored in globals. Irrelevant of player input, if the game is in the "running" state, the function checkPosition() (see figure B2.2) will be called with the player as an argument. This ensures that the player never leaves the screen space while the game is running.

Figure B2.4: asteroids_game.gd 4

Continuation of the asteroids_game.gd script. In this section of the main script, there are two functions for managing the player object: resetPlayerPosition() and initPlayer(). These functions contrast one another, as their functionalities are opposite to one another, though they also depend on one another. The resetPlayerPosition() function makes the player invisible, brings the player to the center of the screen and pauses the scene. This resets the player's position, and primes the scene to be started. The initPlayer() function is built to work off of this setup, as it unpauses the scene, makes the player visible, and brings the player to the center of the screen. The first two of these actions are opposite to that of the resetPlayerPosition() function, while the third action is shared between the two of them. Additionally, the initPlayer() scene also calls upon the player's initInvulnerability() function (see figure B2.8) to start the player in a state in which they cannot be harmed. After this, the player's lives is set to three and updateLives() (see figure B2.2) is called upon to visually represent the player's lives.

Figure B2.5: asteroids_game.gd 5




```
func clearAsteroids():
	for asteroid in asteroids.get_children():
		asteroid.queue_free()

func createNewAsteroid():
	var new_asteroid = load("res://Asteroid/asteroid.tscn").instantiate()
	if randi_range(0,1):
		new_asteroid.position = Vector2([-25, window_width + 25][randi_range(0,1)], randi_range(0, window_height))
	else:
		new_asteroid.position = Vector2(randi_range(0, window_width), [-25, window_height + 25][randi_range(0,1)])
	new_asteroid.splits = randi_range(1, min(floor(level / 9) + 1, 5))
	asteroids.add_child(new_asteroid)

func _on_asteroid_timer_timeout():
	createNewAsteroid()
	createNewAsteroid()
	if asteroid_timer.wait_time * 0.99 <= 1:
		asteroid_timer.wait_time = 1
	else:
		asteroid_timer.wait_time *= 0.99
	level += 1;
```

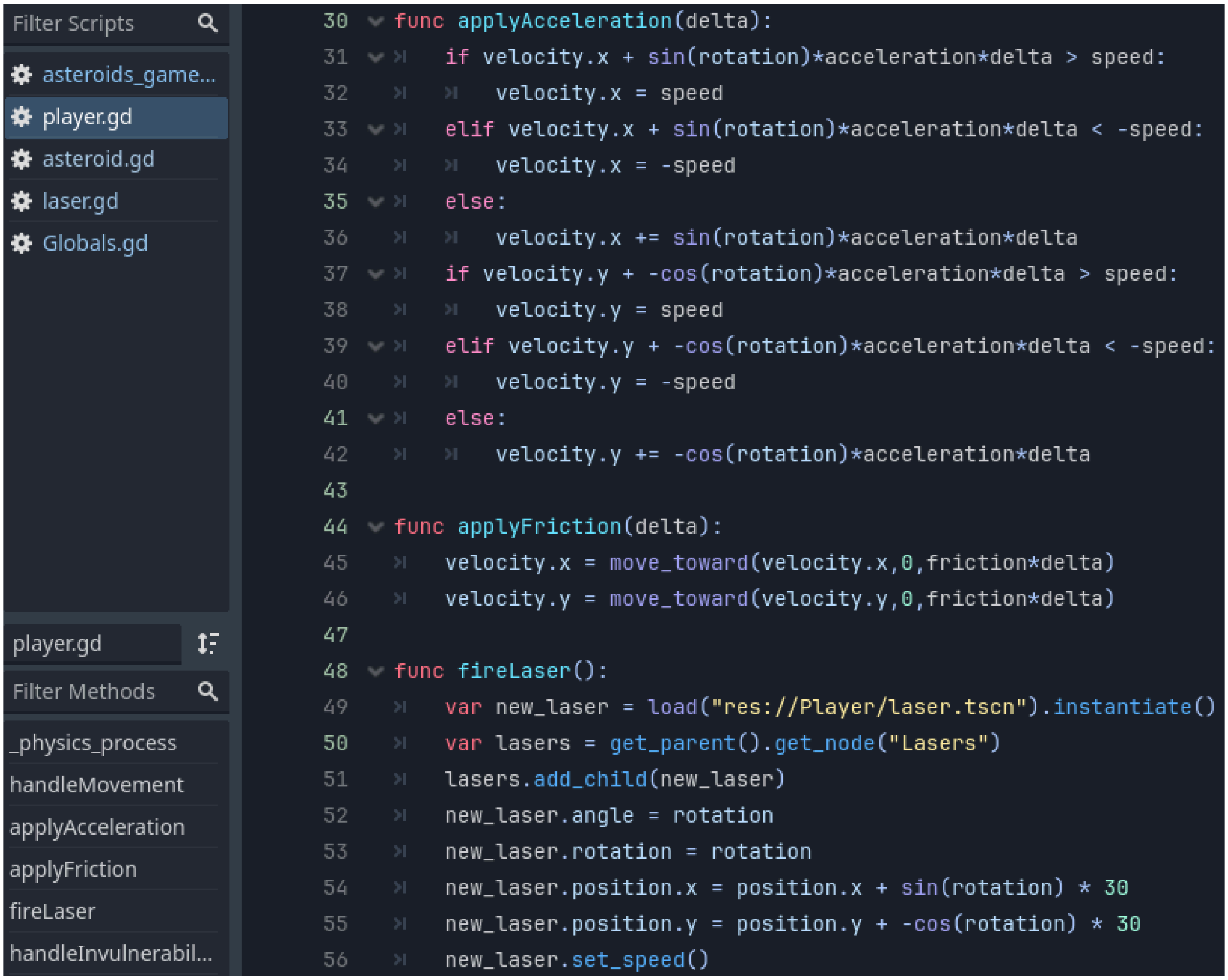
Continuation of the asteroids_game.gd script. Here is what remains of the main script, containing three functions for managing the asteroids of the game. These functions are: clearAteroids(), createNewAsteroid(), and _on_asteroid_timer_timeout(). The clearAsteroid() function is the most simple of the three, designed to iterate through all asteroids present and free them from the scene. The createNewAsteroid() function contains a greater amount of logic, but has a simple premise, which is to generate one asteroid at the perimeter of the screen in a randomized location. One detail of note is that the number of times the asteroid is meant to split is randomized between one and five based on the level of the game. The last function, _on_asteroid_timer_timeout(), is called upon solely when the asteroid timer reaches zero. Upon reaching this point, the function creates two new asteroids using createNewAsteroid(), the wait time between asteroids spawning is decreased, and the level is incremented by one.

Figure B2.6: player.gd 1

```
extends CharacterBody2D

@export var speed = 250
@onready var init_speed = speed
@export var acceleration = 400
@onready var init_acceleration = acceleration
@onready var friction = 100
@onready var init_friction = friction

@onready var area_2d = $Area2D
@onready var invulnerability_timer = $Timers/Invulnerability

signal receiveDamage

func _physics_process(delta):
	handleMovement(delta)
	if Input.is_action_just_pressed("fire"):
		fireLaser()
	handleInvulnerability()
	move_and_slide()

func handleMovement(delta):
	var turn = Input.get_axis("left","right")
	if Input.is_action_pressed("propulsion"):
		applyAcceleration(delta)
	applyFriction(delta)
	if turn:
		rotate(PI*turn*delta)
```

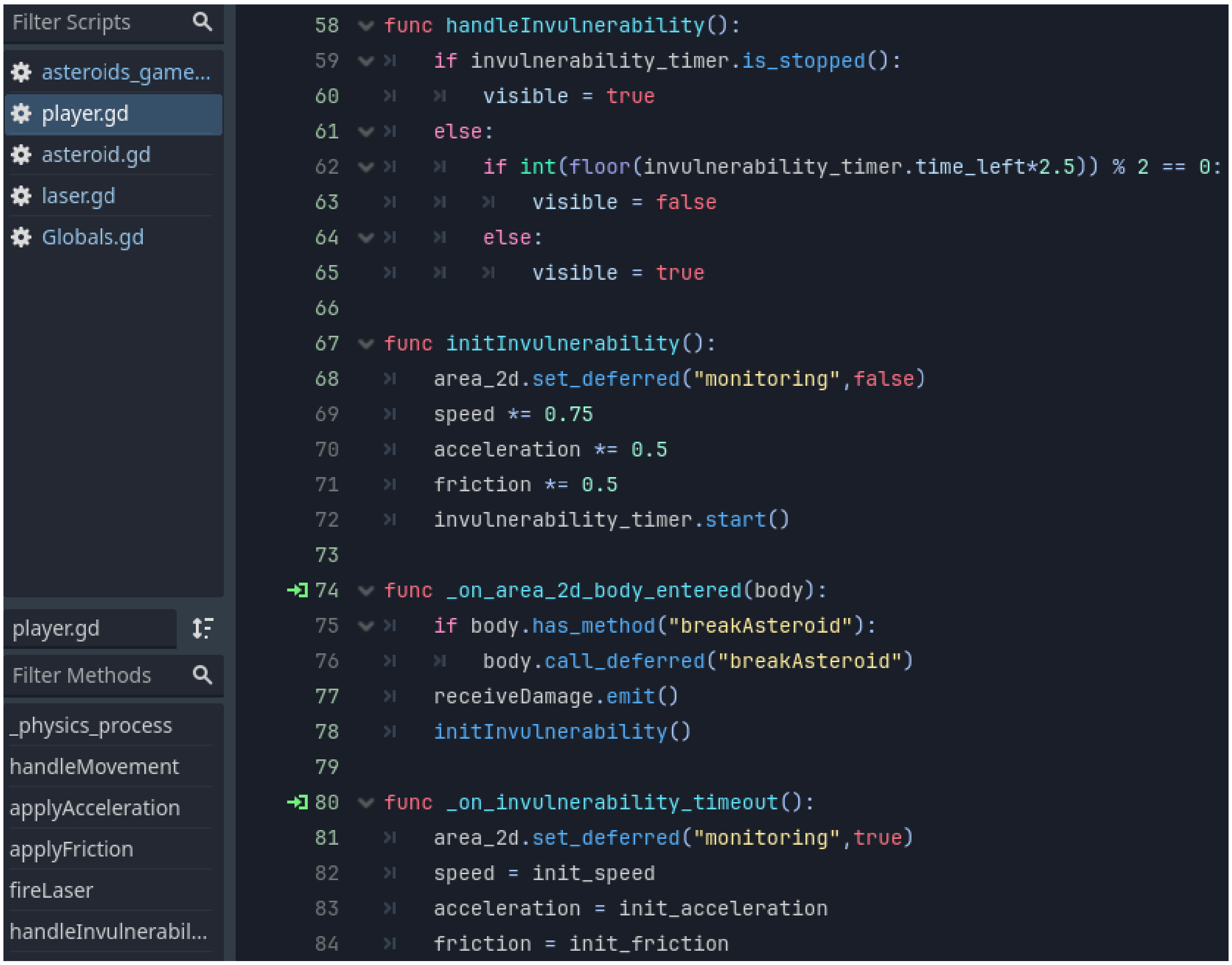
Beginning of the player.gd script. Here the variables used in the player scene alongside the processes run each frame and the handleMovement() function can be seen. As far as variables go, the player script has some variables of note that are valuable during calculations for movement of the player. These variables include the speed, acceleration, and friction values, as well as the initial versions of these values. Notably, there is also a receiveDamage signal which is used later on for the purpose of sending a message up to the main script about the player colliding with an asteroid. Each frame _physics_process(delta) is called, the handleMovement(), handleInvulnerability() (see figure B2.8), and move_and_slide() functions are called. Additionally, provided that the player presses the “SPACE/LEFT MOUSE BUTTON” key, the fireLaser() function is called (see figure B2.7). These function calls are what gives the player mobility, the ability to shoot, and a brief period of invulnerability upon colliding with an asteroid. The handleMovement() function takes delta as a parameter in order to ensure consistent gameplay on the basis of time rather than framerate. For the purpose of determining whether the player should rotate, an axis is generated based on player input of the “A” and “D” keys. The acceleration of the player is handled by checking for whether the player is holding the“W” key; provided that input, the function applyAcceleration() is run. Irrelevant of player input, the function applyFriction() is also run to slowly bring the player to a stop (see figure B2.7). Lastly, provided the axis that is already made from the player’s input, the player character is rotated by that axis at a rate of half a rotation per second.

Figure B2.7: player.gd 2



Continuation of the player.gd script, containing the applyAcceleration(), applyFriction(), and fireLaser() functions. The applyAcceleration() function takes delta as a parameter and accelerates the player forward in the direction they are facing. This acceleration is applied until the player reaches maximum speed in any given direction, at which point they maintain that speed. The applyFriction() function is responsible for bringing the player's speed to zero. This force acts upon the player at all times, but does not exceed the acceleration force of applyAcceleration(), allowing the player to propel themselves forward. The fireLaser() function is responsible for instantiating a new laser scene. It adds these lasers to a node in the main scene and creates the lasers facing the same direction of the player, places them a short distance in front of the player, and initializes their speed vector using set_speed().

Figure B2.8: player.gd 3



Final portion of the player.gd script, this section contains two functions for managing the invulnerability state of the player and two functions attached to a timer signal. These functions include handleInvulnerability(), initInvulnerability(), _on_area_2d_body_entered(), and _on_invulnerability_timeout(). The handleInvulnerability() function is responsible for managing the visual representation of the player's invulnerability state, maintaining the player's visibility when they are not invulnerable and flipping the player between being visible and invisible when they are invulnerable. The initVulnerability() function performs multiple operations to precede the invulnerability state appropriately. This function stops the player from detecting other objects, reduces the player's speed, acceleration, and friction, and starts the invulnerability timer. Upon colliding with an object, the _on_area_2d_body_entered() function is run with the body of the object colliding with the player taken as a parameter. This function calls the function breakAsteroid() on the colliding object if that object contains the method breakAsteroid(), with asteroids being the desired subject. After this, the player emits the receiveDamage signal and the initInvulnerability() function is run. After the invulnerability timer runs out, the _on_invulnerability_timeout() function is run in order to cease the invulnerability state of the player. The function does this by reenabling the detection of other objects and returning the player's speed, acceleration, and friction to their initial values.

Figure B2.9: asteroid.gd 1

```
extends RigidBody2D
class_name Asteroid

var size = 5
var splits = 1
@onready var speed : Vector2

func _ready():
	initMovement()

func _physics_process(delta):
	move_and_collide(speed*delta)

func breakAsteroid():
	if splits:
		var new_asteroid_1 = load("res://Asteroid/asteroid.tscn").instantiate()
		get_parent().add_child(new_asteroid_1)
		new_asteroid_1.size = size - 1
		new_asteroid_1.splits = splits - 1
		new_asteroid_1.position = position
		new_asteroid_1.initMovement()
		if randi_range(0,1):
			var new_asteroid_2 = load("res://Asteroid/asteroid.tscn").instantiate()
			get_parent().add_child(new_asteroid_2)
			new_asteroid_2.size = size - 1
			new_asteroid_2.splits = splits - 1
			new_asteroid_2.position = position
			new_asteroid_2.initMovement()
	queue_free()
```

Beginning of the asteroid.gd script. Here some of the variables involved with the asteroid scene, the _ready(), _physics_process(), and the breakAsteroid() functions can be seen. The asteroid script has two functions to run overall, one being the function shown here and the other being the initMovement() function (see figure B2.10). The initMovement() function is the sole operation called in the _ready() function. As for _physics_process(), only move_and_colide() is run in order to move the asteroid every frame. The breakAsteroid() function is not run by the asteroid script itself, but rather by other objects such as the player when they are struck by an asteroid (see figure B2.8). This function acts to instantiate one to two asteroids at random, that are a size smaller and have one fewer splits than the current asteroid. After this happens, the asteroid is freed, eliminating it from the scene.

Figure B2.10: asteroid.gd 2

Here the second function of the asteroid.gd script can be seen. The initMovement() function modifies two properties of the asteroid object on the basis of one of the asteroid's properties. The property that initAsteroid() depends on for its logic is the size property, which ranges from one to five. Based upon this, the speed and scale of the asteroid are adjusted accordingly. The speed of the asteroid in any given direction can be as high as ~141 for the largest asteroid and as high as

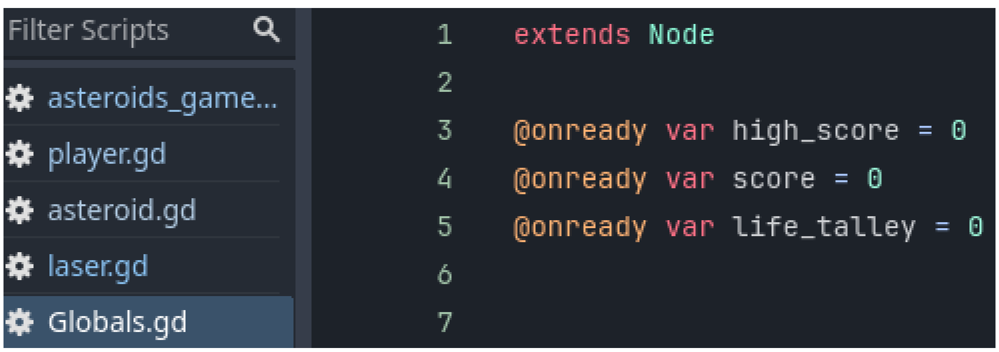
~707 in any given direction for the smallest asteroid. Meanwhile the scale of the asteroid decreases by 0.2 for every size under five.

