# 4 Implementation Chapter

## IsWalled() function

This function will return a Boolean variable, using:

“wallCheck” - an empty game object that is placed on the side of the player.

“wallLayer” - a layer mask of all the walls with the Layer “Wall”, and

“radOfCircle” - the radius used to check if the player and the wall are touching this circle, this variable is also used in the “IsGrounded()” function.

A black background with white text

Description automatically generated

A computer screen with white text

Description automatically generated

This is showing the “Walls” object tagged with the layer “Wall”:

A blue stripe on a black background

Description automatically generated

The “Wall Check” object is part of the player similar to “Ground Check”.



In the inspector the Wall Layer is set on “Wall” and the Wall Check object selected.

A screen shot of a computer

Description automatically generated

## IsGrounded() Function

This function returns a Boolean variable, using the variables “groundCheck” (just under the player’s collider), “radOfCircle” (the radius of that circle), and the layer mask “groundMask”. The ground mask is applied to all objects that are in the layer “Ground”. The Boolean returns true if the ground mask overlaps with the circle under the player’s collider and false if it doesn’t.

Text

Description automatically generated

The author also draws a Gizmo which can be seen in the editor but not in the game. This makes it easier to see where the ground is intersecting with the player.

Graphical user interface, text

Description automatically generated

Graphical user interface, text

Description automatically generated

## WallSlide() Function

This function makes the player slowly slide down a wall when they move into it if they are not on the ground already. The “Y” velocity is Clamped to not exceed a certain speed. This function uses the “isWallSliding” Boolean and “wallSlidingSpeed” float which controlls the speed the player will fall at.

The player will slide down the wall as long as they are moving towards the wall.

A screenshot of a computer screen

Description automatically generated

A computer screen shot of a code

Description automatically generated

## Flip() Function

This function will change the direction of the object it is used on. The author changes the scale of the object’s X value by multiplying it by -1 which just flips the direction the object is facing.

Below shows how the flip function is used In the “Update()” function. If the object is not facing to the right, and the direction value is greater than 0, then it flips the object to the opposite direction and vice versa.

Text

Description automatically generated

Text

Description automatically generated

\*\*\*\*\*\* UPDATE \*\*\*\*

## Player Movement

The author uses Unity’s new Input System for keyboard and mouse, and Gamepad. On keyboard the player will use the “A” and “D” keys to move direction, and the “Space” key to Jump. On Gamepad the player will use the left joystick to move and the button south to jump.

Graphical user interface, application

Description automatically generatedGraphical user interface, text, application

Description automatically generated

In the Events tab the author has their “Player.cs” script which is attached to the player. They use the “Move()”, “Jump()”, “Dash(), “Fire()” and “Interact()” functions. For example, when the space bar is pressed the “Jump()” function checks if the action was performed and if the player has a jump available, it will make the player jump. The Author will go into more detail in the individual Functions later in this section.

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedA grey rectangular box with white text

Description automatically generated

The Author created the “Move()” function which reads the x direction the player is trying to move. The direction is a value of 1, -1 or 0. If the player is not moving the value is 0. If the player is moving the value is either 1 or -1 depending on the direction. This function also starts the run animation which happens when speed is not 0, however, the jump animation takes priority over the run animation. For example, if the player jumps and is moving to the side it will not play the run animation because the jump animation is being played.

### Dash

The Author created the “Fire()” function which first checks:

1. If the player can attack and if they can, it will use the “Attack()” function which activates the “attack1” trigger. This plays the attack animation. It is called attack1 because the author wants to have two swings of the attack if the player presses attack in quick succession, this might not be implemented because of time constraints.
2. Next the “Attack()” function will put all enemies that are in the attack range when the button is pressed into an array called “hitEnemies”.
3. It will run the array through a “foreach” loop that will deal damage to all the enemies hit.
4. “Fire()” will then put the attack on a “Cooldown” where the player cannot attack again for about a second.

### Jump

The Author created the “Jump()” function which will first check:

1. If the player can wall jump, which is something the player can only do when they are touching a wall. If true this will start a “Coroutine” called “WallBounce()” which will play the jump animation and set the player’s gravity scale to 0 so the wall jump isn’t affected by outside forces. It then pushes the player away from the wall and up into the air as if they jumped off the wall. The player’s gravity scale is then reset back to its original value.
2. If the player is not “isWallSliding” (the variable used to check if the player is sliding on the wall) and their max number of jumps is greater than the current number of jumps used, the variable “jumpForce” is applied to the player which makes the player jump. This is also where the coyote timer is used which will be discussed later under “Coyote Time Jump”.
3. “Jump()” will check if the “Jump” key was released or cancelled while the player was moving up (y velocity greater than 0). When this happens the y velocity is multiplied by 0.5 to slow the player down. This allows the player to jump at different heights depending on how long they hold the “Jump” key.

