Individual Implementation Report for Quinn's Escape

Josh Shepherd

Contents

Introduction

Quinn's Escape is a 2.5D video game built inside Unreal Engine 4.22. It is a single player game where the player takes control over Quinn, a mannequin who is stuck inside of an abandoned clothing shop. After being abandoned for many years, the shop has surprisingly come to life, Quinn included. However, not all of the life in the shop is friendly, as Quinn wants to escape from the shop to safety. As Quinn, the player must progress through the shop, discovering the new life in the shop and potentially fighting them. The shop is dangerous as Quinn needs to escape as fast as possible, collecting any hidden treasures or coin he finds.

The main aim of the game is for the player to escape as fast as possible, while gaining as much score as possible. The faster the player solves puzzles, collects coins, defeats enemies, and escapes the level, the more points, and bonuses they will achieve, putting them at the top of their own high score leader board.

In this document, I will cover how I implemented aspects of the game, such as the player, their controls, the world design, and creation, and more. Each section contains a detailed description of each element I implemented or assisted on, including diagrams where relevant, as well as pseudocode to assist any explanations.

Game Contributions

For the game, I feel I contributed to a lot of the game, covering many aspects of the game and being able to build and develop the game quickly over a short period of time.

AHealthCharacter – Character Shared Base Class

To best create the game's characters, I created a base class that inherits from `ACharacter` which allowed me to add on additional functionality that was required for each character in the game, from the main player character to the boss and individual NPC characters. The class contained functionality related to:

- Life of the character (Get/Set)
- Projectile Collision Detection
- Removal of health from any collided projectile
- Sounds: Hit, Jump and Death
- Character Death event, broadcast when no life remains
- Score hit and killed amount (Gets/Sets)
- Stomp ability detection/collision

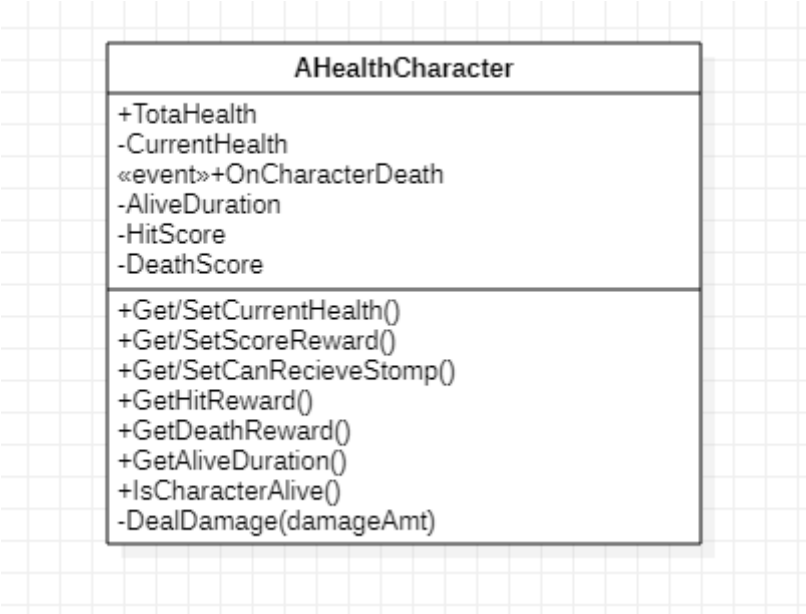

Figure 1: UML diagram of `AHealthCharacter`

The class is also outlined, but not fully, in the UML diagram in figure 1. The class contains its properties and variables related to these elements but are kept private. However, the class' getters and setters can be used by either an inherited class or any other class to modify the functionality of the character, allowing this base class to be flexible and modified to any inherited class' desire.

The `AHealthCharacter` class keeps track of how long a character has been alive, however only allows a get access since this is incremented privately in the `Tick()` function.

The class also contains related code for tracking how much score should be given when the character is hit by a projectile and when the character dies. The attached capsule component inside `ACharacter` is used to listen to any overlap events from a projectile.

Player Character & Control

The player character, Quinn, and the related player controller were implemented in the regular style for Unreal Engine. That being, a separate controller class, that contains functionality related to controlling the character, and a separate character class, containing properties to modify the character and functions for using the character's abilities.