Figure B2.11: laser.gd

```
extends RigidBody2D

var angle : float
var speed : Vector2

func _physics_process(delta):
	move_and_collide(speed*delta)

func set_speed():
	speed.x += sin(angle) * 400
	speed.y += -cos(angle) * 400

func _on_area_2d_body_entered(body):
	if body.has_method("breakAsteroid") and body.size <= 0:
		Globals.score += 1000 / (5)
		Globals.life_talley += 0.2
		body.call_deferred("breakAsteroid")
	elif body.has_method("breakAsteroid"):
		Globals.score += 1000 / (body.size * 5)
		Globals.life_talley += 1.0 / (body.size * 5)
		body.call_deferred("breakAsteroid")
	queue_free()

func _on_clear_timer_timeout():
	queue_free()
```

Here the laser.gd script in its entirety can be seen. This script is simple due to the nature of the laser scene, which has the job of moving in a straight line. It does this using the same means as the asteroid scene (see figure B2.9): by feeding its speed and delta into move_and_collide(). This operation takes place in _physics_process() and is the only operation that takes place each frame. It is also notable that this speed is set in the first place using set_speed() during the instantiation of the laser object. The laser scene meets its end through two possible means: collision with an object resulting in a call of _on_area_2d_body_entered() or timeout of the clear timer resulting in a call of _on_clear_timer_timeout(). The latter of these two functions has the sole purpose of freeing the laser object and nothing more, meanwhile the former of these two, the _on_area_2d_body_entered() function, performs a few operations before freeing the laser. Upon colliding with an asteroid, the laser increments the score according to the size of the asteroid and increments the life_talley, a variable responsible for keeping track of giving the player extra lives, before calling breakAsteroid() on the colliding asteroid.

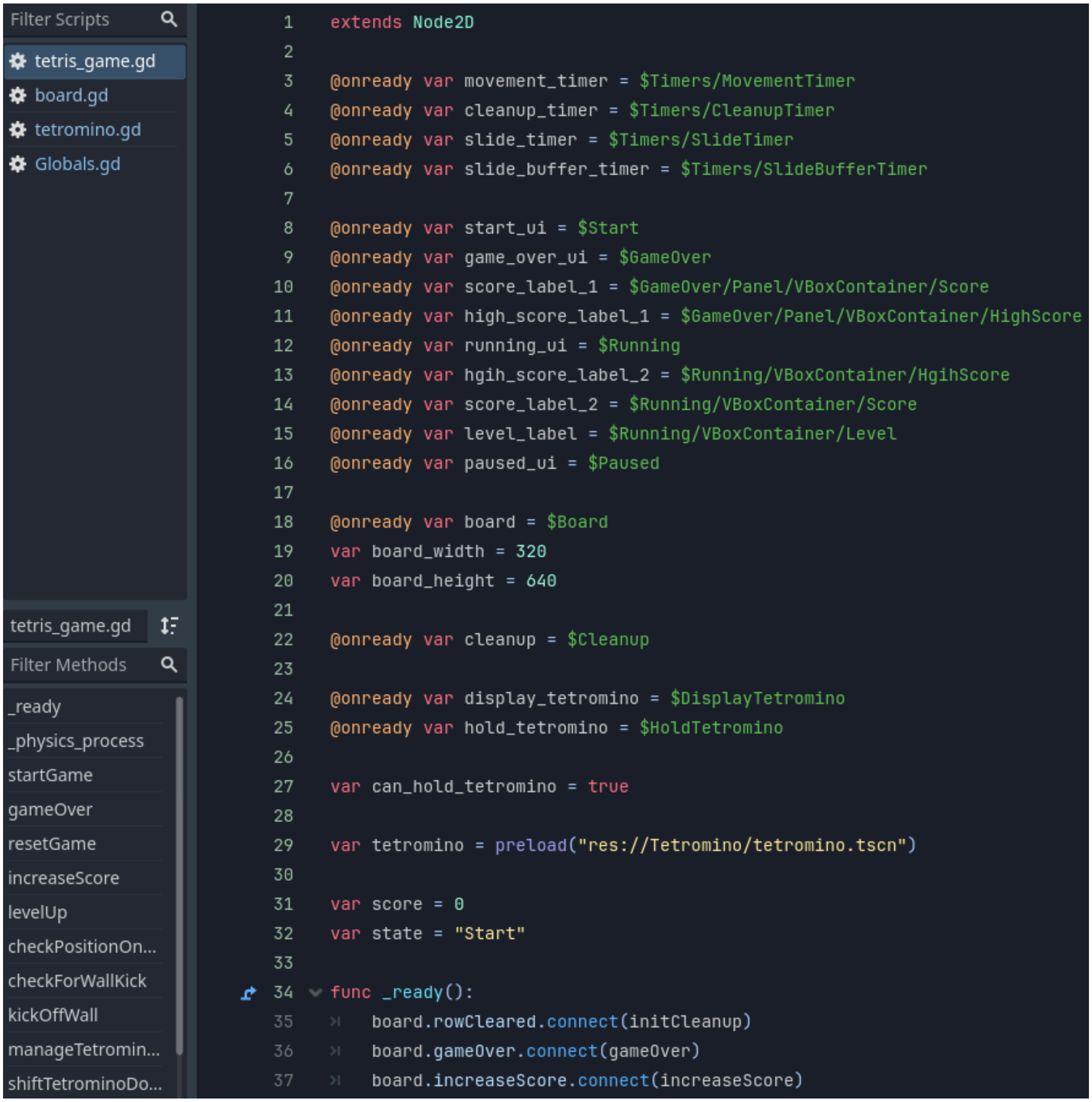
Figure B2.12: Globals.gd



Here are the globals used in the *Asteroids* game. The high_score is maintained between games and changes only when a score higher than it has been achieved. The score keeps track of the points gained by the player in the current run of the game. The life_talley keeps track of the points gained and rewards the player with an extra life at a set increment of points.

B3 Tetris

Figure B3.1: tetris_game.gd 1



This is the beginning of the tetris_game.gd script, the main script of the game *Tetris*. In this part of the script, all of the variables involved and initial setup in the _ready() function can be seen. Some values of note are as follows: the board_width and board_height variables which are set to 320 and 640 respectively. These directly translate to a grid of width 10 and height 20, provided a division of 32. The can_hold_tetromino variable which is a boolean value set to true each time a tetromino has been placed and false each time a tetromino is held. The score variable which is initialized at zero and added upon during gameplay. Lastly, there is the state variable which is initialized to "Start". In the _ready() function, three signals are connected between the board and the main script. These three signals are rowCleared, gameOver, and increaseScore (see figure B3.11) and are respectively connected to functions in the main script initCleanup, gameOver, and increaseScore (see figures B3.2, B3.3, & B3.9).

Figure B3.2: tetris_game.gd 2

```
func _physics_process(delta):
    if state == "Start" and Input.is_action_just_pressed("accept"):
        startGame()
    if state == "GameOver" and Input.is_action_just_pressed("accept"):
        resetGame()
    if state == "Running" and has_node("Tetromino"):
        if Input.is_action_just_pressed("pause"):
            get_tree().paused = !get_tree().paused
            paused_ui.visible = !paused_ui.visible
        if !get_tree().paused:
            manageTetrominoMovement()
            if Input.is_action_just_pressed("hold") and can_hold_tetromino:
                addTetrominoToHold()

func startGame():
    movement_timer.wait_time = 1
    Globals.level = 1
    level_label.text = "Level: " + str(Globals.level)
    movement_timer.start()
    createNewTetromino()
    state = "Running"
    hgih_score_label_2.text = "High Score: " + str(Globals.high_score)
    score_label_2.text = "Score: " + str(score)
    start_ui.visible = false
    running_ui.visible = true

func gameOver():
    state = "GameOver"
    movement_timer.stop()
    if score > Globals.high_score:
        Globals.high_score = score
    score_label_1.text = "Your score is " + str(score)
    high_score_label_1.text = "The high score is " + str(Globals.high_score)
    running_ui.visible = false
    game_over_ui.visible = true
```

Continuation of the tetris_game.gd script, here the _physics_process(), startGame(), and gameOver() functions can be seen. In every frame, on the basis of both game state and user input, different operations are performed in _physics_process(). If the game is in the “Start” state and the player presses the “ENTER” key, the startGame() function is run. Likewise, if the game is in the “GameOver” state and the player presses “ENTER”, the resetGame() function is run (see figure B3.3). Alternatively, so long as there is an active tetromino and the game is in the

"Running" state, the game runs one string of operations at all times and another solely while the game is unpaused. At any and all times during the running state the player may press the "P" key to pause and unpause the game.

Figure B3.3: tetris_game.gd 3

Continuation of the tetris_game.gd script, in this section of code the resetGame(), increaseScore(), and levelUp() functions can be seen. The resetGame() function returns the game back to its initial state, with its first operation being to change the game's state to "Start". The clearBoard() method of the board scene (see figure B3.13) is then called in order to turn the board into a blank slate. All tetrominoes present in the scene are freed, first the active tetromino, then the display tetromino, and last the held tetromino. The score is set to zero, the level is set to 1, and the starting UI is made visible. The increaseScore() function takes in one parameter, which is the amount the score is increasing by. The base functionality of this is to increment the score by the amount provided to the function, but following this act a few additional operations are made. The score label is updated with the new score after the increase and the same occurs for the high score label if the score exceeds the former high score. Upon the increase in score, should a set threshold be reached, the levelUp() function is called. This levelUp() function is responsible for increasing the difficulty of the game as the player progresses. Upon being called, this function first increments the level by one. According to the level of the game, the wait time of the movement timer is reduced by an increment of five percent of its initial magnitude. This occurs every level for the first ten levels gained before this reduction occurs every other level. This will occur until the wait time reaches the minimum threshold of 0.05 seconds and the difficulty ceases scaling. With each call of the function, the level label is updated to match the level of the game.

Figure B3.4: tetris_game.gd 4


```
func checkPositionOnMove(direction):
	for square in get_node("Tetromino").get_children():
		if direction == "right":
			if floor(get_node("Tetromino").rotation) == 1:
				if get_node("Tetromino").position.x - square.position.y + 32 > board_width:
					return false
			elif floor(get_node("Tetromino").rotation) == 3:
				if get_node("Tetromino").position.x - square.position.x + 32 > board_width:
					return false
			elif floor(get_node("Tetromino").rotation) == 4:
				if get_node("Tetromino").position.x + square.position.y + 32 > board_width:
					return false
			else:
				if get_node("Tetromino").position.x + square.position.x + 32 > board_width:
					return false
		else:
			if floor(get_node("Tetromino").rotation) == 1:
				if get_node("Tetromino").position.x - square.position.y - 32 < 0:
					return false
			elif floor(get_node("Tetromino").rotation) == 3:
				if get_node("Tetromino").position.x - square.position.x - 32 < 0:
					return false
			elif floor(get_node("Tetromino").rotation) == 4:
				if get_node("Tetromino").position.x + square.position.y - 32 < 0:
					return false
			else:
				if get_node("Tetromino").position.x + square.position.x - 32 < 0:
					return false
	return true
```

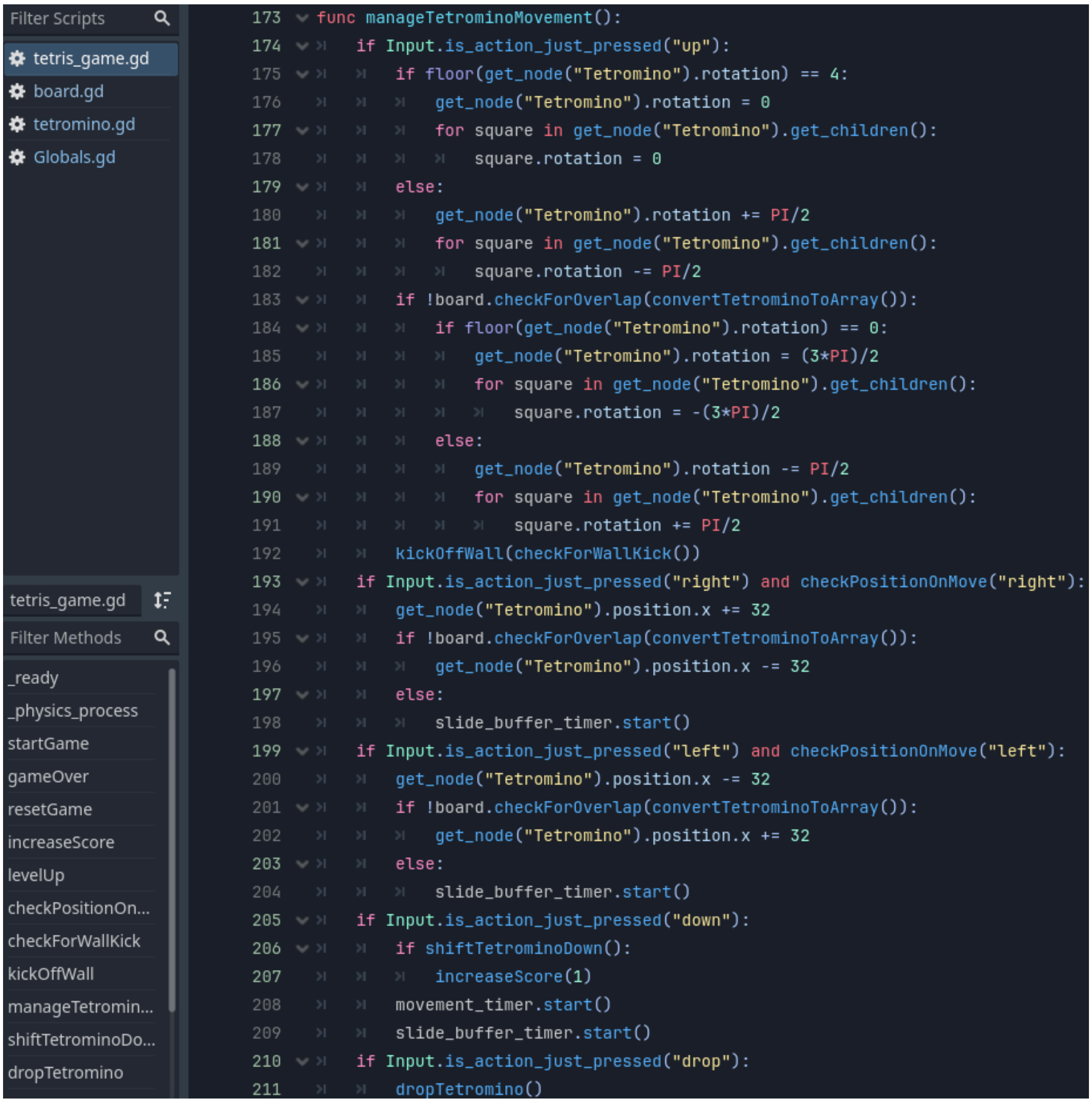
Continuation of the tetris_game.gd script, here the checkPositionOnMove() function can be seen. In terms of regulating player movement, this is one of the most important functions for constraining the player to the limitations of the board. This function takes in one parameter "direction" which is responsible for directing the logical structure that is run. The particular movement that this function constraints is the movement of the player left and right. It constrains the movement of each tetromino by iterating over the squares of the tetromino. Each tetromino has four squares of different offset from the origin of the tetromino. The rotation state of the tetromino and the relative position of the tetromino's squares from its origin are used to inform the game whether the tetromino would exceed the bounds of the board by moving either left or right. A shift to the right is checked by adding one square of distance to each square and comparing it with the width of the board, while a shift to the left is checked by subtracting one square of distance to each square and comparing it to zero, the leftmost part of the board.

Figure B3.5: tetris_game.gd 5

```
#Checks position and rotation of tetromino and returns a direction if the tetromino goes beyond the scope of the board
func checkForWallKick():
	for square in get_node("Tetromino").get_children():
		if floor(get_node("Tetromino").rotation) == 1:
			if get_node("Tetromino").position.x - square.position.y > board_width:
				return 'left'
			elif get_node("Tetromino").position.x - square.position.y < 0:
				return 'right'
		elif floor(get_node("Tetromino").rotation) == 3:
			if get_node("Tetromino").position.x - square.position.x > board_width:
				return 'left'
			elif get_node("Tetromino").position.x - square.position.x < 0:
				return 'right'
		elif floor(get_node("Tetromino").rotation) == 4:
			if get_node("Tetromino").position.x + square.position.y > board_width:
				return 'left'
			elif get_node("Tetromino").position.x + square.position.y < 0:
				return 'right'
		else:
			if get_node("Tetromino").position.x + square.position.x > board_width:
				return 'left'
			elif get_node("Tetromino").position.x + square.position.x < 0:
				return 'right'
	return ''

#Shifts the tetromino left or right according to the direction provided
#Recurs itself if a direction is provided
func kickOffWall(direction):
	if direction == 'left':
		get_node("Tetromino").position.x -= 32
	elif direction == 'right':
		get_node("Tetromino").position.x += 32
	if direction != '':
		kickOffWall(checkForWallKick())
```

Continuation of the tetris_game.gd script. The two functions shown here, checkForWallKick() and kickOffWall(), directly work with one another to “kick” the tetromino off of the side of the board upon rotating. The checkForWallKick() function iterates through the squares of the active tetromino upon rotating. The purpose of this function is to search for whether the tetromino exceeds the boundaries of the board in either direction. If the tetromino exceeds these boundaries, the string “left” or “right” is returned as the direction the tetromino needs to be moved. If the tetromino does not exceed any boundaries, an empty string is returned instead. The purpose of these returns is solely to be fed into the kickOffWall() function, which takes in a string for the direction of movement and moves the tetromino left or right by one square accordingly. This function also will recursively call itself with checkForWallKick() as an argument, provided that it has not received an empty string. This ensures that the function will move the tetromino until it is within the boundaries of the board.

Figure B3.6: tetris_game.gd 6



Continuation of the tetris_game.gd script. Here is the manageTetrominoMovement() script which runs a series of logical expressions checking for player input and performing operations on the active tetromino accordingly. Provided that the player presses the “W” key, the function performs a rotation operation. This rotates the tetromino around its origin by 90 degrees and the squares of the tetromino by 90 degrees in the opposite direction, causing them to stay upright. The tetromino’s new position after rotation is then checked using a combination of the checkForOverlap() function of the board script (see figure B3.12) and the convertTetrominoToArray() function from the main script (see figure B3.9). If there is overlap detected in this operation, the rotation is undone. After this check, the kickOffWall() function is called with checkForWallKick() as an argument (see figure B3.5). Input from the player pressing the “A” or “D” key is checked in order to move the player left or right, respectively. Alongside this check, the checkPositionOnMove() function (see figure B3.4) is called in order to determine whether the shift would move the tetromino outside of the board space. With this movement, the same check that was used for rotation overlap is used again. If the player presses the “S” key, the tetromino is shifted down using the shiftTetrominoDown() function (see figure B3.7). If this shift succeeds, the player is awarded an additional point for accelerating gameplay. The movement and slide buffer timers are also reset upon shifting the tetromino down. Lastly, if the player is to press the “SPACE” key, the dropTetromino() function is called (see figure B3.7).