### Dash

The Author created the “Dash()” function which first checks:

1. If the player can dash with the Boolean “canDash”. If true, the “Coroutine” “Dash()”will start.
2. This “Coroutine” sets the “dashing” animation trigger, makes the player’s gravity scale 0, adds a velocity to the player and starts emitting a trail renderer. It will wait for “dashingTime” seconds.
3. Once done waiting, it will turn off the trail renderer, reset the player’s gravity scale and reset the “dashing” animation trigger. It will then wait for “dashingCooldown” seconds before setting “canDash” back to true.

## Fire() Function

The “Fire()” function is using the new input system in Unity. When the specified input is pressed (J on keyboard, Right Trigger on gamepad) the player will attack. This function needs the following variables:

“attackRange” the circumference of the circle where an enemy can be hit,

“attackPoint” this is a transform, which is attached to the player object and will move with them.

“enemyLayers” this layer mask will be used to check for any enemies that have the “Enemy” layer hit in the attack range.

“attackDamage” is how much damage the player will do to the enemy.

“attacking” is used to check if the player is attacking or not.

The author was planning on having a two-step swing where the first swing is a down-swing and the second is an upswing. If attacking back to back the player would do the first attack then the second. But there was not enough time to implement this.

“attackRate” is used for a cooldown on the player attack so the player cannot attack too quickly, and

the “nextAttackTime” variable is used to check if the player can do their next attack.

A screen shot of a computer program

Description automatically generated

When the fire input action (J key on keyboard, Right trigger on gamepad) is pressed, it checks if the current time is greater than the next attack variable. This means the player cannot attack while the attack cooldown is in progress. “Fire()” then uses the “Attack()” function which plays the attack animation, detects the enemies in the attack range and then damages them.

A screen shot of a computer program

Description automatically generated

### Detecting enemies

The “Collider2D[]” array “hitEnemies” stores any enemies that are tagged with the “enemyLayers” layer mask in the “attackRange”.

### Damaging enemies

Running a “foreach” loop iterates through each enemy in “hitEnemies” (the array that stored all enemies hit in the attack range) then the function “TakeDamage()” is used to deal the attack damage variable to the enemy.

A computer screen shot of a program code

Description automatically generated

The “TakeDamage()” function from “Character.cs” takes in a float “attackDamage”. The damage done is taken away from the “currentHealth” variable that stores the current health of the character. This has been updated so that the player and enemies now share this function with inheritance. Once the damage is done, the animation for “Hurt” will trigger. If the health is less than 0 the character will “Die()”

A computer screen shot of text

Description automatically generated

The “Die()” function from “Character.cs” will play the death animation, and start the “Coroutine” “DisableOnDeath()”

A screen shot of a computer code

Description automatically generated

The “DisableOnDeath()” Coroutine from “Character.cs” will wait for a duration for the death animation to finish then it will disable both the “Collider2D” and the “SpriteRenderer”. This will ensure the player cannot collide with or see the object.

A screen shot of a computer code

Description automatically generated

## Move() Function

In the Update function the rigid body (the player) is given a velocity which is a Vector 2. This value is the direction multiplied by the “speed” variable which is 8 by default. Since this is only changing which direction the player moves, the y value is not changed. Text

Description automatically generated

Text

Description automatically generated

## Updated Move() Function

This was moved to the “FixedUpdate()” because the player was sometimes glitching through walls and objects. It is better to deal with physics in the “FixedUpdate()”.A computer code on a black background

Description automatically generated

The “Move()” function was updated because the player should not be able to move while wall jumping until the jump is finished. This code ensures that the player cannot move while wall jumping.

The author also changed the speed to not be an absolute value. While the absolute value made the game unique with slight physics speed-up/slow-down movement, the author preferred having snappy movement over this.

A screen shot of a computer program

Description automatically generated

## Player Jump

As stated in the player movement section, “Jump()” is using Unity’s new Input System. The “Jump()” function takes in an input from the controller or keyboard. When the jump button is pressed, and the player has more than 0 jumps left (a variable stores how many jumps the player has left) a vertical velocity (“jumpForce”) is applied and the player moves up. The “jump” animation is played here. Once the player has jumped the “jumpsLeft” variable is now -1.

If the player cancels the jump (by letting go of the “jump” button) but their vertical velocity is greater than 0 (i.e. they are still moving up), the velocity will be multiplied by .5 which lets the player choose how high they can jump. The player can jump the highest by holding the “jump” button or they can choose to jump lower by letting go early.

Text

Description automatically generated

## Jump() function Updated

The jump function is almost the same but with added functionality. It will now not let the player jump while the game is paused. It will use the “Coroutine” “WallBounce()” if the jump button is pressed while “canWallJump” is true.