Quinn Controller

The player controller inside Unreal Engine encompasses any input from the player and calculations to convert to being used by the character. The controller simply contains the code that listens to key presses that occur through the Unreal Engine input system. I configured several actions, which are when a button is pressed and released, as well as one axis for moving the character. The axis and action's I configured for the game are listed below, with their relevant code.

- "Fire" action: On Key "Left Mouse Button" released; determines where the mouse location is. Calculates the fire direction from Quinn character's location to the mouse location. Calls the `FireProjectile(directionVector)` function on `AQuinnCharacter` and provides the direction vector to fire in the correct direction.
- "Slam" action: On Key "S" released; calls the `SlamGround()` function on `AQuinnCharacter` to force the character to the ground
- "Jump" action: On Key "Space" pressed and released; calls the `ACharacter::Jump()` and `StopJumping()` functions.
- "MoveRight" axis: Calls the `MoveRight(axisDelta)` function on AQuinnCharacter to provide movement input to the character

The controller also frees the user's mouse to allow for aiming on the screen during gameplay.

Quinn Character

The main character's class `AQuinnCharacter` inherits from the base `AHealthCharacter` class which allows for the reusability of health, collision detection and more. The `AQuinnCharacter` class adds an array of properties used to tune the player character to our design, which also can be changed and edited in blueprints. The properties can be seen in the UML class diagram in figure 2.

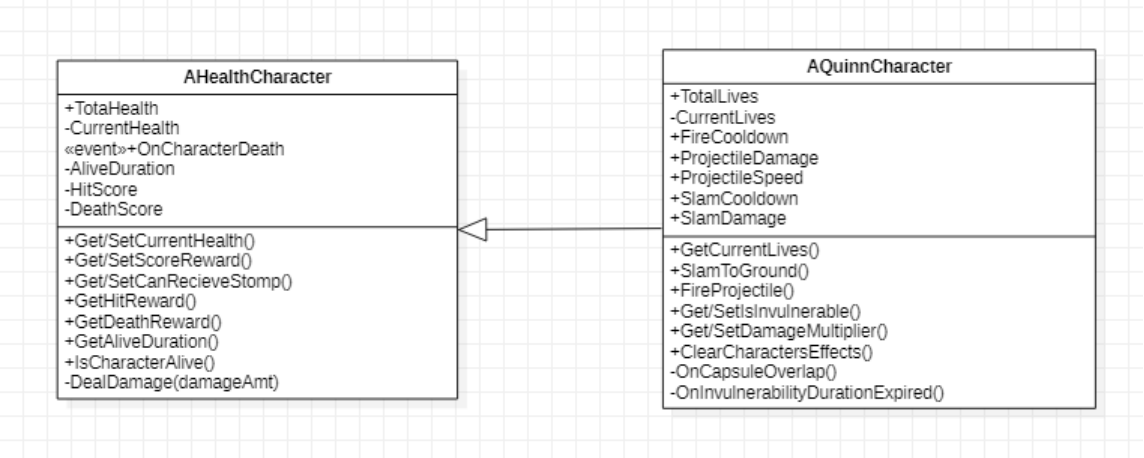

Figure 2: UML diagram of `AQuinnCharacter`

The class overrides the base capsule component's overlap event and triggers when any actor overlaps the character. Specifically, the Quinn character listens for when the character collides with a checkpoint and stores a reference to it. This is so when the character runs out of health, the character can be teleported to the last checkpoint's location.

On death, a life is removed from the player. This functionality also exists inside of the Quinn character since only this character has multiple lives. Once all lives run out, the Quinn character is dead which results in a game over ending.

The class also adds the "Slam" ability, which adds a downward force to the character. The slam ability has a small cooldown period, which can be adjusted by its property, which prevents the player from spamming the ability.

`AQuinnCharacter` also implements some functionality needed for the game's power-ups. The class can set the character to be invulnerable to damage, either constantly until set to false or for a certain duration.

Dead Zone Player Camera

As Quinn's Escape is a side-scrolling 2.5D game, the camera is required to be locked to a horizontal axis. For this, I created a separate camera that follows the player, but allows for a certain region in the middle to act as a dead zone. The dead zone allows the player to move within the box region without the camera following. The camera will only follow the player once they reach the edge of the zone. The dead zone can be seen in figure 3.