Figure B3.7: tetris_game.gd 7

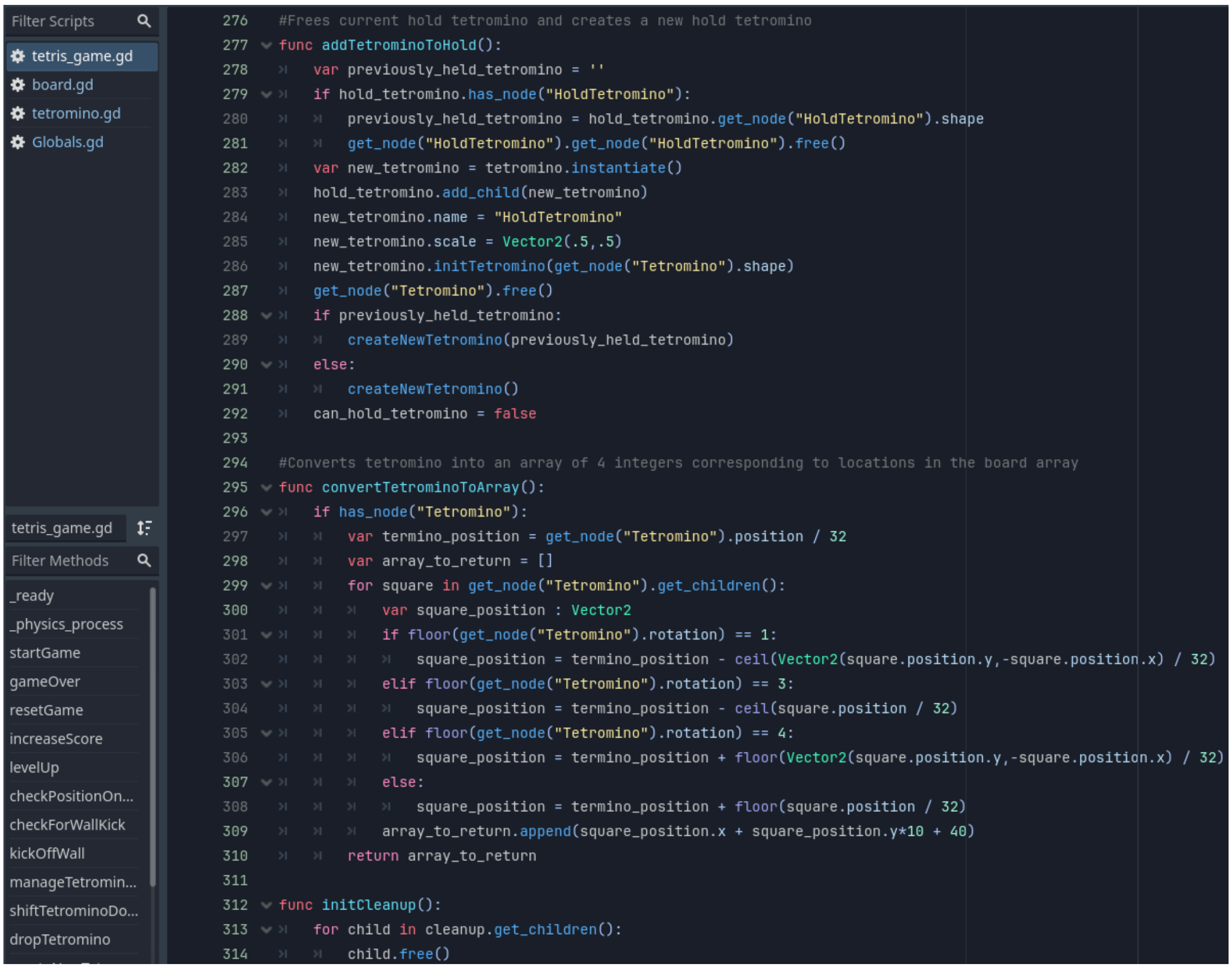

```
#Returns true if tetromino can still shift down, false if tetromino cannot shift down
func shiftTetrominoDown():
	if has_node("Tetromino") and state == "Running":
		var tetromino_can_move = true
		var tetromino_locked = false
		for i in convertTetrominoToArray():
			var last_row = [230,231,232,233,234,235,236,237,238,239]
			if last_row.has(int(i)) and !tetromino_locked:
				board.lockTetrominoToBoard(convertTetrominoToArray(),get_node("Tetromino").shape)
				createNewTetromino()
				tetromino_can_move = false
				tetromino_locked = true
				return false
		if tetromino_can_move:
			get_node("Tetromino").position.y += 32
			if !board.checkForOverlap(convertTetrominoToArray()):
				get_node("Tetromino").position.y -= 32
				board.lockTetrominoToBoard(convertTetrominoToArray(),get_node("Tetromino").shape)
				createNewTetromino()
				return false
	return true

#Recurs shiftTetrominoDown() to bring the tetromino to its lowest possible position
func dropTetromino():
	var can_shift_down = true
	var times_shifted = 0
	while can_shift_down:
		can_shift_down = shiftTetrominoDown()
		if can_shift_down:
			times_shifted += 2
	increaseScore(times_shifted)
```

Continuation of the tetris_game.gd script. The following two functions are responsible for the primary driver of the gameplay of Tetris: the shiftTetrominoDown() and dropTetromino() functions. The shiftTetrominoDown() function first checks whether any one square of the tetromino will exceed the boundaries of the board. If the tetromino would do so, it is then locked to the board using the lockTetrominoToBoard() function (see figure B3.11) and a new active tetromino is created in its place using the createNewTetromino() function (see figure B3.8). If the tetromino does not exceed the boundaries of the board upon moving, it is moved downward by one space and is checked for whether it overlaps with any squares occupied on the grid. If there is overlap, the movement is undone, the teromino is locked to the board, and a new tetromino is created. If there are no obstacles upon moving down by one space, the tetromino continues to move down one space at a time due to action of either the timer or the player. The dropTetromino() function acts as a utility function to snap the tetromino to the lowest position it can reach on the board from the space it occupies. It does this by recursively calling shiftTetrominoDown() and awarding two points each time it succeeds.

Figure B3.8: tetris_game.gd 8

Continuation of the tetris_game.gd script. Here the functions responsible for the generation of a new tetromino. These functions are createNewTetromino() and addTetrominoToDisplay(). The createNewTetromino() function first takes the active tetromino, renames it, and reparents it to the cleanup node while also starting the cleanup timer to dispose of it. A new tetromino is added to the top middle of the board, in-line with the grid, with a type that is either provided for it through the parameter tetromino_type, given to it by the display tetromino, or created at random due to a lack of input and display tetromino. After this new tetromino is created, a new tetromino is added to the display using addTetrominoToDisplay(). This function creates a tetromino of a random type and places it above the top middle of the board. This tetromino is immobile and serves the purpose of displaying the next piece to the user and storing its type data for retrieval when the next tetromino is created.

Figure B3.9: tetris_game.gd 9




```
#Frees current hold tetromino and creates a new hold tetromino
func addTetrominoToHold():
	var previously_held_tetromino = ''
	if hold_tetromino.has_node("HoldTetromino"):
		previously_held_tetromino = hold_tetromino.get_node("HoldTetromino").shape
		get_node("HoldTetromino").get_node("HoldTetromino").free()
	var new_tetromino = tetromino.instantiate()
	hold_tetromino.add_child(new_tetromino)
	new_tetromino.name = "HoldTetromino"
	new_tetromino.scale = Vector2(.5,.5)
	new_tetromino.initTetromino(get_node("Tetromino").shape)
	get_node("Tetromino").free()
	if previously_held_tetromino:
		createNewTetromino(previously_held_tetromino)
	else:
		createNewTetromino()
	can_hold_tetromino = false

#Converts tetromino into an array of 4 integers corresponding to locations in the board array
func convertTetrominoToArray():
	if has_node("Tetromino"):
		var termino_position = get_node("Tetromino").position / 32
		var array_to_return = []
		for square in get_node("Tetromino").get_children():
			var square_position : Vector2
			if floor(get_node("Tetromino").rotation) == 1:
				square_position = termino_position - ceil(Vector2(square.position.y,-square.position.x) / 32)
			elif floor(get_node("Tetromino").rotation) == 3:
				square_position = termino_position - ceil(square.position / 32)
			elif floor(get_node("Tetromino").rotation) == 4:
				square_position = termino_position + floor(Vector2(square.position.y,-square.position.x) / 32)
			else:
				square_position = termino_position + floor(square.position / 32)
			array_to_return.append(square_position.x + square_position.y*10 + 40)
		return array_to_return

func initCleanup():
	for child in cleanup.get_children():
		child.free()
```

Continuation of the tetris_game.gd script. In this portion of the script the addTetrominoToHold(), convertTetrominoToArray(), and initCleanup() functions can be seen. The addTetrominoToHold() function is responsible for adding the active tetromino to the hold space while taking the inactive tetromino in the hold space to make it into the active tetromino, if there is a held tetromino. The function starts by instantiating an empty string variable and changing it to the type of tetromino held in the hold space, if a tetromino is held. A new held tetromino is instantiated using data of the active tetromino, the active tetromino is freed, and a new tetromino is generated on the basis of the string variable created. If this variable has a tetromino type, that type of tetromino is created. Otherwise, a tetromino is created based on the tetromino is the display tetromino space based on the logic of the createNewTetromino() function (see figure B3.8). The convertTetrominoToArray() function is the most important function for allowing the game to communicate information about the active tetromino and the state of the board. This function takes the tetromino and, based upon its position and rotation, finds the position of each square. The position of each square is converted into an index that corresponds to the array of length 240 of the board script (see figure B3.11). The initCleanup() function serves the simple purpose of removing all children of the cleanup node by iterating through every child and freeing them.

Figure B3.10: tetris_game.gd 10

Filter Scripts
tetris_game.gd
board.gd
tetromino.gd
Globals.gd
tetris_game.gd

```
func _on_movement_timer_timeout():
	if !get_tree().paused:
		shiftTetrominoDown()

func _on_cleanup_timer_timeout():
	if !get_tree().paused:
		initCleanup()

func _on_slide_timer_timeout():
	if !slide_buffer_timer.time_left and !get_tree().paused:
		if Input.is_action_pressed("right") and checkPositionOnMove("right"):
			get_node("Tetromino").position.x += 32
			if !board.checkForOverlap(convertTetrominoToArray()):
				get_node("Tetromino").position.x -= 32
		if Input.is_action_pressed("left") and checkPositionOnMove("left"):
			get_node("Tetromino").position.x -= 32
			if !board.checkForOverlap(convertTetrominoToArray()):
				get_node("Tetromino").position.x += 32
		if Input.is_action_pressed("down"):
			if shiftTetrominoDown():
				increaseScore(1)
			movement_timer.start()
```

Final section of the tetris_game.gd script Here the function calls of each timer's timeout can be seen. These functions are as follows: _on_movement_timer_timeout(), _on_cleanup_timer_timeout(), and _on_slide_timer_timeout(). The _on_movement_timer_timeout() is called to move the tetromino downwards. It does this by calling the shiftTetrominoDown() function (see figure B3.7), so long as the game is not paused. The _on_cleanup_timer_timeout() is called to begin the cleanup process. This is done by calling the initCleanup() function (see figure B3.9), again under the condition that the game is not paused. Lastly, the _on_slide_timer_timeout() function is responsible for rapidly shifting the tetromino right, left, or down based on the direction they hold. This operates only while the game is unpaused and is limited by another slide_buffer_timer. This is so that when the player taps left, right, or down the tetromino does not rapidly shift twice at once provided player input close to a cycle of the shift timer. The logic resembles that of the manageTetrominoMovement() function (see figure B3.6).

Figure B3.11: board.gd 1

```
extends Node

var board_array : Array[String] = []

signal rowCleared
signal increaseScore(amount)
signal gameOver

func _ready():
	board_array.resize(240)
	board_array.fill('')

#Inputs tetromino type into board array according to the array of positions provided
func lockTetrominoToBoard(tetromino_array,tetromino_type):
	for i in tetromino_array:
		if board_array[i]:
			gameOver.emit()
		else:
			board_array[i] = tetromino_type
	for i in range(40):
		if board_array[i]:
			gameOver.emit()
	checkForFullRow()
	updateBoard()
```

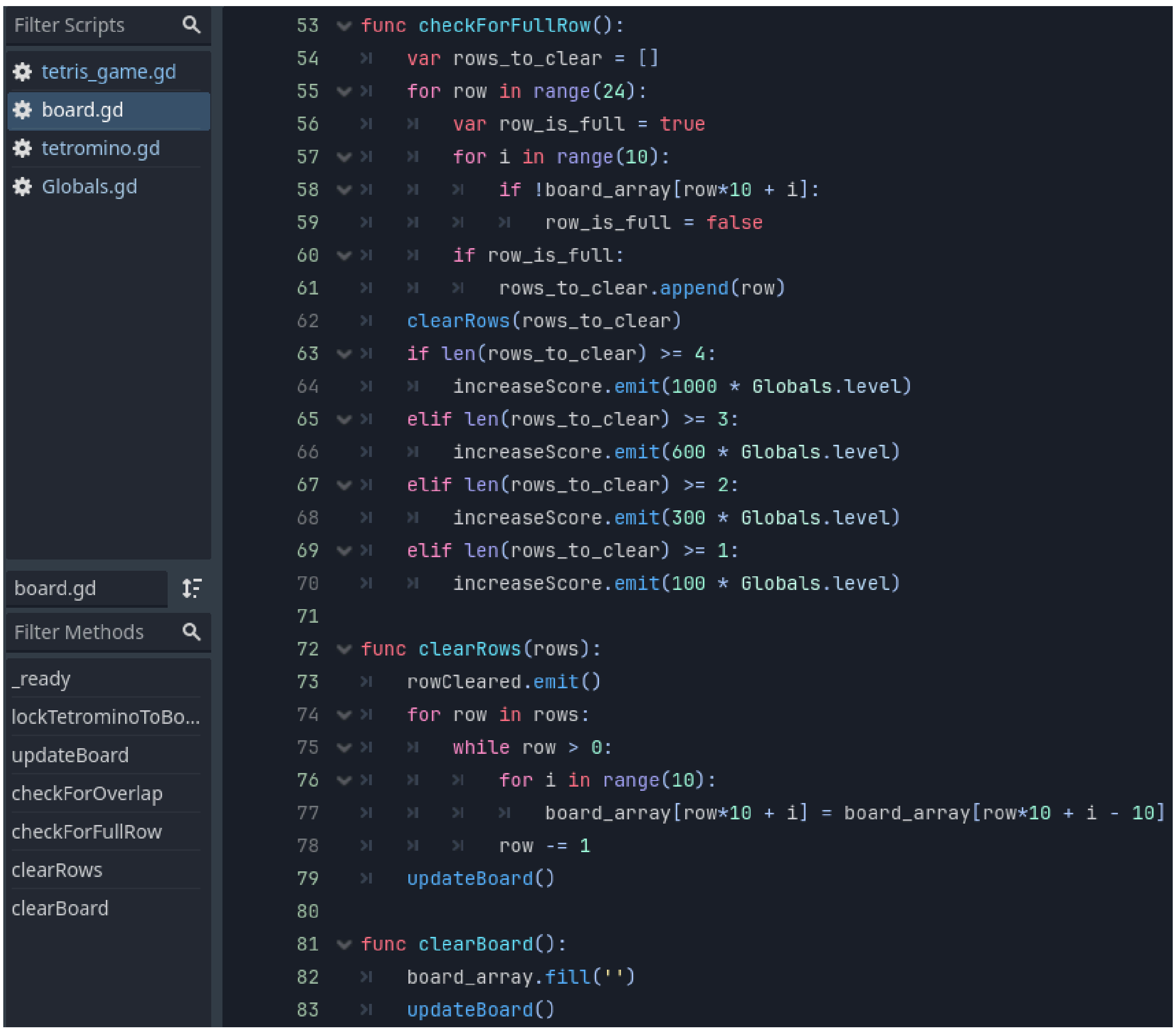
This is the beginning of the board.gd script, a script that works closely with the main script with the purpose of managing the board_array variable. Here the signals, _ready() function, and lockTetrominoToBoard() can be seen. There are three signals emitted by the board script for the main script to pick up: the rowCleared signal is emitted when a row of the board has been completed or filled. The increaseScore signal is emitted when rows are cleared and the score is to be incremented by a set amount. The gameOver signal is emitted when an operation is performed on the board that deems the game finished. In the _ready() function, the board array is initialized to a length of 240 and filled with an empty string value. The lockTetrominoToBoard() function takes in two parameters, a tetromino_array and a tetromino_type. This function is what changes the indices of the board array to have substance, by taking the type of tetromino provided and placing that type into the array indices provided. If any of those spaces are occupied or any of those spaces are among the first 40 indices of the array, then the gameOver signal is emitted, ending the game. If the space is already occupied, it indicates that the tetromino is not in a space where it can appropriately shift downward or lock to the grid, indicating it is at the top of the board at an inappropriate location. As for the first 40 spaces of the array, these indices are associated with spaces above the board that a tetromino cannot occupy. At the end of the function, the checkForFullRow() and updateBoard() functions are called (see figures B3.12 & B3.13) in order to check whether any rows have been completed and update the board graphically.