A screen shot of a computer program

Description automatically generated

“WallBounce()” is explained in detail in the “WallBounce()” function later in this chapter.

The “jumpsLeft” variable was changed to the more fitting name “jumps”.

When the “jump” button is pressed, “maxJumps” is greater than “jumps” and the player isn’t wall sliding, the jump force is applied to the player’s rigidbody. The “jumps” variable then increases by 1 until the player touches the ground.

A computer screen shot of text

Description automatically generated

When the jump is cancelled while the player is moving up (while the jump is not yet complete) it will make the player slow down its upward velocity and start to fall.

A computer screen shot of text

Description automatically generatedA computer screen with white and blue text

Description automatically generated

## Coyote Time Jump

Coyote time is used to add a short window after the player leaves the ground to still be able to jump. This takes account for human error giving the player a .2 second window to jump after they leave the ground or a platform.

Two variables are needed. One to store how long the error window will be and one that can be changed.

Text

Description automatically generated

In the “Update()” function when the player is on the ground the coyoteTime is assigned to the variable “coyoteTimeCounter”. Once the player leaves the ground the “coyoteTimeCounter” will count down, when the timer gets to 0 or less the player will no longer be able to use their first jump.

Text

Description automatically generated

If the coyote timer is less than or equal to 0 then it will take away 1 jump from the “jumpsLeft” variable. This makes it so the player cant get extra jumps from the coyote timer.

Text

Description automatically generated

To make sure the player can’t jump indefinitely by pressing the jump button fast, the timer has to be set to 0 once the jump button is released.

Text

Description automatically generated

The “Jump()” Function was Updated but the Coyote Timer is the same.

A screenshot of a computer program

Description automatically generated

## WallBounce() Coroutine function or Wall Jump

This Coroutine is performed in the “Jump()” function when the “canWallJump” variable is true. “WallBounce()” will make “canWallJump” false, “isWallJumping”, true and trigger the “jump” animation. It will set the player’s gravity scale to 0 and apply the velocity “wallJumpingPower” to the player. The direction the player is launched is opposite to the direction the player is facing. Then it will wait for “wallJumpingTime” seconds before resetting the player’s gravity scale, changing “isWallJumping” to false and resetting the “jump” animation trigger.

A screen shot of a computer code

Description automatically generated

A screen shot of a computer program

Description automatically generated

A screen shot of a computer program

Description automatically generated

In the “Update()” function the “wallJumpingDirection” is set to the opposite of the current direction, and “wallJumpingTime” is assigned while the player is touching the wall.

This has been updated to the “Player.cs” script “Update()” function.

A computer screen shot of white text

Description automatically generated

### WallJumpCooldown() Coroutine function

This function is used to make sure the first time the player jumps onto the wall and every time after, there will be a cooldown between jumps. In the “FixedUpdate()” it checks if the player “IsWalled()”. If they are, it will start the “WallJumpCooldown()” Coroutine.

A screen shot of a computer code

Description automatically generated

This Coroutine will wait for “wallJumpingCooldown” seconds before making “canWallJump” true again, which allows the player to jump.

A computer screen shot of a code

Description automatically generated

## Double Jump

In “PlayerScript.cs” the “Start()” function sets the current amount of times the player has jumped to the “jumps” variable.

Text

Description automatically generated

In the “Update()” function every time the player is on the ground the “jumpsLeft” variable will reset to the max number of jumps.

Text

Description automatically generated

In the “Jump()” function as long as there is more than 0 “jumpsLeft” the player will be able to jump. So, if “maxJumps” is at 2 the player will be able to double-jump.

Text

Description automatically generated

Once the jump is pressed the “jumpsLeft” will be updated with the new number of jumps left which is 1 less.

Text

Description automatically generated

In the “Update()” function “maxJumps” is set to “jumpsLeft” when the player is on the ground. This means the player’s “jumpsLeft” will reset to whatever value of “maxJumps” when they are on the ground.

Text

Description automatically generated

## Dash() Function

When the dash button is pressed (K on keyboard / button east on gamepad) and the player is able to dash, the “Dash()” coroutine will run. This function uses a trail renderer, a float for the power of the dash, a float for the duration of the dash, a Boolean to check if the player can dash, a Boolean to check if the player is dashing (to inhibit movement while dashing) and a float for the dash cool down.

A computer screen shot of a program

Description automatically generated

### Dash() Coroutine Function

This function sets the “canDash” variable to false to stop multiple dashes while the current dash is activated. “IsDashing” is set to true to stop the player’s other movement abilities. A screen shot of a computer program

Description automatically generated

The “dashing” animation is played, the gravity scale is set to 0, the dashing power is given to the player’s “rigidbody”, and the trail renderer starts emitting. The coroutine waits for “dashingTime” seconds and turns off the trail renderer, resets the player’s gravity scale, animation and “isDashing”. It waits for the cool down to finish before setting “canDash” back to true.