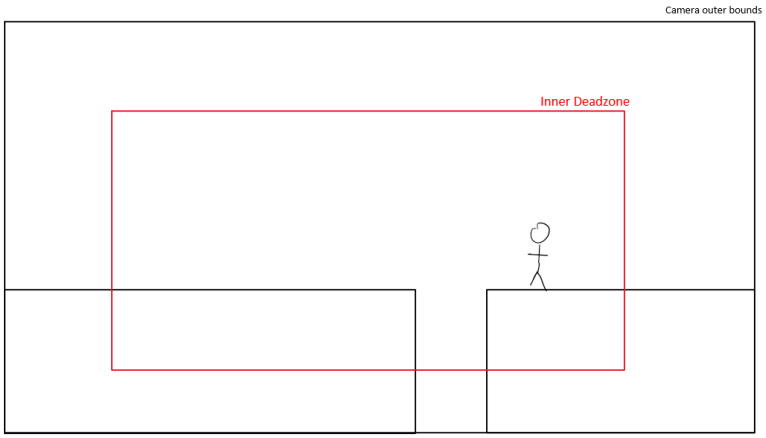

Figure 3: Deadzone camera and its inner dead zone

Projectiles

All projectiles share and use the same class, since the functionality behind them is very simple. The projectiles come with public functions to customize how that projectile will fly through the world. The class is fully outlined in the UML diagram in figure 4.

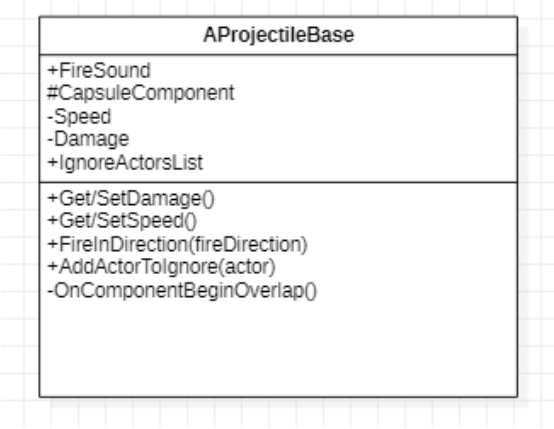

Figure 4: UML diagram of the projectile class

The projectile contains a capsule component that listens to the `OnComponentBeginOverlap` event. Once triggered, the listener function determines if the overlapped actor is not within a list of ignored actors and if so, deals its damage amount to the overlapped actor is possible and destroys itself.

Gameplay

Checkpoints

As Quinn's Escape level is long and contains many enemies, the player's progress can be saved at checkpoints. A checkpoint works by allowing the player to pass through the checkpoint to allow them to respawn at that location if they ran out of health. The checkpoints contain a simple box collision that listens for when the player overlaps with the checkpoint. The checkpoint also has a mesh component to allow for a visual representation of the checkpoint. When the player does trigger the checkpoint, the flag plays a raising animation by rotation the flag model by -90 degrees. The checkpoint contains a point reward that is given to the player for passing through and also plays a success sound.

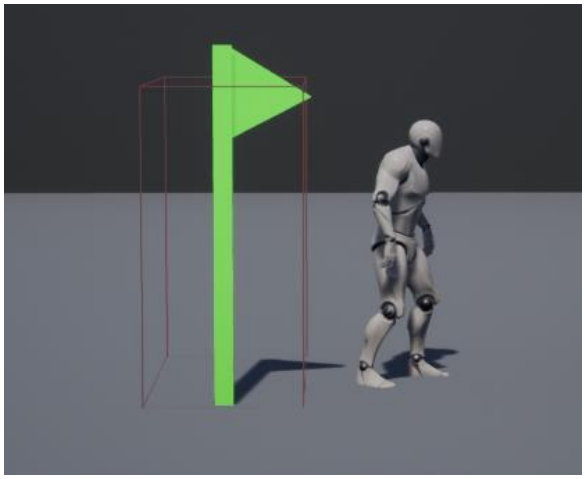

Figure 5: A checkpoint and its box collision, with Quinn's player next to it.