Figure B3.12: board.gd 2

```
#Displays un-displayed squares, clears empty spaces
func updateBoard():
    for square in get_children():
        var square_index = int(square.name.get_slice('Square',1))
        if !board_array[square_index]:
            square.free()
        elif board_array[square_index] != Globals.Tetromino.find_key(square_index):
            square.set_frame(Globals.Tetromino[board_array[int(square.name.get_slice('Square',1))]])
    for i in board_array.size():
        if board_array[i] and !has_node("Square" + str(i)):
            var x = i % 10
            var y = i / 10
            var new_square = load("res://Tetromino/tetromino_square.tscn").instantiate()
            add_child(new_square)
            new_square.position = Vector2(x + .5,y - 3.5) * 32
            new_square.name = "Square" + str(i)
            new_square.set_frame(Globals.Tetromino[board_array[i]])

#Provided and array of array location, checks for overlap between the array and the board array
func checkForOverlap(tetromino_array):
    for i in tetromino_array:
        if i > 239:
            return false #There is overflow through bottom of board
        elif board_array[i]:
            return false #There is overlap
    return true #There is no overlap
```

Continuation of the board.gd script, here the updateBoard() and checkForOverlap() functions can be seen. The updateBoard() function is designed to visually represent the board array as individual square objects shown on the grid. To do this, the function iterates through the existing squares on the board and removes them if they are no longer in the board array. Alternatively, if a square has a color that does not line up with the tetromino type that occupies that space on the board, the color of the square is changed to match the color of the corresponding board array space. After this is completed, the board array is iterated through and has squares added to it if any of the grid spaces are missing. The checkForOverlap() function takes in a tetromino array and compares that array to the spaces of the board. This function returns a boolean that is used by the main script in order to determine whether the tetromino should be able to perform an operation.

Figure B3.13: board.gd 3



Final section of the board.gd script, here the checkForFullRow(), clearRows(), and clearBoard() functions can be seen. The checkForFullRow() function iterates through every row in the board array and checks for whether a space is missing. If a space is missing, the row number is appended to an array to store all filled lines. After iterating through the board array in its entirety, the list of rows is passed to the clearRows() function. The increaseScore signal is then emitted with variable points based upon how many lines were completed simultaneously. The clearRows() function takes in a list of rows as a parameter and starts by emitting the rowCleared signal for the main script to pick up. Each row in rows is iterated through and every value is shifted down the array by 10 until the row to clear has been reached. This functionally eliminates all cleared rows, after which the updateBoard() function is called (see figure B3.12). The clearBoard() function serves to fully reset the board array and graphics by setting all values to an empty string and calling updateBoard().

Figure B3.14: tetromino.gd

```
Filter Scripts
tetris_game.gd
board.gd
tetromino.gd
Globals.gd
```

```
extends Node2D

var cells = Globals.cells
var tetromino = Globals.Tetromino
var shape : String

func initTetromino(tetromino_type):
	shape = tetromino_type
	for i in cells[tetromino[tetromino_type]]:
		var new_square = load("res://Tetromino/tetromino_square.tscn").instantiate()
		if i != Vector2.ZERO:
			add_child(new_square)
			new_square.position.x += i.x * 32
			new_square.position.y += i.y * 32
			new_square.name = "TetrominoSquare"
	for child in get_children():
		child.set_frame(tetromino[tetromino_type])
```

This is the tetromino.gd script, which solely has the purpose of initiating the tetromino object. This script has one function, initTetromino(), which takes a tetromino type as a parameter. Based upon the tetromino type provided and the cells and tetromino global variables (see figure B3.13), squares are added to the base tetromino object to form the desired shape. After this is done, the color of each square is changed to fit the tetromino type accordingly.

Figure B3.15: Globals.gd

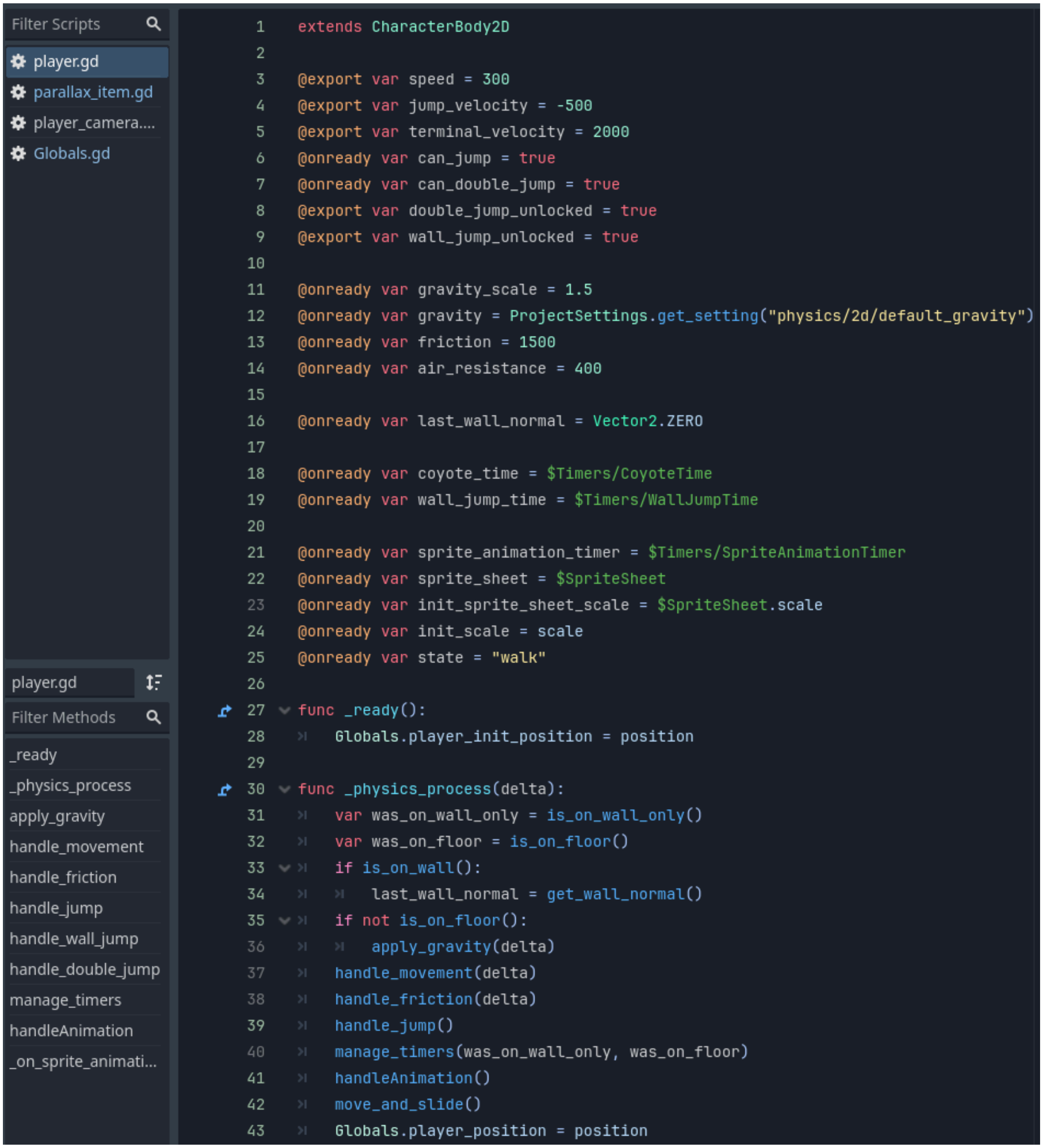
Filter Scripts
tetris_game.gd
board.gd
tetromino.gd
Globals.gd

```
extends Node

enum Tetromino {
	I,J,L,O,S,T,Z
}

var cells = {
	Tetromino.I: [Vector2(-2, 0), Vector2(-1, 0), Vector2(0, 0), Vector2(1, 0)],
	Tetromino.J: [Vector2(1,1), Vector2(-1, 0), Vector2(0,0), Vector2(1,0)],
	Tetromino.L: [Vector2(-1, 1), Vector2(-1, 0), Vector2(0,0), Vector2(1, 0 )],
	Tetromino.O: [Vector2(-1,0), Vector2(-1,1), Vector2(0,0), Vector2(0,1)],
	Tetromino.S: [Vector2(-1, 1), Vector2(0, 1), Vector2(0,0), Vector2(1, 0)],
	Tetromino.T: [Vector2(0,1), Vector2(-1, 0), Vector2(0,0), Vector2(1,0)],
	Tetromino.Z: [Vector2(0,1), Vector2(1,1), Vector2(-1, 0), Vector2(0,0)]
}

var high_score = 0
var level = 1
```

This is the global script of the Tetris game, which stores several key variables. The Tetromino enumeration sets the types of tetrominoes that can be created, limiting them to seven shapes containing four squares each. This enumeration is used by the cells variable which shows the distribution of each square from the origin point of each tetromino. Each cell item contains four vectors, one for each square. The last two variables in this script are the high score and level variables which are utilized by the main script for difficulty scaling and storing the high score between gameplay sessions.

B4 Platformer

Figure B4.1: player.gd 1




```
extends CharacterBody2D

@export var speed = 300
@export var jump_velocity = -500
@export var terminal_velocity = 2000
@onready var can_jump = true
@onready var can_double_jump = true
@export var double_jump_unlocked = true
@export var wall_jump_unlocked = true

@onready var gravity_scale = 1.5
@onready var gravity = ProjectSettings.get_setting("physics/2d/default_gravity")
@onready var friction = 1500
@onready var air_resistance = 400

@onready var last_wall_normal = Vector2.ZERO

@onready var coyote_time = $Timers/CoyoteTime
@onready var wall_jump_time = $Timers/WallJumpTime

@onready var sprite_animation_timer = $Timers/SpriteAnimationTimer
@onready var sprite_sheet = $SpriteSheet
@onready var init_sprite_sheet_scale = $SpriteSheet.scale
@onready var init_scale = scale
@onready var state = "walk"

func _ready():
	Globals.player_init_position = position

func _physics_process(delta):
	var was_on_wall_only = is_on_wall_only()
	var was_on_floor = is_on_floor()
	if is_on_wall():
		last_wall_normal = get_wall_normal()
	if not is_on_floor():
		apply_gravity(delta)
	handle_movement(delta)
	handle_friction(delta)
	handle_jump()
	manage_timers(was_on_wall_only, was_on_floor)
	handleAnimation()
	move_and_slide()
	Globals.player_position = position
```

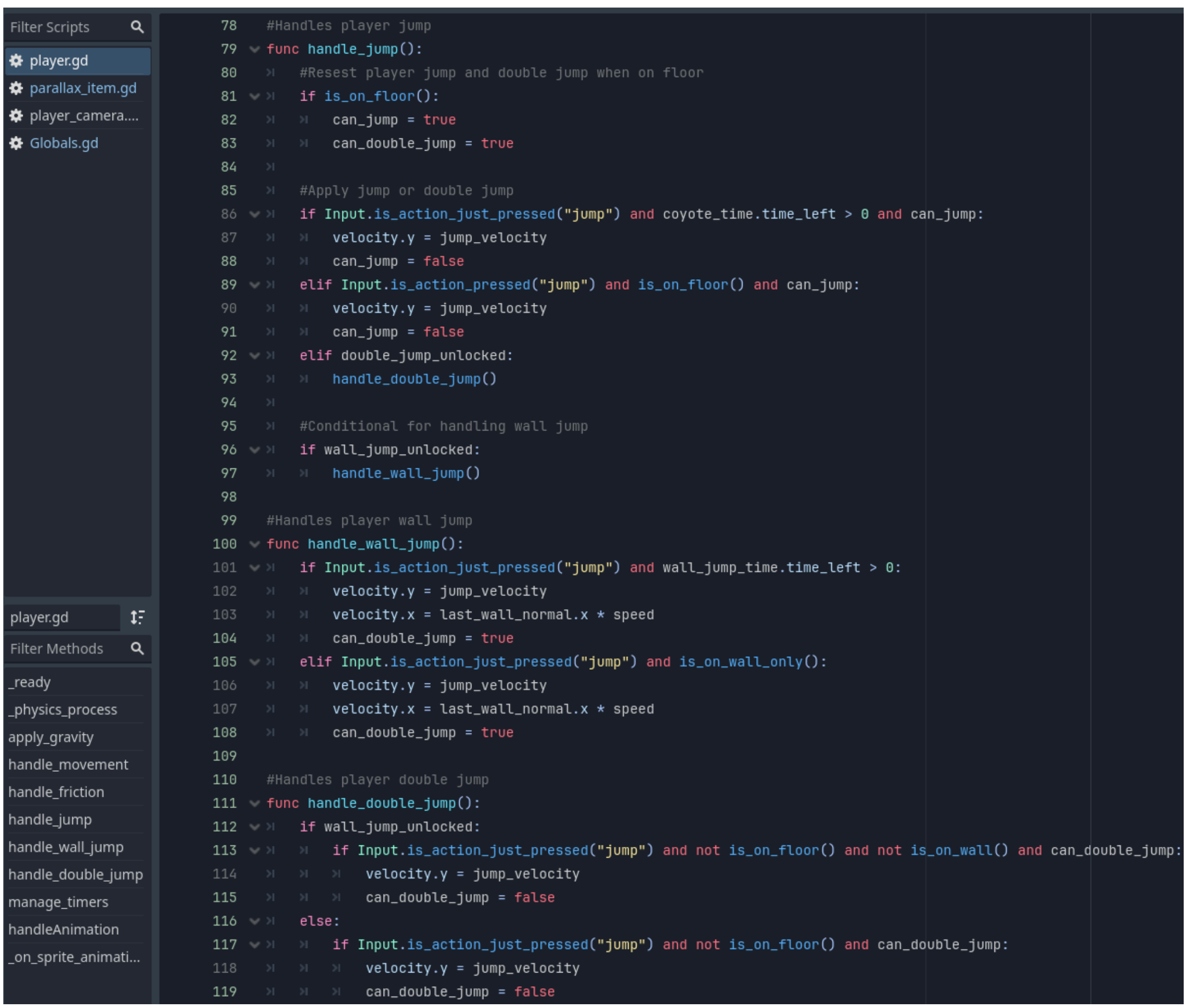
This is the beginning of the player.gd script. Tne of the core objects of the 2D platformer. In this script, the properties and operations of the player every frame can be seen. As far as variables go, the player has many values relevant to the player object's behavior such as speed for regulating the player's top horizontal speed, jump velocity for the amount of velocity applied to the player on jump, terminal velocity for limiting the player's highest speed of descent, alongside a list of boolean values for dictating actions the player can take and is able to take including can_jump, can_double_jump, double_jump_unlocked, and wall_jump_unlocked. Beyond these, the player has a set gravity, friction, and air resistance to add some limitations to their movement. The player also has a state variable which is used to keep track of the state of the player and regulate their behavior according to their state. In the _ready() function, the player's initial position is the only thing recorded. As for the _physics_process() function, all of the operations performed on the player to handle their movement, friction, jump, timers, and animations are completed in order (see figures B4.2 to B4.4). The global for player position is also set to equal the player position in this script to be utilized by other scripts more freely.

Figure B4.2: player.gd 2


```
#Applies gravity to player
func apply_gravity(delta):
	if state != "float":
		velocity.y = move_toward(velocity.y, terminal_velocity, gravity * gravity_scale * delta)
	else:
		velocity.y = move_toward(velocity.y, terminal_velocity/15, gravity * gravity_scale * delta)

#According to input, accelerates player left or right to max player speed
func handle_movement(delta):
	var direction = Input.get_axis("left", "right")
	var acceleration = speed * 10
	var air_acceleration = speed * 5
	if direction:
		if is_on_floor():
			state = "walk"
			velocity.x = move_toward(velocity.x, direction * speed, acceleration * delta)
		else:
			velocity.x = move_toward(velocity.x, direction * speed, air_acceleration * delta)
	elif is_on_floor():
		state = "idle"

#Applies friction to player speed according to contact with surfaces
func handle_friction(delta):
	#Apply friction when on wall
	if is_on_wall_only() and (velocity.y + gravity * delta) >= 100:
		velocity.y = 100
	
	#Apply friction or air resistance
	if is_on_floor():
		velocity.x = move_toward(velocity.x, 0, friction * delta)
	else:
		velocity.x = move_toward(velocity.x, 0, air_resistance * delta)
```

Continuation of the player.gd script. Here the apply_gravity(), handle_movement(), and handle_friction() functions can be seen. Each of these functions take in delta as a parameter and work to manage the various facets of the player's behavior. The apply_gravity() function moves the player's vertical velocity towards their terminal velocity at the rate of gravity. One detail of note in this function is that if the player is in the "float" state, they will be limited to a significantly reduced terminal velocity. The handle_movement() function first finds the direction of player input based on whether the player is pressing "A/LEFT" or "D/RIGHT". An acceleration and air acceleration is created for the player on the basis of the player's speed. If the player is holding a direction and is touching the ground, they will be accelerated in that direction up to their speed and will be placed into the "walk" state. If the player is in the air, they will not be placed into the "walk" state and will be accelerated at half the speed. If the player is not holding a direction and is on the floor, they will instead be placed into the "idle" state. The handle_friction() function is responsible for applying friction to the player when they are moving through the air, on the ground, and against a wall. Provided the player is pressed against a wall, their maximum vertical velocity is set to a finite point. If the player is in contact with the ground, they will be decelerated to a static velocity at a rate of the player's friction, meanwhile when the player is in the air, their horizontal velocity is decelerated to zero at a rate of the player's air resistance.

Figure B4.3: player.gd 3



Continuation of the player.gd script. The handle_jump(), handle_wall_jump(), and handle_double_jump() functions can be seen here. The handle_jump() function begins by checking for whether the player is in contact with the ground and resets the can_jump and can_double_jump booleans to true. After this, the function checks for whether the player has pressed "SPACE" and is able to jump. Provided that the player fulfills these conditions, its vertical velocity is set to the jump velocity of the player and the can_jump variable is set to false, not allowing the player to jump again until they make contact with the ground again. Provided that the player was not on the ground, the coyote timer had no time remaining upon pressing "SPACE", and double jump is unlocked, the handle_double_jump() function is run instead. Nested at the end of the handle_jump() function, given that the player has unlocked wall jump, the handle_wall_jump() function is run as well. The handle_wall_jump() function checks for whether the player is in contact with or was just in contact with a wall. Provided that this is true and the player presses "SPACE", the player's vertical velocity is set to their jump velocity and their horizontal velocity is set to their speed in the opposite direction of the wall. After a wall jump has been performed, the player's double jump is recovered. Last, the handle_double_jump() function sets the player's vertical velocity to the jump velocity provided that the player presses "SPACE" and the player is not in contact with the ground and they have not performed a double jump already while airborne.