A computer screen shot of a program code

Description automatically generated

## Interact() Function

The “Interact()” function can be improved still but for now it does its job. The author will explain how they will improve it at the end of this section. The button to Interact on Keyboard is “E” and on Gamepad is the north button on the right side. When performed the Function “CheckInteractions()” is used. In the “Player.cs” script, the variables that will be used are:

“interactRange” (the radius of the circle) which surrounds the NPC or interactable object, and

“NPCLayers” a layer mask used to check if the player is overlapping the “interactRange” of any NPC or interactable object.

A black background with white text

Description automatically generated

A screen shot of a computer program

Description automatically generated

### CheckInteractions() Function

This function finds an array of colliders adjacent to the player (Collider2D) and stores them in a variable. It then uses a “foreach” loop to iterate through each “Collider2D” in the array and uses the Unity Function “TryGetComponent()”

The author has Created a C# script “NPCInteractable.cs” which is attached to NPCs. This script will be explained after this section.

If it finds the component, it will use the “Interact()” function from that script.

A screen shot of a computer program

Description automatically generated

## NPCInteractable.cs

This script was created to allow the player to interact with characters, through text, but in the future could be changed to also include object interactions such as opening door or using a lift. Unfortunately, project time constraints do not allow the author to explore this further.

The “Interact()” function in the “NPCInteractable.cs” script checks if:

The dialogue panel is visible or not, if its not visible it will make it visible and start the “Typing()” Coroutine.

The dialogue text is the same as the dialogue index, the “NextLine()” function is used to move the text to the next line, and

if neither of these are used the “ResetText()” is used which removes the text and dialogue panel.

The continue button is usable once the text is finished typing or when the player presses the interact button again.

A screen shot of a computer program

Description automatically generated

The “ResetText()” function is used to remove the current text and make the dialogue panel invisible.

A computer screen shot of text

Description automatically generated

The “Typing()” Coroutine function is used to display a typing effect, it uses a foreach loop to go through each letter in the dialogue array to show each letter as if its being typed. The “wordSpeed” variable will dictate how fast this is.

A screen shot of a computer code

Description automatically generated

The “NextLine()” function will go to the next line of the dialogue. It will remove the current dialogue text and start the “Typing()” function for the next line of text. This has a bug, if the player uses the interact button repeatedly this script will not stop the typing coroutine each time it is interacted with, thus making a jumble of words that are being typed from different parts of the script at the same time. This can be fixed by stopping the “Typing()” coroutine once the “NextLine()” function is used. However, due to the time constraints of this project this will not be fixed by the due date.

A screen shot of a computer program

Description automatically generated

The “OnTriggerEnter2D()” and “OnTriggerExit2D()” functions will check if the player is in range to interact with the NPC, and will reset the text once they leave the radius of the trigger. This prevents the player from having the dialogue box open while not being near the NPC.

A computer screen with text and images

Description automatically generatedA screen shot of a computer program

Description automatically generated

## One Way platforms

The one-way platforms use the variable “fallThrough”. This will be a toggleable Boolean for whether the player can fall through the object or not.

A black background with white text

Description automatically generated

This is currently only working for keyboard. When the player presses the down arrow or “S” key the player will fall through the object.

A computer screen shot of a computer error

Description automatically generated

## OneWayPlatform.cs script

This script will first get the “BoxCollider2D” from the object it is placed on.

A black background with white text

Description automatically generated

When another object first collides with this object it checks for the Tag “Player “.

A screen shot of a computer program

Description automatically generated

Updated the GetComponent to <Player>() from <PlayerScript>() This was the reason the “One Way Platforms” didn’t work when inheritance was being added. The creator did not update this until the end of the project.

A screen shot of a computer code

Description automatically generated

When the player is still in contact with this object and the variable “fallThrough” from “Player.cs” is true, the “Coroutine” “DisableCollision()” will start.

A computer screen with text

Description automatically generated

The “DisableCollision()” function disables the “box” which will turn off the box collider of the object. After half a second the object’s “box” will be enabled again, and the “playerControls” are set back to null to ensure the player does not fall through the object again.

A screen shot of a computer program

Description automatically generated

The object’s box collider must have “Used By Effector” ticked.

A screenshot of a computer

Description automatically generated

The Platform Effector 2D lets the player jump onto the object but only if the player is landing on the object from a specified angle e.g., below image is 90 Degrees. This allows the player the to jump up through the object but not down unless the down button is pressed while in contact with it.

A screenshot of a video game

Description automatically generated

A screenshot of a computer

Description automatically generated

## RespawnScript.cs

This script uses the player position and the respawn point position.