Puzzles

The game contains two puzzles for the player to solve and get rewarded with score, as well as a power-up if one is within the puzzle. For a puzzle to function, it required a series of classes and elements that work together to build the functionality.

- `PlayerEscapeTrigger`: A trigger that opens the escape door if the player collides with it
- `ProjectileEscapeTrigger`: Trigger that opens the escape door if a player's projectile collides with it
- `PuzzleAccessTrigger`: Trigger that teleports the player to an entrance target point and locks the camera to a specific Z value.
- `PuzzleEscapeDoor`: Door that can be opened. When closed by default, blocks the exit to the puzzle until opened by a trigger. A sound can be set and played when opened.

To get to puzzle one, the player falls down a red pipe in the level. The pipe contains a `PuzzleAccessTrigger` that is set to lock the camera Z to a certain value and teleport the player to the entrance. In the puzzle, the player must jump on small platforms to reach a `ProjectileEscapeTrigger`, which opens the escape once shot. The score is rewarded to the player and the door opens, which contains a `PuzzleAccessTrigger` that teleports them out of the puzzle and frees the camera.

Figure 6: Puzzle one, with the entrance and exit, and a `ProjectileEscapeTrigger` in the top right

Puzzle two is configured very similar to puzzle one. A `PuzzleAccessTrigger` is used to enter and exit the puzzle and lock or free the camera. However, this puzzle requires the player to hit a series of boxes with their head in a specific sequence. For this, I created a `PuzzleSequenceBox` class that contains a box index, as well as displaying that index in the world. I also created a `SequencePuzzleController` class that can be configured to hold a specific sequence of numbers.

The `SequencePuzzleController` listens to each `PuzzleSequenceBox` in the puzzle for when it is triggered. Once the player triggers a box, the index is tracked and stored in the controller. If the correct index is used, then the controller tracks and continues listening until the correct sequence is entered. Once the player guessed the correct sequence, the `PuzzleEscapeDoor` is raised and allows the player to escape. If the player guessed the incorrect index, the tracked list of indexes is reset and starts again until the correct index is triggered again. Once the correct index is triggered by the player, the index highlights green to indicate it is correct. If not, all indexes are reset back to white.

Power-ups

As the player progresses through the level, they will find special boxes that are indicated with a "#" character. These are power-up boxes that are triggered by the player hitting the bottom of the box with their head. I created a class that inherited from the shared trigger base class `AHeadHittableBox` to be able to get the functionality behind being able to hit and trigger a box with the player's head. Once listening to the `OnBoxRecievedHit` event, which is triggered when the player hits it with their head, the power-up box is able to spawn a power-up above the box. As there are a range of power-ups, the box has a list of power-up classes that can be set in the editor that will be created and spawned when required. Once triggered, the box chooses a random power-up in the list and spawns it above the box for the player to pick up.

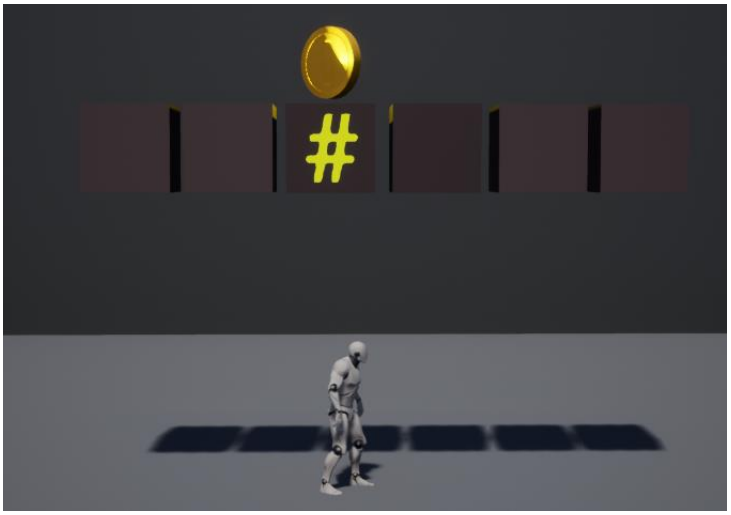

Figure 7: Player below a power-up box that spawned a coin power-up

The game contains a total of 4 power-ups that all share the same base class, `APowerupPickup`. The base class contains the main code behind the power-up, such as being able to collide, play the floating animation, as well as provide a special call back for applying the power-ups effects. The `APowerupPickup` also contains a mesh for a visual representation to the player, as well as a box component for collision detection and a light for emitting the power-up's colour and making it noticeable and special to the player.