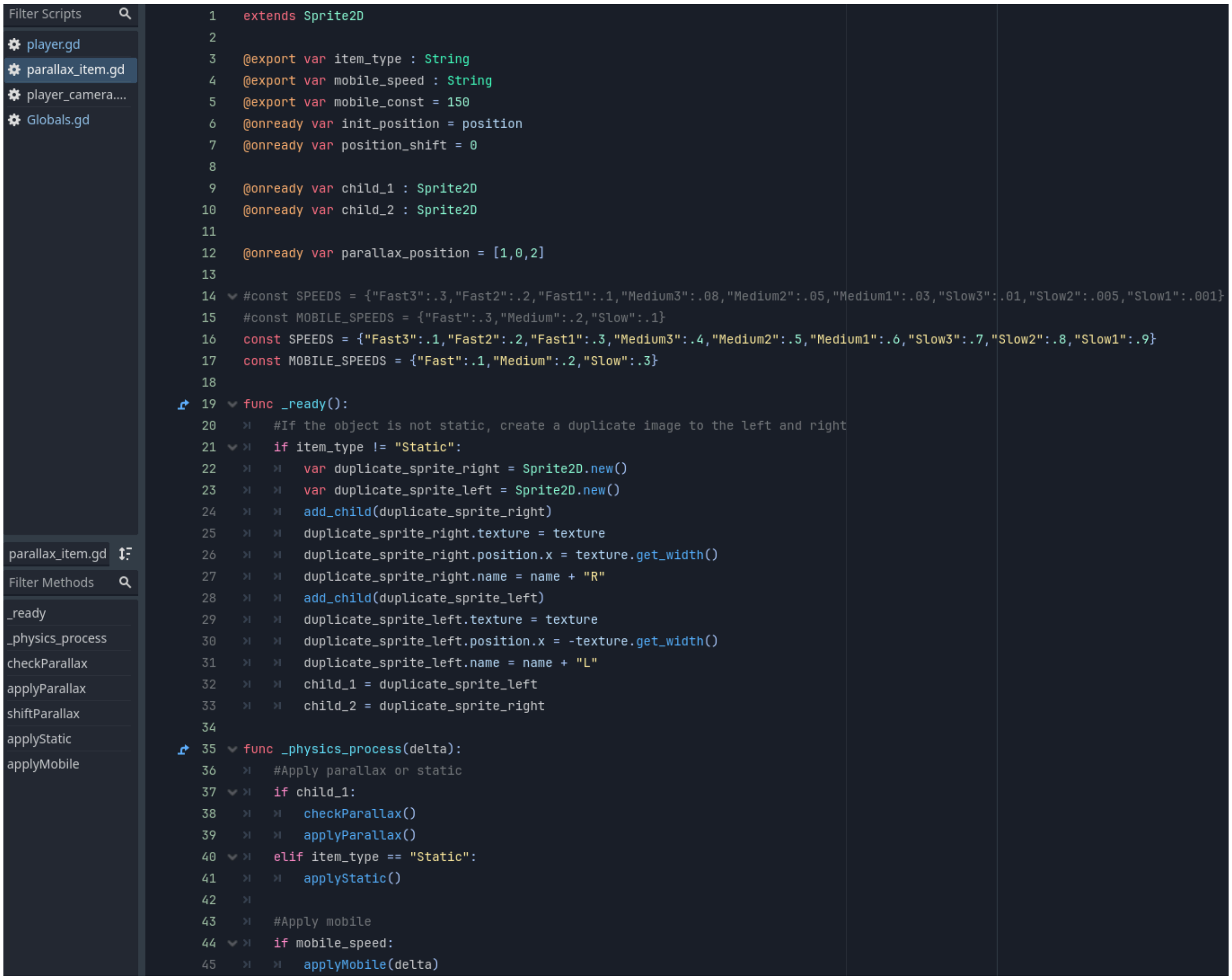
Figure B4.4: player.gd 4

```
Filter Scripts
player.gd
parallax_item.gd
player_camera....
Globals.gd

player.gd
Filter Methods
_ready
_physics_process
apply_gravity
handle_movement
handle_friction
handle_jump
handle_wall_jump
handle_double_jump
manage_timers
handleAnimation
_on_sprite_animati...
```

```
#Starts timers according to contact with wall and floor
func manage_timers(was_on_wall, was_on_floor):
	if (was_on_wall and not is_on_wall()) or is_on_wall_only():
		wall_jump_time.start()
	if (was_on_floor and not is_on_floor() and velocity.y >= 0) or is_on_floor():
		coyote_time.start()

#Handle player states and animations
func handleAnimation():
	var direction = Input.get_axis("left", "right")
	if is_on_wall_only():
		state = "wall"
		sprite_sheet.scale.x = get_wall_normal().x * init_sprite_sheet_scale.x
	elif not (is_on_floor() or is_on_wall()) and velocity.y > 0 and Input.is_action_pressed("up"):
		state = "float"
	elif not (is_on_floor() or is_on_wall()):
		state = "jump"

	if direction and not is_on_wall_only():
		sprite_sheet.scale.x = direction * init_sprite_sheet_scale.x

	if state == "jump":
		sprite_sheet.frame = 4
	elif state == "float":
		sprite_sheet.frame = 5
	elif state == "wall":
		sprite_sheet.frame = 6
	elif state == "idle" and sprite_sheet.frame > 1:
		sprite_sheet.frame = 0
	elif state == "walk" and sprite_sheet.frame < 2 or sprite_sheet.frame > 3:
		sprite_sheet.frame = 2

#Timer signal on a loop to cycle animation frames
func _on_sprite_animation_timer_timeout():
	if state == "idle":
		if sprite_sheet.frame == 0:
			sprite_sheet.frame = 1
		else:
			sprite_sheet.frame = 0
	elif state == "walk":
		if sprite_sheet.frame == 2:
			sprite_sheet.frame = 3
		else:
			sprite_sheet.frame = 2
```

Last section of the player.gd script. Here the manage_timers(), handleAnimation(), and _on_sprite_animation_timer_timeout() functions can be seen. The manage_timers() function takes two parameters was_on_wall and was_on_floor and serves the purpose of starting the wall_jump_time timer and coyote_time timer according to the surfaces the player was and is in contact with. Provided that the player was in contact with a wall and is no longer in contact with a wall, the wall_jump_time timer is started. Provided that the player was in contact with the floor and is no longer in contact with the floor, the coyote_time_timer is started. The handleAnimation() function takes one last look at the player's positioning and user input to make final adjustments to the player's state before changing the player's animation accordingly. The player's state works as a conditional code that determines the frame of the player's spritesheet is set. For the states "jump", "float", and "wall", the frame is directly set, regardless of any other factors. As for the "walk" and "idle" states which correspond to multi-frame animations, the frame is set to the first frame in the animation so long as the frame is outside of the bounds of the animation cycle. The _on_sprite_animation_timer_timeout() is responsible for the multi-frame animations of the player. When called, this function shifts the "walk" and "idle" animations forward by one, provided that the player is in the corresponding state.

Figure B4.5: parallax_item.gd 1



Beginning of the parallax_item.gd script. Here the setup of the script, the _ready() function, and _physics_process() function can be seen. One variable and two constants of note in the setup of this script are the parallax_position variable and the SPEEDS and MOBILE_SPEEDS constants. The parallax_position variable keeps track of the order of the main image and its two children images. The speed constants are a couple of dictionaries that are used to apply motion to the background items with variable speeds. The _ready() function is responsible for creating a duplicate of the background item to the left and right of itself. The purpose of these copies is to create a flush transition of the background while it moves according to the movement of the player. Copies of the image are created only in the case that the image is not static. The _physics_process() function runs a few sequences of logic in order to call appropriate functions for the given parallax item. If the item is static, the applyStatic() function is run (see figure B4.7) meanwhile if the item is a parallax, the checkParallax() and applyParallax() functions are run instead (see figure B4.6). If the image is a moving object with a mobile speed, the applyMobile() function is applied to it as well (see figure B4.7).

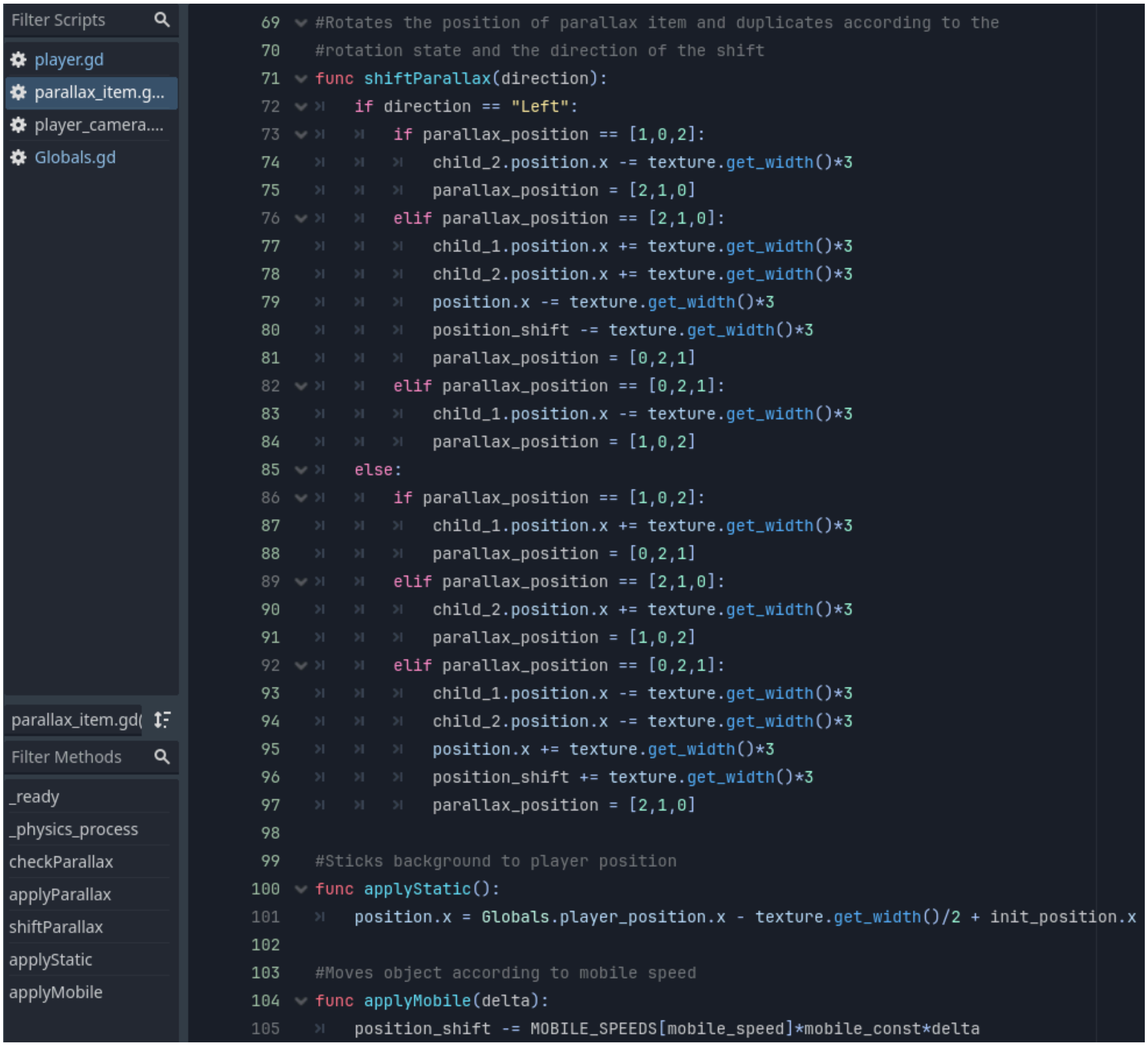
Figure B4.6: parallax_item.gd 2

Filter Scripts
player.gd
parallax_item.gd
player_camera....
Globals.gd

```
#Checks for player position and shifts the parallax left or right accordingly
func checkParallax():
	if parallax_position[0] == 1:
		if Globals.player_position.x - position.x > texture.get_width():
			shiftParallax("Right")
		elif Globals.player_position.x - position.x < 0:
			shiftParallax("Left")
	elif parallax_position[1] == 1:
		if Globals.player_position.x - position.x - child_1.position.x > texture.get_width():
			shiftParallax("Right")
		elif Globals.player_position.x - position.x - child_1.position.x < 0:
			shiftParallax("Left")
	elif parallax_position[2] == 1:
		if Globals.player_position.x - position.x - child_2.position.x > texture.get_width():
			shiftParallax("Right")
		elif Globals.player_position.x - position.x - child_2.position.x < 0:
			shiftParallax("Left")

#Moves parallax background according to change of player position from player initial position
func applyParallax():
	position.x = init_position.x + position_shift - (Globals.player_init_position.x - Globals.player_position.x) * SPEEDS[item_type]
```

Continuation of the parallax_item.gd script. Here the checkParallax() and applyParallax() functions can be seen. The checkParallax() function checks the state of the parallax item using the parallax_position variable which has three values that correspond to the order of the main image, the first child image, and the second child image respectively. Based on the state of the item and position relative to the player position, the shiftParallax() function (see figure B4.7) is called with either “Left” or “Right” as an argument. The applyParallax() function works to move the parallax item according to movement of the player. The speed of this movement is variable based on the item_type of the parallax item which is used as a key for the SPEEDS dictionary.

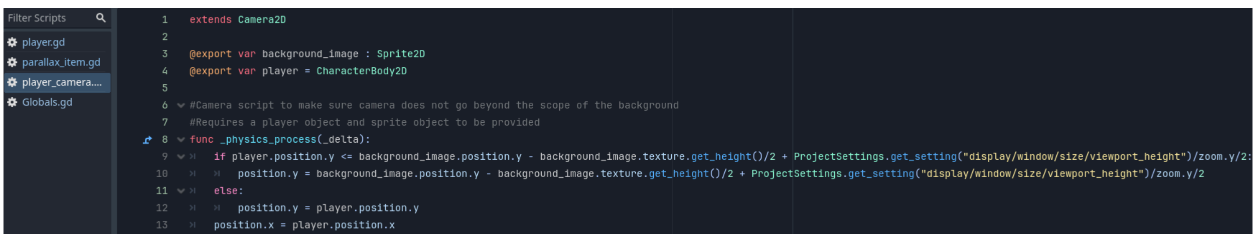
Figure B4.7: parallax_item.gd 3




```
#Rotates the position of parallax item and duplicates according to the
#rotation state and the direction of the shift
func shiftParallax(direction):
	if direction == "Left":
		if parallax_position == [1,0,2]:
			child_2.position.x -= texture.get_width()*3
			parallax_position = [2,1,0]
		elif parallax_position == [2,1,0]:
			child_1.position.x += texture.get_width()*3
			child_2.position.x += texture.get_width()*3
			position.x -= texture.get_width()*3
			position_shift -= texture.get_width()*3
			parallax_position = [0,2,1]
		elif parallax_position == [0,2,1]:
			child_1.position.x -= texture.get_width()*3
			parallax_position = [1,0,2]
	else:
		if parallax_position == [1,0,2]:
			child_1.position.x += texture.get_width()*3
			parallax_position = [0,2,1]
		elif parallax_position == [2,1,0]:
			child_2.position.x += texture.get_width()*3
			parallax_position = [1,0,2]
		elif parallax_position == [0,2,1]:
			child_1.position.x -= texture.get_width()*3
			child_2.position.x -= texture.get_width()*3
			position.x += texture.get_width()*3
			position_shift += texture.get_width()*3
			parallax_position = [2,1,0]

#Sticks background to player position
func applyStatic():
	position.x = Globals.player_position.x - texture.get_width()/2 + init_position.x

#Moves object according to mobile speed
func applyMobile(delta):
	position_shift -= MOBILE_SPEEDS[mobile_speed]*mobile_const*delta
```

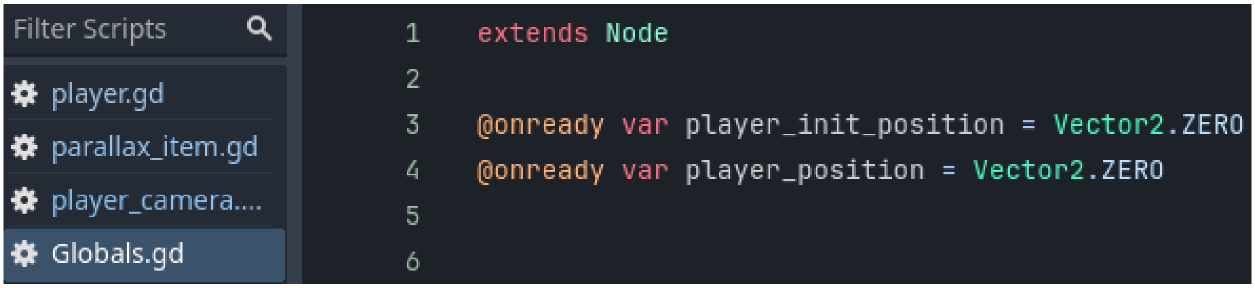
Final section of the parallax_item.gd script. Here the shiftParallax(), applyStatic(), and applyMobile() functions can be seen. The shiftParallax() function takes in one parameter for direction, which is used to inform which way the parallax item is to be shifted. This function works to move each image in the parallax item around one another in order to create a seamless transition as the player moves between them. When the player moves to the right, the leftmost image is moved to the right of the rightmost image. Meanwhile, when the player moves left, the rightmost image is moved to the left of the leftmost image. The parallax_position property is used to keep track of the relative position of these images and is changed accordingly when the images shift. The applyStatic() function works to keep the horizontal position of the image centered with the player in order to make the image static relative to the player. The applyMobile() function takes delta as a parameter and works to move a parallax item across the screen at a constant rate determined by the mobile_speed property of the item which is used as a key for the MOBILE_SPEEDS constant.

Figure B4.8: player_camera.gd



This script shows the custom player_camera.gd script used for keeping the player in view. As opposed to a typical player camera that is attached to the player as a descendent of the player, this camera operates separately from the player, with the player's information accessed via the exported player variable. Alongside this exported variable, is the exported background_image variable, which is used for the script to get an idea of the scale of the background the player is traversing through. This script has solely one function, the _physics_process() function which is responsible for keeping the player in the center of the viewport. The exception to this is if keeping the player in the center of the viewport would cause the camera's view to exceed the upper bounds of the background. Should that occur, based upon the camera's zoom, the viewport height, and background height, the vertical positioning of the camera is locked to the top of the background.