A screen shot of a computer program

Description automatically generated

This script is attached to the “RespawnCollider” object whose box collider’s “Is Trigger” is checked.

When the player enters the box collider of “RespawnCollider” the “OnTriggerEnter2D()” function moves the player’s current position to the respawn point’s position.

A screenshot of a computer

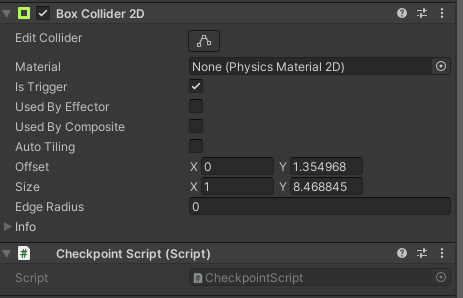
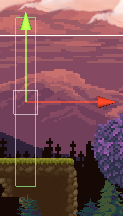
Description automatically generated

## CheckpointScript.cs

This script uses a reference to the “RespawnScript.cs”. The “Awake()” function runs as soon as the script is started, it gets the component of the “RespawnCollider” and now the respawn point can be updated to a new point.

## A screen shot of a computer program Description automatically generated

These checkpoint objects are placed throughout the level and use a Box Collider 2D with the “Is Trigger” checked.



## PauseScript.cs

This pause script is put on the empty menu controller object.



A reference to “playerControls” is used.

The variables used in this script are:

The GameObject “PauseMenu”, and

“isPaused” which will enable / disable actions depending on its state.

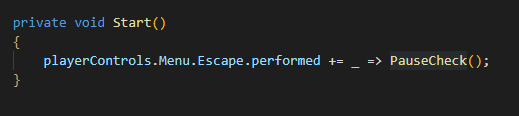
The “isPaused” variable is a “public static bool”. This is to make sure it is accessible to other classes.

A screenshot of a computer screen

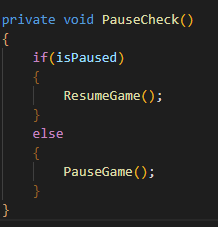
Description automatically generatedA black screen with white text

Description automatically generated

When the player presses the escape key, the “PauseCheck()” function will run.



“PauseCheck()” will check if the “isPaused” variable is true or false. If it is true it will run the “ResumeGame()” function, otherwise it will run the “PauseGame()” function.

A screen shot of a computer program

Description automatically generated

“PauseGame()” sets the timescale to 0, stops the game sounds, sets “isPaused” to true, and enables the Pause Menu UI.

A computer screen shot of a program code

Description automatically generatedA screenshot of a computer menu

Description automatically generated

“ResumeGame()” sets the timescale back to 1, enables the game sounds, sets “isPaused” to false and disables the Pause Menu UI.

A computer screen shot of a program code

Description automatically generatedA screenshot of a menu

Description automatically generated

The Menu Input system using Unity’s new input system.

A screenshot of a program

Description automatically generated A screenshot of a computer

Description automatically generated

## EnemyScript.cs

All of the functions in this script have been explained previously in the “Fire()”. This script is placed on enemies. It gets their rigid body and animator components, and their “maxHealth” can be changed in the inspector. This script will be combined with the “PlayerScript.cs” once Inheritance is implemented as they both should use all these functions and variables.

A computer screen with text

Description automatically generated

## MainMenu.cs

This Script has two functions: “PlayGame()”and “QuitGame()”. This script is used in the starting menu. “PlayGame()” uses the scene manager to load the first scene and “QuitGame()” closes the application.

## A screen shot of a computer program Description automatically generated

## SettingsMenu.cs

In the settings menu the player can change the audio or video. They can change resolutions and graphics, or SFX volume, Background music, and Master volume.

This script uses an array of resolutions, the “Text Mesh Pro” dropdown button, and an audio mixer. A black screen with white text

Description automatically generated

### SetResolution()

This function uses an array of resolutions, and a reference to the dropdown box. It will get the resolutions that the screen can use and display them in the dropdown box.

A screen shot of a computer code

Description automatically generated

In the “Start()” function the resolutions array variable stores the screen’s possible resolutions. The dropdown box is then cleared of the current options.

A screen shot of a computer program

Description automatically generated

The “AddOptions()” Unity function takes in a list of strings, not an array of resolutions. The array of resolutions needs to be formatted into a list of strings.

A for loop goes through the elements in the resolutions array. The option string gets the width and height of the [i] element in the resolutions array and then it is added to the options list.

A computer screen with text and symbols

Description automatically generated

The options list is then added to the resolution dropdown.



To check if the resolutions match with the screen’s current resolution the if statement will update the current resolution index.