As there are 4 power-ups, each power-up is a separate class that inherits from the base `APowerupPickup` class. However, the code inside of these classes is minimal as the only code needing to be run is applying any effects of the power-up. Below is a list of the power-ups in the game:

- **Coin Bonus:** Provides the player with an instant score bonus, and applies a coin multiplier, if set in the editor.
- **Bonus Damage:** Gives the player a damage multiplier for a certain duration. Able to set the duration and multiplier in the editor.
- **Extra Life:** Provides the player with an extra life, else rewards them with an instant score amount. Able to set the number of lives power-up provides, as well as the instant score amount.
- **Invulnerability:** Gives the player invulnerability (immunity to damage) for a certain duration of sections. Able to set the second duration in editor.

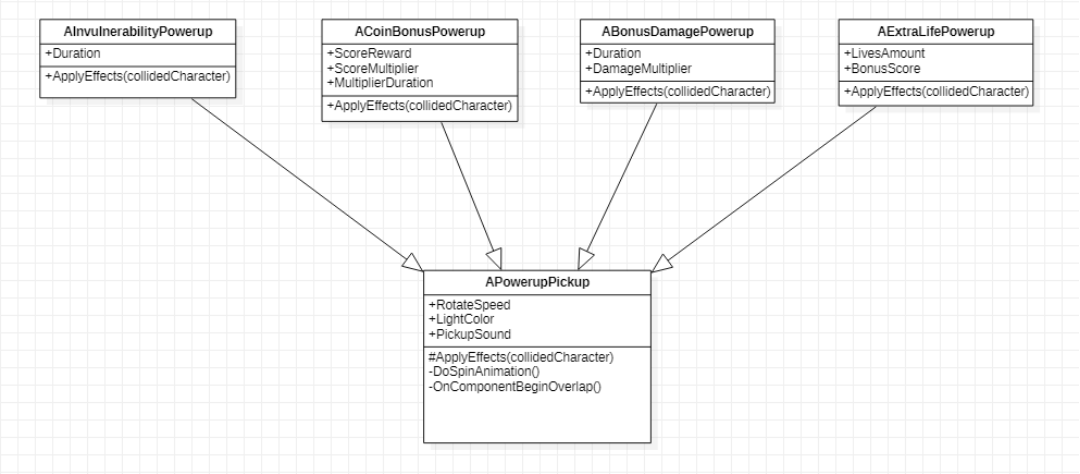

Figure 8: UML diagram of inheritance between power-ups

Shooter Character & Controller

I also implemented one of the enemies in the game, the shooter enemy. The shooter enemy is a simple character that is very similar to the player's character, Quinn. The shooter enemy's code is split between a controller and a character.

Controller

The controller of the shooter enemy uses a finite state machine (FSM) to handle its state and run code. The `AShooterFSMController` inherits from the base `AAIController` inside Unreal Engine to inherit code for pathfinding and other useful methods. The FSM is implemented as per the design of the shooter enemy, that is the enemy having 4 states: idle, chasing, shooting and death. The states are implemented in an `EShooterStates` enumerator that contains all of the potential states. Each state is able to have `Begin()` and `Tick()` to their states to help run relevant code to their actions.

The controller is hard coded to be used inside of the `AShooterEnemy` class and will always possess the shooter character.

Character

For the shooter character, I created the class `AShooterEnemy` that inherited from the base `AHealthCharacter` to be able to inherit the base collision, health removal and death that has been previously implemented for the Quinn character. Since the character is simple, the only code needing to be implemented was the ability to fire its own projectile and create public properties that adjust the character's behaviour and can be changed in the editor.

I created public variables and labelled them as `UPROPERTY(EditAnywhere)` which allowed for changing the behaviour of the character from the inheriting blueprint class. In the inherited blueprint, we could also set the character's mesh and materials to differentiate it as an enemy.