Figure B4.9: Globals.gd

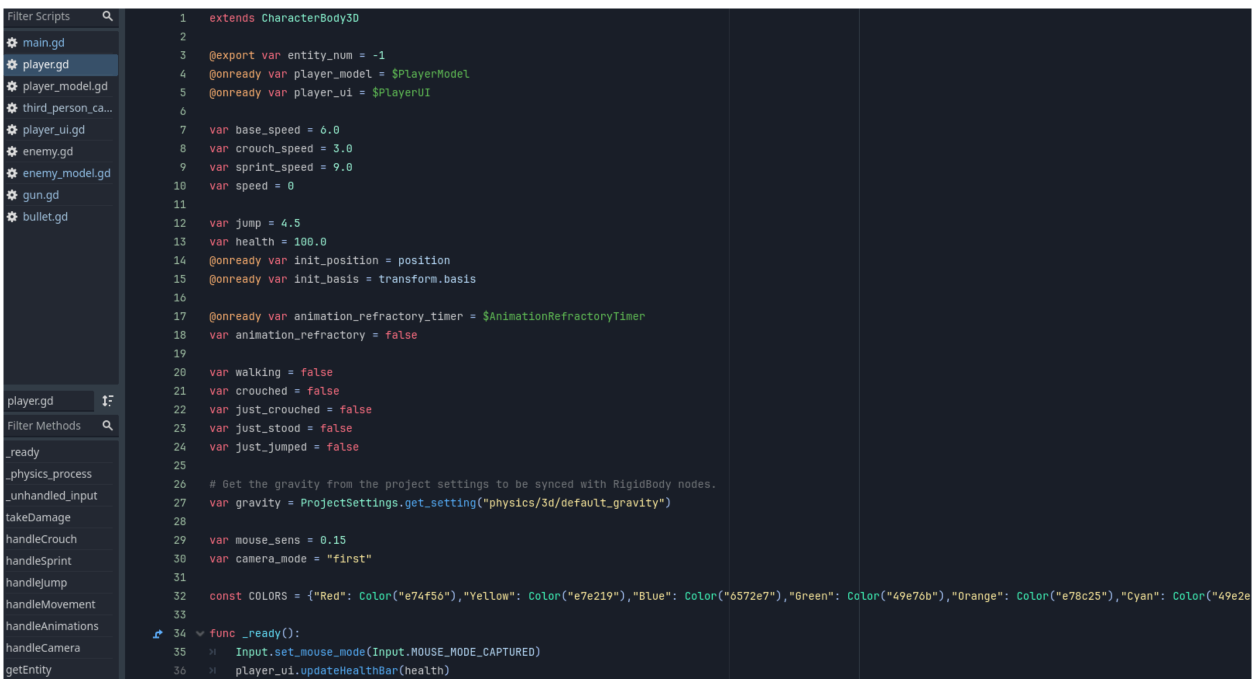



```
extends Node

@onready var player_init_position = Vector2.ZERO
@onready var player_position = Vector2.ZERO

```

Here the Globals script of the 2D platformer can be seen which is responsible for keeping track of the player's initial and active position.

B5 First Person Shooter

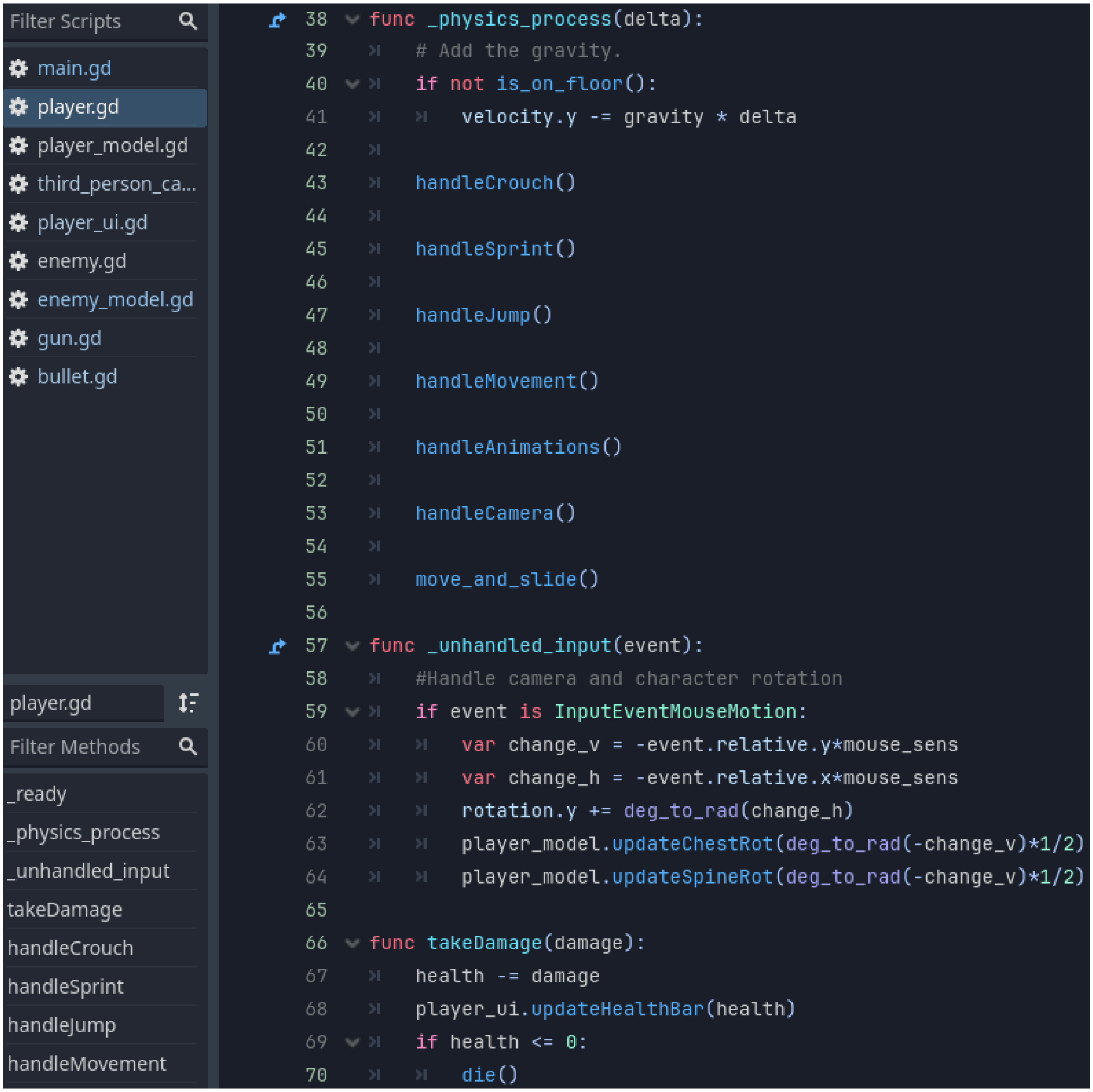
Figure B5.1: player.gd 1




```
extends CharacterBody3D

@export var entity_num = -1
@onready var player_model = $PlayerModel
@onready var player_ui = $PlayerUI

var base_speed = 6.0
var crouch_speed = 3.0
var sprint_speed = 9.0
var speed = 0

var jump = 4.5
var health = 100.0
@onready var init_position = position
@onready var init_basis = transform.basis

@onready var animation_refractory_timer = $AnimationRefractoryTimer
var animation_refractory = false

var walking = false
var crouched = false
var just_crouched = false
var just_stood = false
var just_jumped = false

# Get the gravity from the project settings to be synced with RigidBody nodes.
var gravity = ProjectSettings.get_setting("physics/3d/default_gravity")

var mouse_sens = 0.15
var camera_mode = "first"

const COLORS = {"Red": Color("e74f56"),"Yellow": Color("e7e219"),"Blue": Color("6572e7"),"Green": Color("49e76b"),"Orange": Color("e78c25"),"Cyan": Color("49e2e

func _ready():
	Input.set_mouse_mode(Input.MOUSE_MODE_CAPTURED)
	player_ui.updateHealthBar(health)
```

Beginning of the player.gd script of the first person shooter game. Here the setup of the player's variables and the _ready() function can be seen. The player holds an entity number to keep track of their identity alongside several values for the player's properties including an array of variables in charge of the player's current speed and max speeds, the player's jump velocity, and the player's health. Due to the muli-faceted nature of the player's animations, there are several booleans in charge of capturing the state of the player, as opposed to a string value to keep track of the player's state. A string-based state variable, however, is used to keep track of the camera_mode which is initialized in the "first" person mode. The _ready() function works to capture the mouse of the player, keeping it in the center of the screen, but allowing mouse movement to be read by the program. The updateHealthBar() method of the player UI (see figure B5.9) is also called with the player's health passed as an argument to maintain an accurate representation of the player's health.

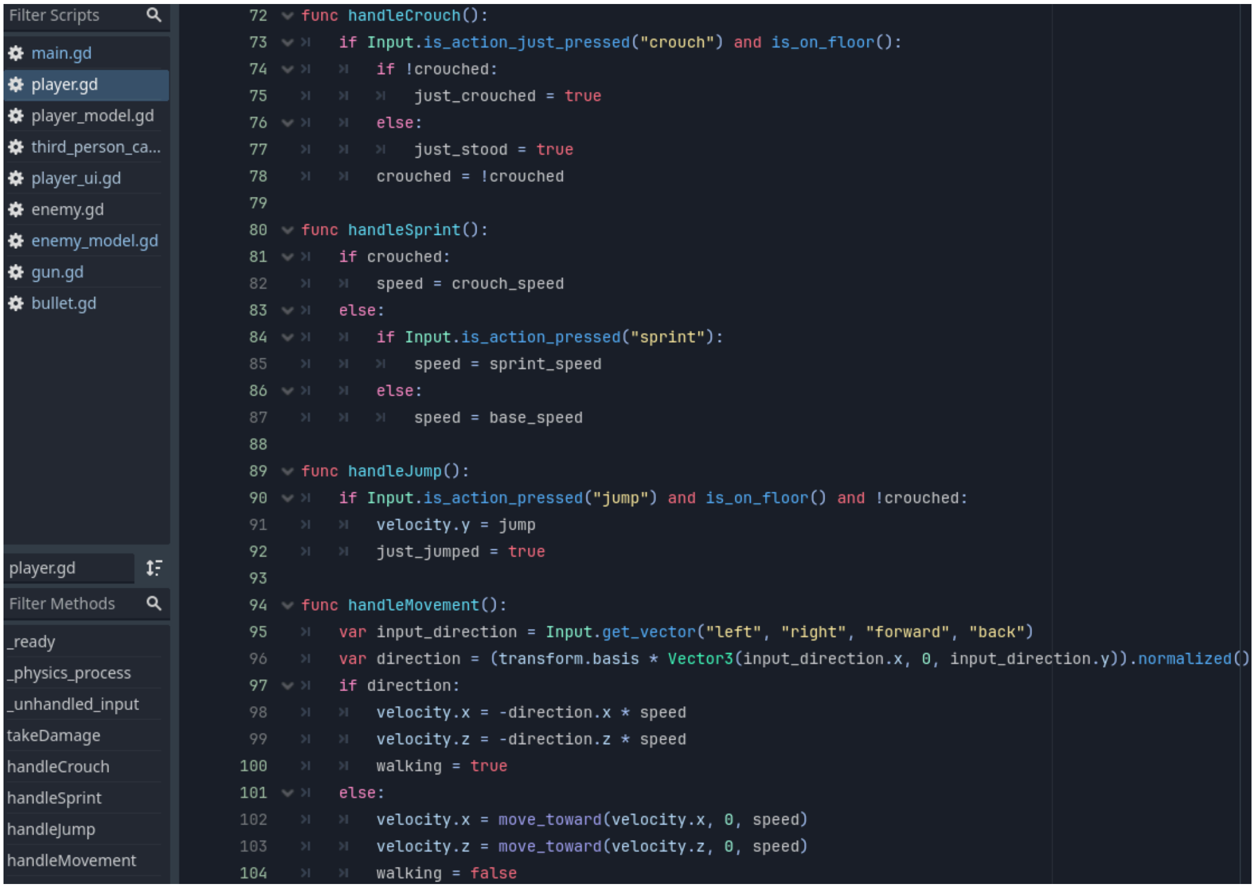
Figure B5.2: player.gd 2



Continuation of the player.gd script. Here the _physics_process(), _unhandled_input(), and takeDamage() functions can be seen. The _physics_process() function is run every frame and starts by checking for whether the player is in contact with the ground. If the player is not on the ground, gravity is applied to them. Following this logical structure, the various functions that constitute the player script are run in a sequential order. These functions are as follows:

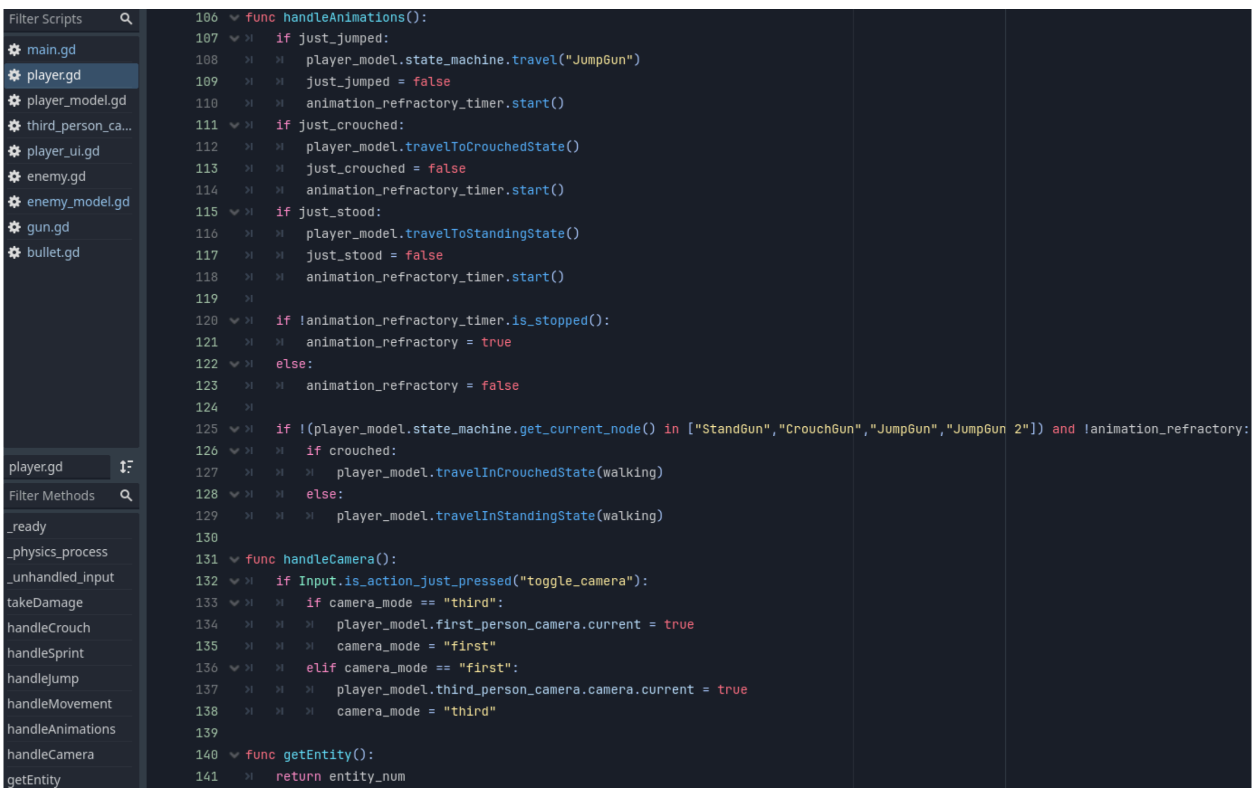
handleCrouch(), handleSprint(), handleJump(), handleMovement(), handleAnimations(), handleCamera(), and move_and_slide() (see figures B5.3 & B5.4). These functions are responsible for managing the movement, animations, and camera of the player. The _unhandled_input() function takes in an input event as a parameter and rotates the player horizontally corresponding to horizontal mouse movement. Provided vertical mouse movement, the updateChestRot() and updateSpineRot() functions are called on the player model (see figure B5.7) with the change in mouse position passed as an argument. Changes here are scaled with the mouse_sens variable. The takeDamage() function takes in one argument "damage" and works to decrement the player's health. The function reduces the player's health by the provided value, calls the updateHealthBar() function on the player UI scene (see figure B5.9), and checks for whether the player's health has fallen to or beneath zero. Should this happen, the die() function is called (see figure B5.5).

Figure B5.3: player.gd 3



Continuation of the player.gd script. Here the handleCrouch(), handleSprint(), handleJump(), and handleMovement() functions can be seen. The handleCrouch() function checks for whether the player has pressed “SHIFT” and either places the player into the crouched position, keeping track of the player’s state using the just_crouched, just_stood, and crouched boolean values. The handleSprint() function works to set the player speed equal to either their base_speed, their sprint_speed, or their crouched_speed. If the player is crouched, their speed is set to their crouch_speed. Otherwise, if the player is holding “CONTROL” their speed is set to their sprint_speed while if they are not holding “CONTROL” their speed is set to their base_speed. The handleJump() function checks for whether the player has pressed “SPACE” and that the player is standing and on the ground. Provided these conditions are fulfilled, the player’s vertical velocity is set to the jump variable and the just_jumped variable is set to true. The handleMovement() function finds the direction of the player by getting a vector of the player input between “W”, “A”, “S”, and “D” corresponding to the forward, left, backward, and right directions. This input is then converted into a direction in the form of a vector that can be applied to the player and is normalized. The player is then moved at their set speed in the corresponding direction. If there is no direction provided, the player is decelerated to a stationary position with a speed of zero. Provided movement, the walking variable is set to true whereas without movement the walking variable is set to false.