A screen shot of a computer

Description automatically generated

The resolution value is set, and the value is refreshed so the correct value is shown.

A screen shot of a computer program

Description automatically generated

When the dropdown value is updated the “SetResolution()” function is used.

A screenshot of a computer menu

Description automatically generated

It uses the “Screen.SetResolution()” function to make the screen resolution equal to what was selected.

A screen shot of a computer code

Description automatically generated

### SetQuality() function

This function will change the quality of the game to fit the selected value.

The “qualityIndex” variable is 0 for Low, 1 for Medium and 2 for High. When accessing the “QualitySettings” the quality is set to the value selected.

A screenshot of a video game

Description automatically generatedA screen shot of a computer program

Description automatically generatedA screenshot of a menu

Description automatically generated

### SetVolume()

This function takes in a float variable.

A computer screen shot of text

Description automatically generated

In the audio mixer the value of the master volume is called “mastervolume”. Using the “SetFloat” function above, the master volume becomes the float value passed into the “SetVolume()” function.

A screenshot of a computer

Description automatically generatedA screenshot of a video game

Description automatically generated

Using the slider “On Value Changed” augment, the value on the slider is passed into the function.

A screenshot of a computer

Description automatically generated

The minimum value of the slider is -80 and the max is 0 matching with the Master Audio Mixer.

A screenshot of a computer

Description automatically generated

## Updated SetVolume() Saves between scenes.

An “AudioManager.cs” Script was created to handle audio in the scene.

“SetVolume()” has been broken up into 3 separate functions:

“SetMasterVolume()”, “SetMusicVolume()” and “SetSFXVolume()”.

The Variables that control the mixer groups were turned into const variables, the naming convention for const variables are all capital letters. These values will not change.

A black screen with white text

Description automatically generated

In the “Awake()” function, when a value on the slider is changed, the Set x volume functions are used for whichever slider was changed.

A screen shot of a computer program

Description automatically generated

The math equation here is used to change the volume logarithmically as the volume on the mixer changes logarithmically. The slider value was also changed to fit this version, from -80 to 0 to between 0 and 1. The unity function “SetFloat()” changes the Audio Mixer’s variable (“MASTER\_MIXER”) to the value of the math equation.

A screenshot of a computer program

Description automatically generated

A number on a black box

Description automatically generated

In the “OnDisable()” function the “SetFloat()” function will be used to store the values to the “PlayerPrefs” which is player preferences that are stored on the computer. The “AudioManager.MASTER\_KEY” is the variable name for the master volume mixer, and the slider value is “masterSlider.value”.

This is saving the sound details to the player preferences.A computer screen shot of a program

Description automatically generated

In the “AudioManager.cs” script the settings are loaded, since an “Audio Manager” is going to be in every scene.

In the “LoadSound()” function the value of the float that’s stored in player preferences is acquired with the Unity function “GetFloat()”. If it can’t find a value for “MASTER\_KEY” it will default to 1 (100% volume). The volume of each mixer is then set with “SetFloat()”, and the value is put into the logarithmic equation to get the correct value.

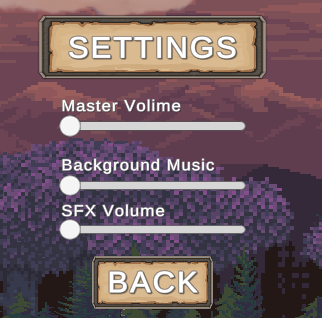
A computer screen shot of a program

Description automatically generated

The “AudioManager” object has the “AudioManager.cs” script on it. Once the scene loads it will check if there is an instance of “AudioManager”. If there is not, it will set an instance that will not be destroyed when scene is loaded. If there is an instance, it will destroy the game object to make sure there is only 1 “AudioManager” per scene.

A screen shot of a computer program

Description automatically generated



The groups of this mixer are set up so the Master controls both the Music and SFX sounds but they don’t have any impact on each other.

A screenshot of a music system

Description automatically generated

## Inheritance

Inheritance is used to ensure there is no repeat of code. This can happen when two or more classes use the same or similar functions. In this situation the “Character” class is the parent class, and the “Player”, “NPC”, and “Enemy” classes are the child classes that inherit from “Character”. For example, all characters will have health, speed, attack damage, and jump force. Instead of making a Player class with these variables and then creating an NPC and Enemy class with the same variables, the “Character” class exists and can give its child the information. This works the same with functions not just variables.

### Character.cs

This class will be used to hold shared functions and variables that its child classes use. All child objects will need to turn around so the “Flip()”, and “ChangeDirection()” functions are here.

A screen shot of a computer program

Description automatically generatedA screen shot of a computer program

Description automatically generated

All characters should be able to take damage and die so the “TakeDamage()”, “Die()” and “DisableOnDeath()” functions are here.