For the shooter character to be able to constantly shoot at a target while in range, I set up the character to have the public methods of `BeginShooting()` and `StopShooting()`. Internally, the shooting works as a timer with a delay set in a public variable. Every custom duration, the timer would run and shoot a projectile towards the Quinn character. The `AShooterFSMController` is able to call the public functions to enable or disable shooting when in the correct state.

User Interface

All user interfaces in the game all share a same base class, called `AQuinnUIWidget`. The class inherits from the required `UserWidget` class in Unreal Engine and adds additional functionality to access the relevant values for the main gameplay user interface, as well as performing general code for interfacing with the save game to retrieve high scores.

Main Menu

When the player first starts the game, they are greeted with the main menu, which is simple and allows the user to either "Play" or "Quit". The menu also displays a simple high score list to the right side, as seen in figure 9.

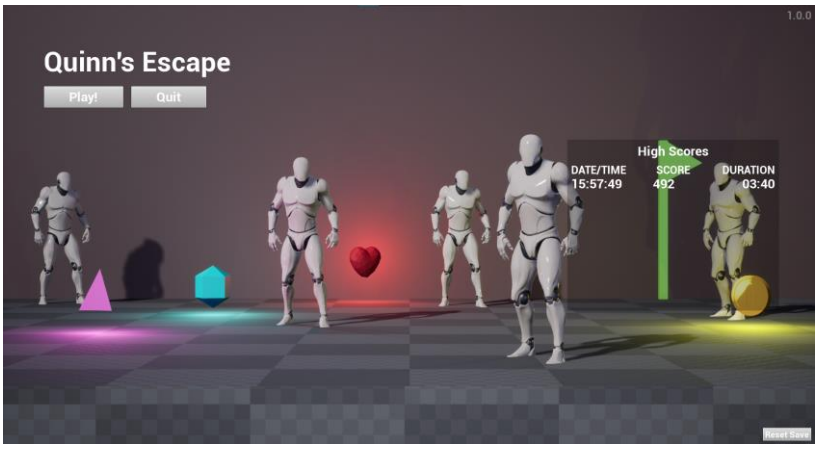

Figure 9: Main Menu of Quinn's Escape, displaying the high scores and buttons

The main menu inherits from the same User Widget base class `AQuinnUIWidget` to get access to the save game. On initialization, the high score widget simply retrieves the current high score list and populates the interface for every entry, displaying it to the user. For the "Play" and "Quit" buttons, the blueprint simple listen to the on clicked event for each and calls the `OpenLevel(levelName)` or `QuitGame()` functions. Since these are simple, these were implemented in blueprints to save time.

Gameplay Interface

The gameplay interface shows three main elements: the player's score and multiplier, Quinn character's health and Quinn's remaining lives. Similar to the main menu, I created a blueprint that inherits from the base C++ class, `AQuinnUIWidget`. The blueprint simply accesses the relevant functions implemented in C++ every tick to update the current health, lives, score, and multiplier in real time, as seen in figure 10.