Figure B5.4: player.gd 4



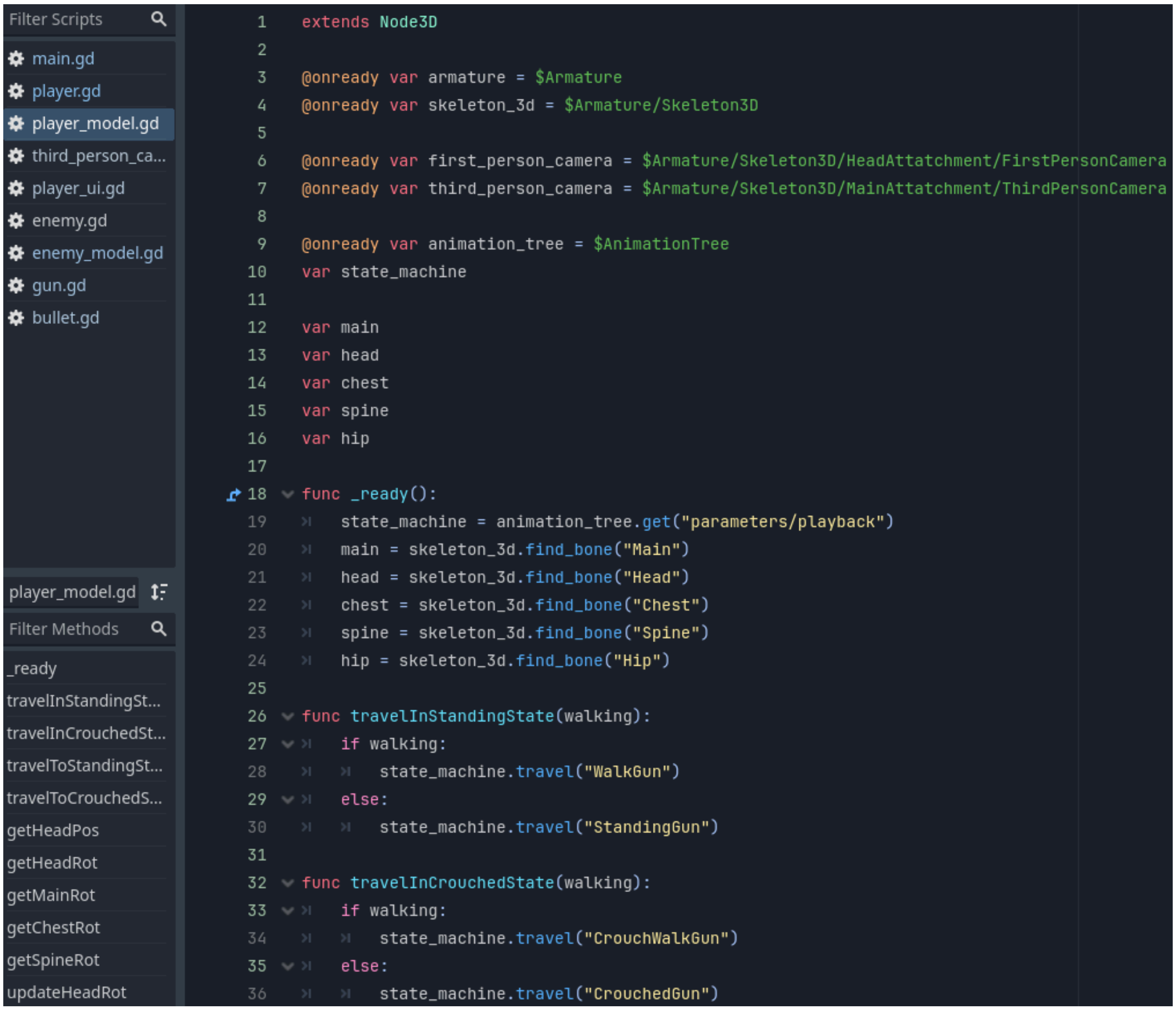

```
func handleAnimations():
	if just_jumped:
		player_model.state_machine.travel("JumpGun")
		just_jumped = false
		animation_refractory_timer.start()
	if just_crouched:
		player_model.travelToCrouchedState()
		just_crouched = false
		animation_refractory_timer.start()
	if just_stood:
		player_model.travelToStandingState()
		just_stood = false
		animation_refractory_timer.start()

	if !animation_refractory_timer.is_stopped():
		animation_refractory = true
	else:
		animation_refractory = false

	if !(player_model.state_machine.get_current_node() in ["StandGun","CrouchGun","JumpGun","JumpGun 2"]) and !animation_refractory:
		if crouched:
			player_model.travelInCrouchedState(walking)
		else:
			player_model.travelInStandingState(walking)

func handleCamera():
	if Input.is_action_just_pressed("toggle_camera"):
		if camera_mode == "third":
			player_model.first_person_camera.current = true
			camera_mode = "first"
		elif camera_mode == "first":
			player_model.third_person_camera.camera.current = true
			camera_mode = "third"

func getEntity():
	return entity_num
```

Continuation of the player.gd script. Here the handleAnimations(), handleCamera(), and getEntity() functions can be seen. The handleAnimations() function is responsible for managing the player model’s animations according to the booleans setters for the player’s state. Provided that the player has just_jumped, the state machine of the player model travels to the "JumpGun" animation. Provided that the player has just_crouched, the travelToCrouchedState() function of the player model is called (see figure B5.7). Provided that the player has just_stood, the travelToStandingState() function of the player model is called (see figure B5.7). In all three of these logical expressions, the boolean check value is set to false and the animation refractory timer is started. The animation_refractory variable is set to true or false depending on whether the animation refractory timer has time remaining or not. Provided that the player is walking, according to the state machine of the player model, the travelInCrouchedState() and travelInStandingState() functions of the player model (see figure B5.6) are called depending on whether the player is crouched or standing respectively. The handleCamera() function manages the state of the camera, toggling between the "first" and "third" person mode when the player presses "T". When toggled, the current camera switches between the first person camera and third person camera attached to the player model. The getEntity() function is a getter that returns the entity_num of the player.

Figure B5.5: player.gd 5

Final section of the player.gd script. Here the reset(), die(), and changeColor() function can be seen. The reset() function resets the transformation basis, position, health, and health bar of the player. The die() function calls the reset function. The changeColor() takes a color as a parameter and sets the surface material color of the player model's mesh to equal the color provided.

Figure B5.6: player_model.gd 1



Beginning of the player_model.gd script. Here the initial variable declarations alongside the _ready(), travelInStandingState(), and travelInCrouchedState() functions can be seen. The player model uses a state machine and five core bones to modify its model. These bones are the main, head, chest, spine, and hip bones which are all set to their respective bones on the skeleton in the _ready() function. The state machine is also set to the animation tree playback in the _ready()

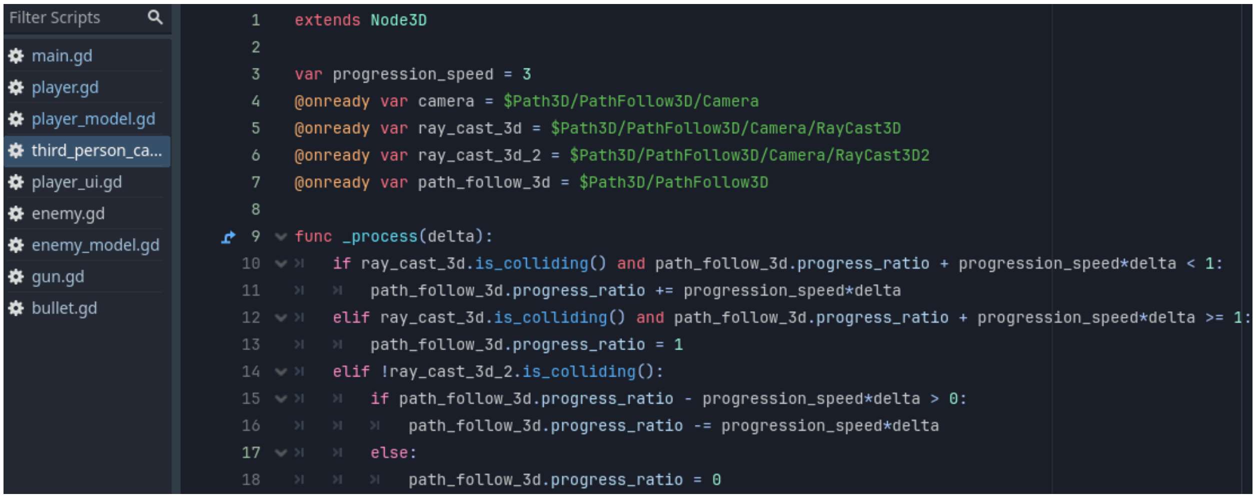
function. The travelInStandingState() and travelInCrouchedState() both take in whether the player is walking as a parameter and brings the state machine into the "WalkGun/CrouchWalkGun" state if the player is walking and the "StandingGun/CrouchedGun" state if the player is staying still.

Figure B5.7: player_model.gd 2


```
func travelToStandingState():
	state_machine.travel("StandGun")

func travelToCrouchedState():
	state_machine.travel("CrouchGun")

func getHeadPos():
	return skeleton_3d.get_bone_pose_position(head)

func getHeadRot():
	return skeleton_3d.get_bone_pose_rotation(head)

func getMainRot():
	return skeleton_3d.get_bone_pose_rotation(main)

func getChestRot():
	return skeleton_3d.get_bone_pose_rotation(chest)

func getSpineRot():
	return skeleton_3d.get_bone_pose_rotation(spine)

func updateHeadRot(amount):
	skeleton_3d.set_bone_pose_rotation(head,Quaternion(getHeadRot().x + amount,getHeadRot().y,getHeadRot().z,getHeadRot().w))

func updateChestRot(amount):
	skeleton_3d.set_bone_pose_rotation(chest,Quaternion(clamp(getChestRot().x + amount, deg_to_rad(-20), deg_to_rad(20)),getChestRot().y,getChestRot().z,getChest

func updateSpineRot(amount):
	skeleton_3d.set_bone_pose_rotation(spine,Quaternion(clamp(getSpineRot().x + amount, deg_to_rad(-20), deg_to_rad(20)),getSpineRot().y,getSpineRot().z,getSpine
```

Continuation of the player_model.gd script. Here two state machine functions, five getters, and three setters can be seen. The two state machine functions are the travelToStandingState() and travelToCrouchedState() functions which bring the player into the "StandGun" and "CrouchGun" states respectively. The five getters include the getHeadPos(), getHeadRot(), getMainRot(), getChestRot(), and getSpineRot(). These functions return the rotation or position of their respective bones. The three setters include updateHeadRot(), updateChestRot(), and updateSpineRot(). These functions take an amount as an argument by which the rotation of their respective bones are changed. For the updateChestRot(), and updateSpineRot() functions, the rotation is limited to 20 degrees from the base rotation.

Figure B5.8: third_person_camera.gd




```
extends Node3D

var progression_speed = 3
@onready var camera = $Path3D/PathFollow3D/Camera
@onready var ray_cast_3d = $Path3D/PathFollow3D/Camera/RayCast3D
@onready var ray_cast_3d_2 = $Path3D/PathFollow3D/Camera/RayCast3D2
@onready var path_follow_3d = $Path3D/PathFollow3D

func _process(delta):
	if ray_cast_3d.is_colliding() and path_follow_3d.progress_ratio + progression_speed*delta < 1:
		path_follow_3d.progress_ratio += progression_speed*delta
	elif ray_cast_3d.is_colliding() and path_follow_3d.progress_ratio + progression_speed*delta >= 1:
		path_follow_3d.progress_ratio = 1
	elif !ray_cast_3d_2.is_colliding():
		if path_follow_3d.progress_ratio - progression_speed*delta > 0:
			path_follow_3d.progress_ratio -= progression_speed*delta
		else:
			path_follow_3d.progress_ratio = 0
```

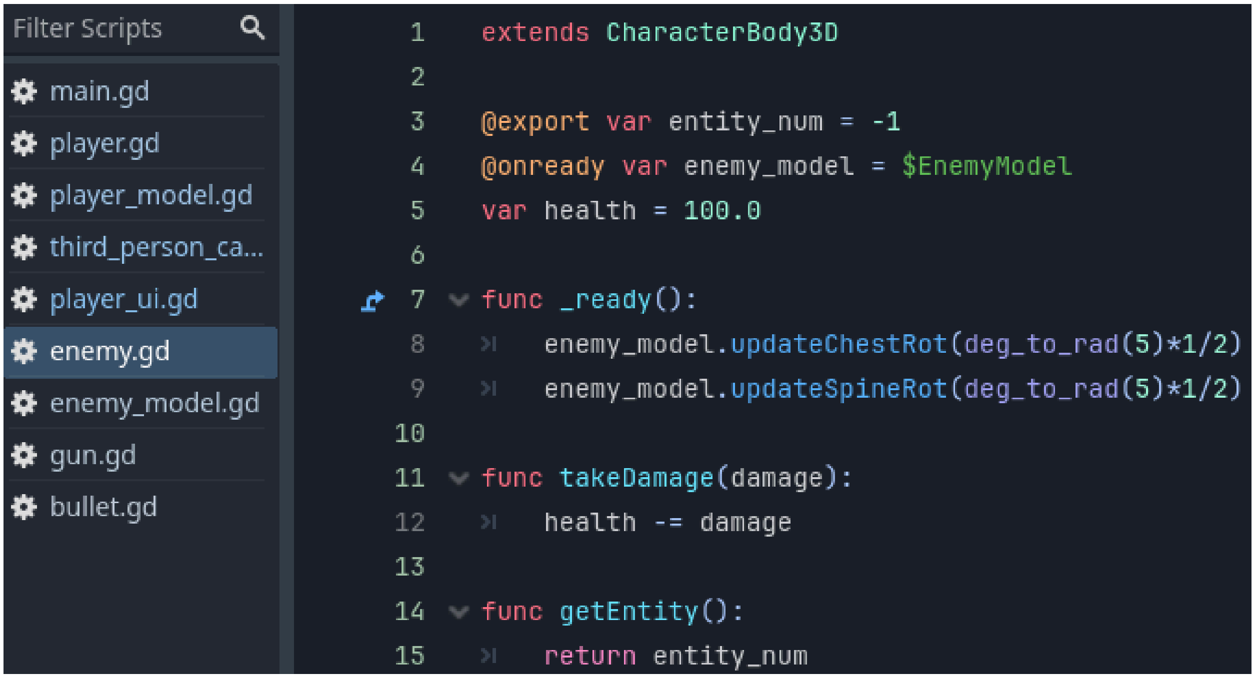
This script is responsible for moving the camera along a set path provided with collisions of child raycast objects. Each frame the logic of the third person camera script is run in the _process() function, in which the ray casts are checked for their collision status. If the first ray cast is colliding, the progress of the camera along the set path is incremented by the set progression speed of the camera. If this incrementation would set the progression of the camera along the path to be above one, the progression is instead set to one. If neither of the ray casts are colliding, the camera will move back down the path until it reaches the start.

Figure B5.9: player_ui.gd


```
extends Control

@onready var health_bar = $HealthBar

func updateHealthBar(health):
	health_bar.value = health

```

This script contains a single method to update the health bar shown to the player. The updateHealthBar() takes in a heath value as a parameter to set the value of the health bar.

Figure B5.10: enemy.gd



```
extends CharacterBody3D

@export var entity_num = -1
@onready var enemy_model = $EnemyModel
var health = 100.0

func _ready():
	enemy_model.updateChestRot(deg_to_rad(5)*1/2)
	enemy_model.updateSpineRot(deg_to_rad(5)*1/2)

func takeDamage(damage):
	health -= damage

func getEntity():
	return entity_num
```

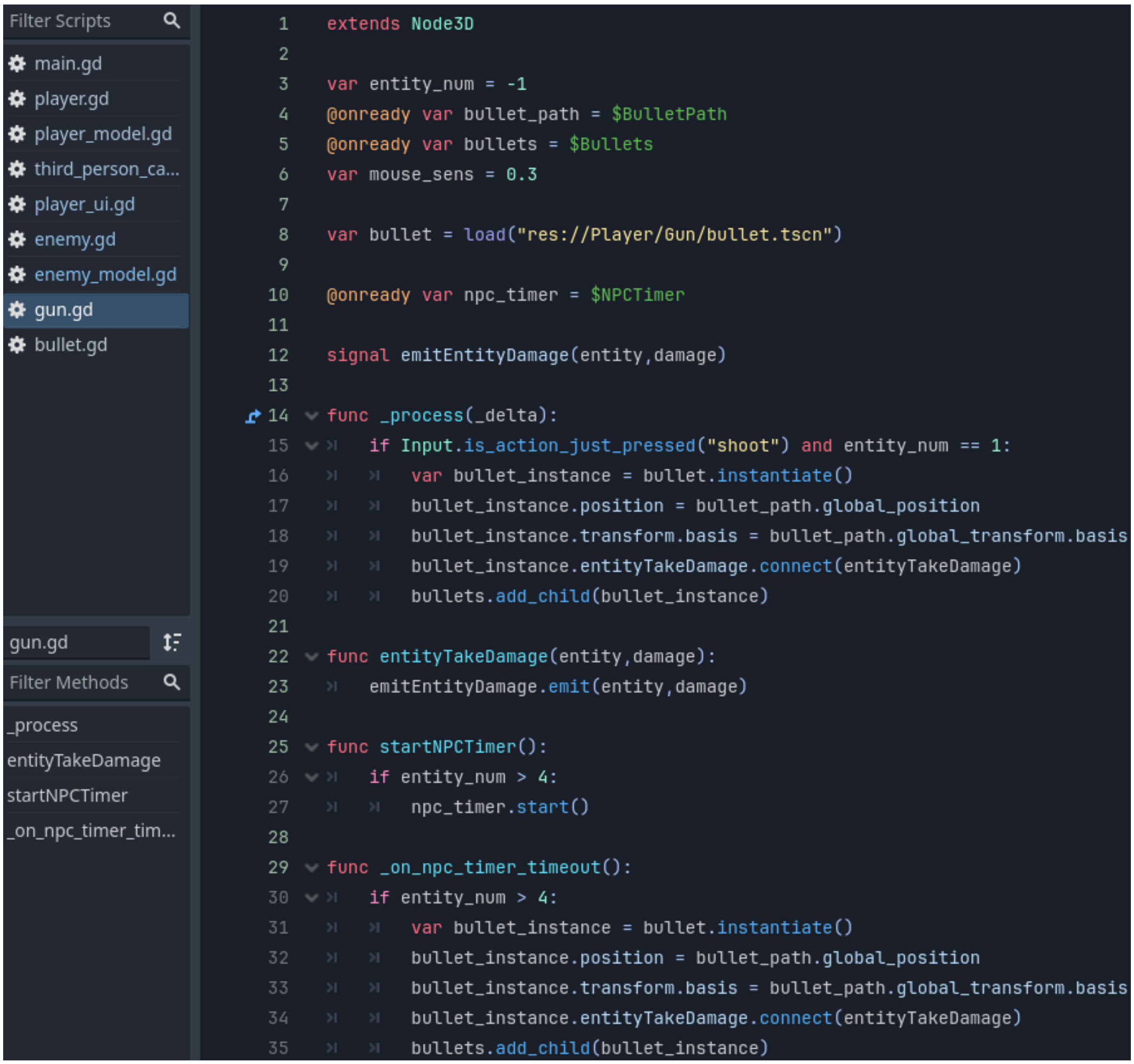
This script contains a few functions for simple enemy behavior. These functions include the _ready(), takeDamage(), and getEntity() functions. The _ready() function initializes the model of the enemy to have a set torso positioning using the updateChestRot() and updateSpineRot() functions of the enemy model (see figure B5.11). The takeDamage() function takes a damage value as a parameter and decrements the enemy's health by the provided value. The getEntity() function is a getter that returns the entity_num of the enemy.