A screen shot of a computer program

Description automatically generated

All characters should be able to attack. However, for this project the author only has time to have Enemy and Player attack. This function would be an abstract void function instead of virtual void, because each of its child classes would attack differently.

A black screen with white text

Description automatically generated

Each character should not be able to move while the game is paused. The “Update()” and “FixedUpdate()” functions do this. In the future the creator wants the player to have allies to fight alongside. However, for the scope of this project it is not possible. The “Start()” function will set each of the character’s health and speed and make sure they are not attacking.

A screen shot of a computer program

Description automatically generated

The “OnDrawGizmos()” function draws wireframes that are visible in the editor and not in the game. This is helpful to see where the attack points / range etc of characters are.

A screen shot of a computer program

Description automatically generated

### Player.cs

#### Player Variables

Below are the variables the player uses under the headings.

A screen shot of a computer program

Description automatically generatedA computer screen shot of text

Description automatically generated

The player will need to check if they are on the ground or against a wall so the “IsGrounded()” and “IsWalled()” functions are here.

A screen shot of a computer program

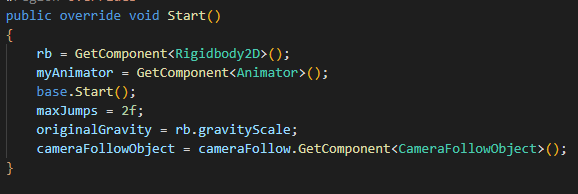
Description automatically generated

#### Function overrides

The functions that use the override word will be able to use the “Character” classes version of the function and add onto it or have a new function with the same name.



The functions that are overridden in “Player.cs” are:

“Start()”, 

“Update()”

Update is shown and explained previously in the “Player.cs Update()” function.

“FixedUpdate()”

Fixed update is explained previously in the “Player.cs FixedUpdate()” function.

A screen shot of a computer code

Description automatically generated

“Attack()”

Enemies and the player will have different attacks. Below is the player’s attack. This is also explained previously in the “Player.cs Attack()” function.

A screen shot of a computer code

Description automatically generated

“TakeDamage()”

Each character has a different animation. This function adds the animation for the character to the “TakeDamage” function.

A screen shot of a computer program

Description automatically generated

“DisableOnDeath()” adds the player death animation.

A screen shot of a computer program

Description automatically generated

“Flip()” makes the dash go in the correct direction, since the new way of flipping the player doesn’t use the player’s X scale anymore.

A computer screen with white and blue text

Description automatically generated

“OnDrawGizmos()” is used to show the interact range of NPCs.

A screen shot of a computer program

Description automatically generated

#### Animation Control

The “PlayerAnimations()” function could be improved but the time constraints of this project did not allow a rehash of what was already done with this section. It is used to make sure the player is using the correct animation.

A screen shot of a computer program

Description automatically generated

#### Player Input

The player input classes explained earlier in the chapter are placed in the “Player.cs” class:

“Move()”, “Fire()”, “Jump()”, “Interact()”, “Dash()”, and “CheckInteractions()”.

#### Coroutines

The coroutines have been explained previously. The “WallJumpCooldown()”, “WallBounce()”, and “Dash()” coroutine functions are placed in this class.

#### Extra functions

The “WallSlide()” function is placed in this class

### NPC.cs

\*\*\*\*\*\*\*\*\*\*

### Enemy.cs

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

## SceneController.cs

This script is attached to the “GameManger” object. This object holds the “AudioManager”, the “Canvas”, the “MenuController”, and the “EventSystem”. These are necessary in each scene.

A screen shot of a computer program

Description automatically generated

When a scene is loaded, the “DontDestroyOnLoad()” function keeps the object from the previous scene. If there is another object with this script attached, the object will be destroyed.

The “SceneController.cs” script also contains the functions:

“NextLevel()” which loads the next scene.

“LoadScene()” which loads a scene by name, and

“MainMenu()” which sends the player to the main menu.

A screen shot of a computer program

Description automatically generated

## Tile map

Tile maps are used to create 2D worlds. They allow the creator to select the tile they would like to place and draw the tiles into the scene.

A screenshot of a video game

Description automatically generated

A screenshot of a video game

Description automatically generated

## Level Creation

### The Crystal Village 1

A video game with trees and clouds

Description automatically generatedA video game screen with houses and trees

Description automatically generated

### The Crystal Village 2

### The Crystal Mines 1

A video game with a purple background

Description automatically generated

A video game screen with purple and grey squares

Description automatically generated

A video game screen with a pixelated area

Description automatically generated with medium confidence

A screenshot of a video game

Description automatically generated

### The Crystal Mines 2

## Sprite sheet

The player’s sprite sheet has been downloaded from the Unity assets store. This came with different animations that weren’t set up properly. This has been fixed by re-selecting the animation frames that combine.

Graphical user interface, text, application

Description automatically generatedGraphical user interface, text, application

Description automatically generated

## 

## Camera

The camera plays a huge part in Metroidvania games. The package the author uses to manoeuvre the camera is “Cinemachine”.

### Cinemachine settings

This camera is used in open rooms. The author had planned to have multiple cameras with different settings, such as remaining static, showing only a specific area or not moving on a certain axis. However, the time constraints of this project did not allow for this.

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

The camera boundary is comprised of multiple 2D box colliders and a single 2D composite collider. The composite collider has its “IsTrigger” box ticked. A composite collider is used to combine every collider on the game object. The 2D box colliders must have the “Used By Composite” box ticked.

## A screenshot of a video game Description automatically generated

A screenshot of a computer

Description automatically generated

## Parallax

The author uses Parallax to give depth to the 2D game they are creating. The image below is an example of how to give the illusion of depth to the game. In the diagram below, the layer labled “A” is the player’s layer. In the Parallax.cs script the background behind the player layer will move as the player moves depending on how far away from the player layer they are.

### A screenshot of a video game Description automatically generated

### Parallax.cs

This script uses the “Main Camera” object, and the player’s transform to determine how the background scene will move.

A screenshot of a computer

Description automatically generated

The “Start()” function assigns the startPosition and startZ variables.

A screen shot of a computer program

Description automatically generated

The “Vector2” variable “travel” is calculated each time the variable needs to be used. It holds the distance the camera has travelled from the “startPosition”.

The float “distanceFromSubject” is calculated each time it is used. This distance is along the Z axis because the further behind the “subject” the background is, then the less it should move.

The float “clippingPlane” is calculated each time it is used. If the “distanceFromSubject” variable is greater than 0 (it is behind the subject) add the z position of the camera to the “cam.farClipPlane”. Otherwise add the z position of the camera to “cam.nearClipPlane”

To get the parallax factor, the float “parallaxFactor” is used and calculates what it is by dividing the “distanceFromSubject” by the “clippingPlane”.

A computer screen shot of a program code

Description automatically generated

In the “FixedUpdate()” function, a new Vector 2 called “newPos” is used to calculate where the new position should be. Then it is moved using a “transform.position”.

A screen shot of a computer

Description automatically generated

## Animation

In the “PlayerMvt.cs” script “Update()” function, - Updated to “Player.cs Update()” - when the player is on the ground the falling Boolean is set to false. This will stop the fall animation from continually playing after the player falls.

Text

Description automatically generated

Once the player’s velocity is less than 0, the fall animation is played, and the jump trigger is reset so the player can jump again. The nested if statement checking if “IsGrounded()” is true to stop the falling animation if the player is still holding one of the movement buttons.

In “PlayerMvt.cs” in the “Jump()” function - Updated to “Player.cs Jump()” - when the player presses the “Space” button, the trigger variable jump is toggled and the Boolean variable “falling” is set to false. This will play the “Jump” animation. When the “Space” key is released the “Jump” trigger is reset and the “falling” variable is set to true.

Text

Description automatically generated

The “Idle”, “Jump”, “JumptoFall” and “Run” animations all work.

## A picture containing graphical user interface Description automatically generated

When the user doesn’t press any buttons and isn’t falling, the idle animation will play. If the user presses the “A” or “D” key, the run animation will play. If the user presses the “jump” key the jump animation will play.

From “Idle” to “Jump”, a trigger condition is needed. This checks when the parameter “jump” is toggled.

Graphical user interface, application

Description automatically generated

Once the user reaches the peak of their jump, stops, or has a negative y velocity, the “JumpToFall” animation will play, and the variable “falling” is set to true.

Graphical user interface, application

Description automatically generated

Once the Character lands on the ground “falling” is set to false and will no longer play the “JumptoFall” animation.

Graphical user interface

Description automatically generated

If the user falls off a platform their y velocity is negative the “JumptoFall” animation will play.

A screenshot of a computer

Description automatically generated with low confidence

If the user presses the “A” or “D” key to move, the “Run” animation will play. When checking if speed is greater than 0.0001 is making the response quicker so the character will react faster to the input of the user.

Graphical user interface, application

Description automatically generated

If the user stops pressing the “A” or “D” keys the “Run” animation will stop playing. When checking if speed is less than 0.0001 the player will stop faster.

A picture containing accessory, case

Description automatically generated

The graph below has been updated so that the “JumpToFall” will go back to “Idle” at the correct time. The player can now go from “Run” to “JumpToFall”. This means the player isn’t running in mid-air or falling while moving on the ground.

A screenshot of a computer

Description automatically generated with low confidence

Updated Player animator below.

A screenshot of a computer

Description automatically generated