Figure B5.11: enemy_model.gd

```
extends Node3D

@onready var skeleton_3d = $Armature/Skeleton3D

var chest
var spine

func _ready():
	chest = skeleton_3d.find_bone("Chest")
	spine = skeleton_3d.find_bone("Spine")

func getChestRot():
	return skeleton_3d.get_bone_pose_rotation(chest)

func getSpineRot():
	return skeleton_3d.get_bone_pose_rotation(spine)

func updateChestRot(amount):
	skeleton_3d.set_bone_pose_rotation(chest,Quaternion(clamp(getChestRot().x + amount, deg_to_rad(-20), deg_to_rad(20)),getChestRot().y,getChestRot().z,getChest

func updateSpineRot(amount):
	skeleton_3d.set_bone_pose_rotation(spine,Quaternion(clamp(getSpineRot().x + amount, deg_to_rad(-20), deg_to_rad(20)),getSpineRot().y,getSpineRot().z,getSpine
```

The enemy_model.gd script can be seen here bearing a significant resemblance to the player_model.gd script to a more limited degree (see figure B5.6 & B5.7). It contains only the chest and spine bones alongside the getters and setters of their rotation with the getChestRot(), getSpineRot(), updateChestRot(), and updateSpineRot() functions.

Figure B5.12: gun.gd




```
extends Node3D

var entity_num = -1
@onready var bullet_path = $BulletPath
@onready var bullets = $Bullets
var mouse_sens = 0.3

var bullet = load("res://Player/Gun/bullet.tscn")

@onready var npc_timer = $NPCTimer

signal emitEntityDamage(entity,damage)

func _process(_delta):
	if Input.is_action_just_pressed("shoot") and entity_num == 1:
		var bullet_instance = bullet.instantiate()
		bullet_instance.position = bullet_path.global_position
		bullet_instance.transform.basis = bullet_path.global_transform.basis
		bullet_instance.entityTakeDamage.connect(entityTakeDamage)
		bullets.add_child(bullet_instance)

func entityTakeDamage(entity,damage):
	emitEntityDamage.emit(entity,damage)

func startNPCTimer():
	if entity_num > 4:
		npc_timer.start()

func _on_npc_timer_timeout():
	if entity_num > 4:
		var bullet_instance = bullet.instantiate()
		bullet_instance.position = bullet_path.global_position
		bullet_instance.transform.basis = bullet_path.global_transform.basis
		bullet_instance.entityTakeDamage.connect(entityTakeDamage)
		bullets.add_child(bullet_instance)
```

This script contains key values to the scene and the functions relevant to the purpose of the gun scene. There functions involved with the gun script are the _process(), entityTakeDamage(), startNPCTimer(), and _on_npc_timer_timout(). The _process() function reads for player input of the "LEFT MOUSE BUTTON" and performs the act of firing the gun accordingly. It does this by creating an instance of the bullet scene and giving it a position and transformation basis of the end of the gun. The entityTakeDamage signal of the bullet scene is also connected to the entityTakeDamage() function. The entityTakeDamage() function takes an entity and a damage value as parameters and works to emit the emitEntityDamage signal with the same two values as arguments. This works to chain the signal sent by the bullet scene up to the gun scene and beyond. The startNPCTimer() is designed to start a timer of the gun scene solely for NPC characters, enemy characters. It checks that the entity has an entity number of greater than four, a value that is only provided for NPCs and starts the timer provided that is fulfilled. The _on_npc_timer_timeout() function also operates solely for NPC characters and instantiates a bullet in the very same manner as it would be for the player character pressing "LEFT MOUSE BUTTON".

Figure B5.13: bullet.gd

```
extends Node3D

var speed = 80
var damage = 20
var collisionMade = false

@onready var clear_timer = $ClearTimer

signal entityTakeDamage(entity,damage)

func _process(delta):
	position += transform.basis * Vector3(0, 0, speed) * delta

func destroyBullet():
	queue_free()

func _on_clear_timer_timeout():
	destroyBullet()

func _on_hit_box_body_entered(body):
	if !collisionMade:
		var entity = body.get_parent().get_parent().get_parent().get_parent().get_parent()
		if body.get_collision_layer() == 4:
			entityTakeDamage.emit(entity.entity_num,damage*.5)
		elif body.get_collision_layer() == 8:
			entityTakeDamage.emit(entity.entity_num,damage*.75)
		elif body.get_collision_layer() == 16:
			entityTakeDamage.emit(entity.entity_num,damage)
		elif body.has_method("takeDamage"):
			body.takeDamage(damage)
	collisionMade = true
	destroyBullet()
```

This script contains several functions designed around the purpose of moving the bullet forward, colliding with objects, and destroying the bullet, very similar to that of the laser scene of the Ateroids game (see figure B5.11). These include the _process(), destroyBullet(), _on_clear_timer_timeout(), and _on_hit_box_body_entered() functions. The _process() function simply works to move the bullet straight forward based upon its angle of instantiation by applying the speed of the bullet to the bullet's position over time. The destroyBullet() function simply frees the bullet. The _on_clear_timer_timeout() calls the destroyBullet() function. The _on_hit_box_body_entered() takes in a body as a parameter and is called when the bullet collides with an object. Provided that the object collided with is on the collision layer corresponding to the player and enemy objects, the entityTakeDamage signal is sent out with two arguments: The entity number of the object collided with and the damage inflicted which is modified on the basis of which body part is hit. The bullet is then destroyed using destroyBullet().

Figure B5.14: damageManager.gd


```
extends Node

# Entity number -1 = Non Assigned, 1-4 = Player, 5+ = Enemy
func _ready():
	var player_num = 1
	var enemy_num = 5
	for player in get_tree().get_nodes_in_group("Player"):
		if player.entity_num < 0:
			player.entity_num = player_num
		if player.has_node("PlayerModel/Armature/Skeleton3D/HandRAttatchment/Gun"):
			player.get_node("PlayerModel/Armature/Skeleton3D/HandRAttatchment/Gun").emitEntityDamage.connect(manageDamage)
			player.get_node("PlayerModel/Armature/Skeleton3D/HandRAttatchment/Gun").entity_num = player.entity_num
			player_num += 1
	for enemy in get_tree().get_nodes_in_group("Enemy"):
		if enemy.entity_num < 0:
			enemy.entity_num = enemy_num
		if enemy.has_node("EnemyModel/Armature/Skeleton3D/HandRAttatchment/Gun"):
			enemy.get_node("EnemyModel/Armature/Skeleton3D/HandRAttatchment/Gun").emitEntityDamage.connect(manageDamage)
			enemy.get_node("EnemyModel/Armature/Skeleton3D/HandRAttatchment/Gun").entity_num = enemy.entity_num
			enemy.get_node("EnemyModel/Armature/Skeleton3D/HandRAttatchment/Gun").startNPCTimer()
		enemy_num += 1

func manageDamage(entity_num,damage):
	for entity in get_tree().get_nodes_in_group("Player") + get_tree().get_nodes_in_group("Enemy"):
		if entity_num == entity.entity_num:
			entity.takeDamage(damage)
```

This script is responsible for connecting damage signals to their appropriate sources through the _ready() and manageDamage() functions. Upon initiation, in the _ready() function all players and enemies in the active scene are provided with an entity number and have their emitEntityDamage signal connected with the manageDamage function. The manageDamage() function takes in an entity number and damage value as parameters and calls the takeDamage() function on the entity with the damage value as an argument.

Appendix C - Glossary

C1 General

Elastic: Ability of matter to maintain its energy upon collision with another object.

Frame: Single instance of calculation visible to the player.

Parent-Child Hierarchy: Object organization structure in which objects can be the parent of many and the child of one. Children share attributes with their parent and are able to directly communicate with them. Two children beneath the same parent are called siblings.

Print: To draw text to the screen or console of a program.

Scale: The multiplicative factor over the dimensions of an object.

Score: Numeric value denoting performance and/or achievement.

Screenshot: Image of a single instance of the screen of an electronic device.

Size: The volume of an object; typically measured in pixels.

User Interface (UI): Also known as Graphical User Interface (GUI); overlay of visualized data for the purposes of a user.

Video Game: A program on a computer or console centered around entertainment and/or competition often with a set of rules and goals.

Viewport: Space in which a player is able to observe the game space.

C2 Godot

C2.1 Base Nodes

Node: A base class from which all nodes inherit; contains all of the base properties, methods, signals, enumerations, and constants of an object in *Godot*.

CanvasItem: Base class for all 2D nodes; is inherited by Node2D and Control.

Timer: A timer that counts down by a set amount.

AnimationMixer: Base class for animation nodes; is inherited by AnimationPlayer and AnimationTree.

AnimationPlayer: A node responsible for cycling through animations with set parameters; all manipulatable properties can be animated including sprite frame, position, and rotation.

AnimationTree: A node responsible for creating complex animation transitions; allows for movement between one animation and another when set conditions are met.

C2.2 Control Nodes

Control: A base class for all UI nodes; contains features tailored to the development of a GUI

TextureRect: A node responsible for displaying a texture.

Panel: A node responsible for displaying a StyleBox (stylized 2D resource).

Container: A base class for UI containers.

BoxContainer: A node responsible for organizing items along one axis of alignment; is inherited by VBoxContainer and HBoxContainer.

VBoxContainer: A node responsible for organizing items with vertical alignment.

HBoxContatiner: A node responsible for organizing items with horizontal alignment.

Label: A node responsible for displaying text.

C2.3 2D Nodes

Node2D: A base class for all 2D game objects; contains features tailored to the development of game objects in a 2D scene.

CharacterBody2D: A 2D node responsible for the development of a kinematic object; tailored to the development of a player character.

StaticBody2D: A 2D node responsible for creating stationary objects uninfluenced by the physics engine.

Area2D: A 2D node responsible for detecting other objects and areas.

CollisionShape2D: A 2D node responsible for providing a shape for physics objects with preset shapes; shapes include circle, rectangle, and capsule.

CollisionPolygon2D: A 2D node responsible for providing a shape for physics objects using connecting points which come together to form a polygon.

Polygon 2D: A 2D node responsible for creating a visible polygonal shape.

Sprite2D: A 2D node responsible for visually representing a sprite.

Camera2D: A 2D node responsible for presenting the viewport to the player; can be constrained and dynamically modified during gameplay.

C2.4 3D Nodes

Node 3D: A base class for all 3D game objects; contains features tailored to the development of game objects in a 3D scene.

CharacterBody3D: A 3D node responsible for the development of a kinematic object; tailored to the development of a player character.

StaticBody3D: A 3D node responsible for creating stationary objects uninfluenced by the physics engine.

Area3D: A 3D node responsible for detecting other objects and areas.

CollisionShape3D: A 3D node responsible for providing a shape for physics objects with preset shapes; shapes include circle, rectangle, and capsule.

CollisionPolygon3D: A 3D node responsible for providing a shape for physics objects using connecting points which come together to form a polygon.

MeshInstance3D: A 3D node responsible for creating visuals for all 3D objects; 3D models are automatically converted into this node when brought into *Godot*.

RayCast3D: A 3D node composed of a single line in 3D space; used to detect any collision objects intersecting with a given space.

Path3D: A 3D node responsible for creating a path in 3D space for other nodes to follow.

PathFollow3D: A 3D node responsible for mediating behavior of nodes on a given path; works closely with a parent Path3D node.

Skeleton3D: A 3D node responsible for holding a hierarchy of the bones of a 3D model; used for the animation of 3D models.

BoneAttachment3D: A 3D node that attaches itself dynamically to the transformation basis of the bone of a Skeleton3D.

Camera3D: A 3D node responsible for presenting the viewport to the player; can be constrained and dynamically modified during gameplay.

References

AABB. (n.d.). Godot Engine Documentation. Retrieved March 26, 2025, from https://docs.godotengine.org/en/stable/classes/classes/class_aabb.html

Asteroids (video game). (2025). In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Asteroids_(video_game)&oldid=1289654658

blender.org - Home of the Blender project - Free and Open 3D Creation Software. (n.d.). Blender.Org. Retrieved May 14, 2025, from https://www.blender.org/

Celeste. (n.d.). Celeste. Retrieved May 14, 2025, from http://celestegame.com/

cplusplus. (n.d.). Retrieved May 14, 2025, from https://cplusplus.com/

Doom (1993 video game). (2025). In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Doom_(1993_video_game)&oldid=1290423544

First-person shooter. (2025). In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=First-person_shooter&oldid=1288872248

Fortnite. (n.d.). Fortnite | Free-to-Play Cross-Platform Game - Fortnite. https://www.fortnite.com/

GDScript reference. (n.d.). Godot Engine Documentation. Retrieved February 27, 2025, from https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/tutorials/scripting/gdscript/gdscript_basics.html

Godot Docs. (n.d.). Godot Engine Documentation. Retrieved February 27, 2025, from https://docs.godotengine.org/en/stable/index.html

Griffin, J. (n.d.). *Leaderboards - the original and best social feature* ... Retrieved April 7, 2025, from https://www.gamedeveloper.com/design/leaderboards---the-original-and-best-social-feature-

Is Python faster and lighter than C++? (2013, February 16). [Forum post]. Cross Validated. https://stackoverflow.com/questions/801657/is-python-faster-and-lighter-than-c

Jose, S. (2024, October 30). Why C++ Is Climbing the Ranks: An In-Depth Look at its TIOBE Popularity Surge. *Medium*. https://medium.com/@najascj/why-c-is-climbing-the-ranks-an-in-depth-look-at-its-tiobe-popularity-surge-2a5f7b69e7f3

Godot Engine - Free and open source 2D and 3D game engine. (n.d.). Godot Engine. Retrieved February 27, 2025, from https://godotengine.org/

Jump King. (n.d.). Retrieved May 14, 2025, from https://www.jump-king.com/

LOVE. (n.d.). Retrieved February 27, 2025, from https://love2d.org/wiki/Main_Page

LÖVE - Free 2D Game Engine. (n.d.). Retrieved February 27, 2025, from https://love2d.org/

Mendes, L. O., Cunha, L. R., & Mendes, R. S. (2022). Popularity of Video Games and Collective Memory. *Entropy*, *24*(7), 860. https://doi.org/10.3390/e24070860

Minecraft Website. (n.d.). Retrieved May 14, 2025, from https://www.minecraft.net/en-us

Node. (n.d.). Godot Engine Documentation. Retrieved May 14, 2025, from https://docs.godotengine.org/en/stable/classes/classes/class_node.html

Nolan Bushnell | Lemelson. (n.d.). Retrieved March 18, 2025, from https://lemelson.mit.edu/resources/nolan-bushnell

Official Portal 2 Website. (n.d.). Retrieved May 14, 2025, from https://www.thinkwithportals.com/

Oliver, M. B., Bowman, N. D., Woolley, J. K., Rogers, R., Sherrick, B. I., & Chung, M.-Y. (2016). Video games as meaningful entertainment experiences. *Psychology of Popular Media Culture*, *5*(4), 390–405. https://doi.org/10.1037/ppm0000066

Platformer. (2025). In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Platformer&oldid=1290148893

Pong. (2025). In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Pong&oldid=1290096191

Pygame Front Page — pygame v2.6.0 documentation. (n.d.). Retrieved February 27, 2025, from https://www.pygame.org/docs/

RollerCoaster Tycoon. (n.d.). Atari®. Retrieved February 27, 2025, from https://atari.com/pages/rollercoaster-tycoon

RollerCoaster Tycoon (video game). (2024). In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=RollerCoaster_Tycoon_(video_game)&oldid=1266324800

Shivang. (2024, August 2). Game Development Team - Structure, Roles & Cost. *Richestsoft*. https://richestsoft.com/blog/game-development-team-structure-roles/

Skopljakovic, E. (2019). *Gaming as a Social Construct: Towards a Framework for Player Socialization in Massive Multiplayer Online Videogames - ProQuest*. Retrieved March 21, 2025, from https://www.proquest.com/docview/3059336401

Tetris. (2025). In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Tetris&oldid=1290198146

The Programming Language Lua. (n.d.). Retrieved May 14, 2025, from https://www.lua.org/

The Ren'Py Visual Novel Engine. (n.d.). Retrieved February 27, 2025, from https://www.renpy.org/

TIOBE Index. (n.d.). TIOBE. Retrieved February 27, 2025, from https://www.tiobe.com/tiobe-index/

Unity Real-Time Development Platform | 3D, 2D, VR & AR Engine. (n.d.). Unity. Retrieved February 27, 2025, from https://unity.com

Unreal Engine. (n.d.). Unreal Engine. Retrieved February 27, 2025, from https://www.unrealengine.com/

VALORANT. (2025, May 13). https://playvalorant.com/en-us/

Welcome to Python.org. (2025, May 7). Python.Org. https://www.python.org/

Yoon, S. (n.d.). *Gaming Culture: A New Language for the Digital Age*. Forbes. Retrieved March 20, 2025, from https://www.forbes.com/sites/forbesbooksauthors/2024/05/14/gaming-culture-a-new-language-for-the-digital-age/