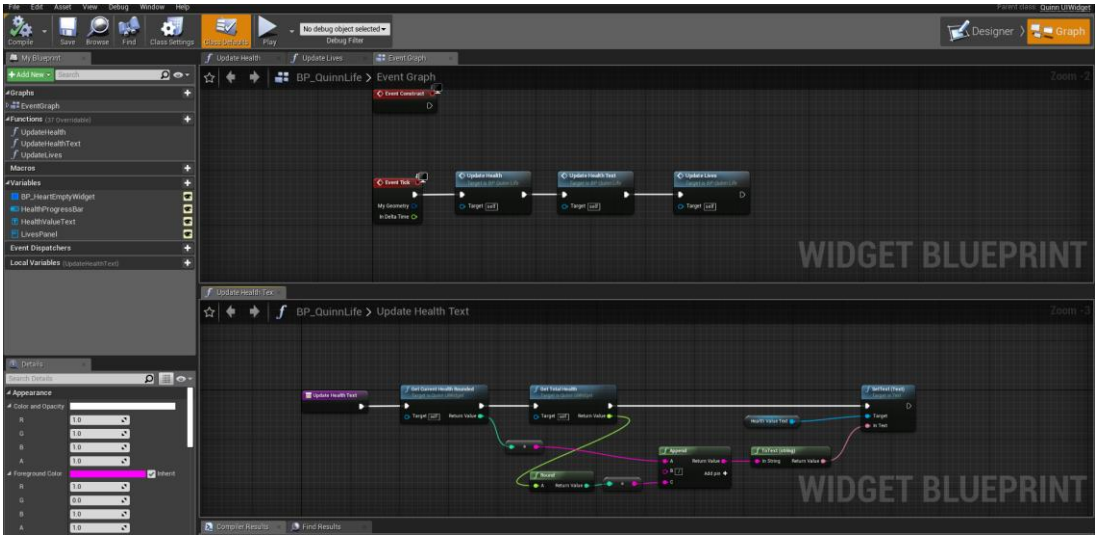

Figure 10: Blueprint for displaying and updating health in real time

The implemented game mode also listens to the player and final level complete events to display the relevant interface. Once the player dies and loses all life, the game over screen is displayed, containing buttons returning to the menu or retrying the level. Once the level complete event is fired, the game complete interface is displayed, which contains the players current score, time, and the current high scores. The menu also contains a button to return to the main menu, completing the application flow. The functionality in these menus, again, utilizes the `AQuinnUIWidget` functions for retrieving previous high scores, and current high score and times.

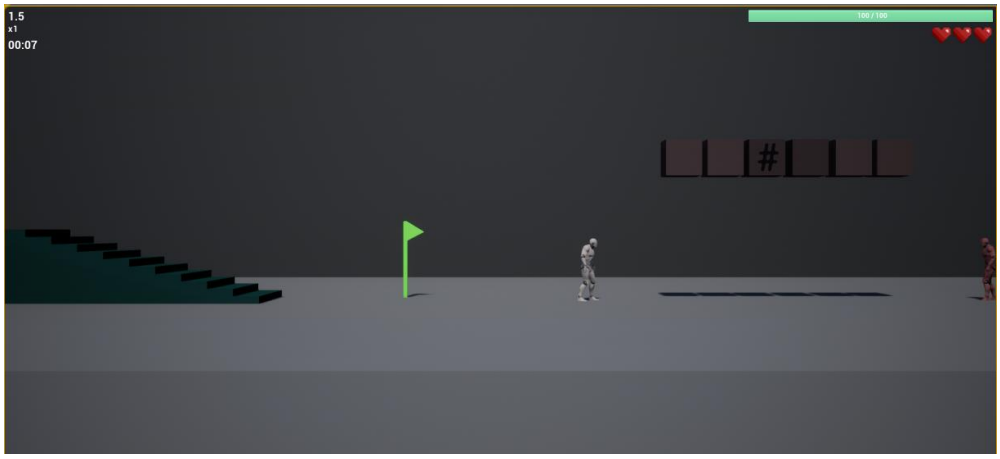

Figure 11: Gameplay user interface, showing score, multiplier, time, health and lives

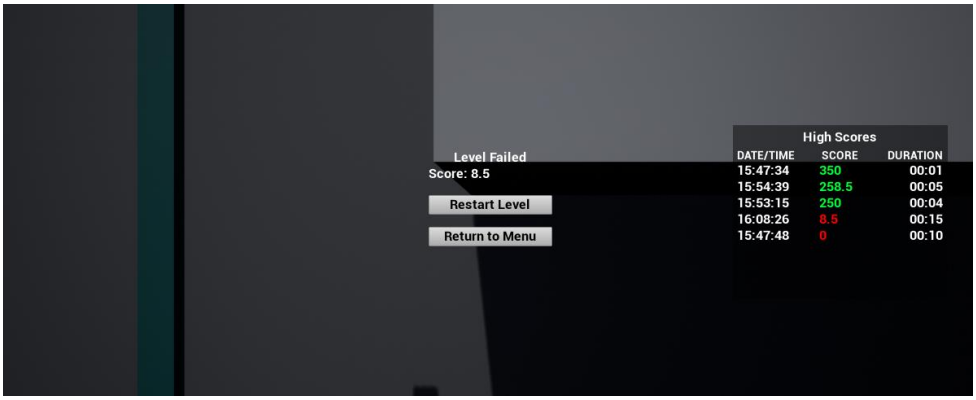

Figure 12: Level failed user interface

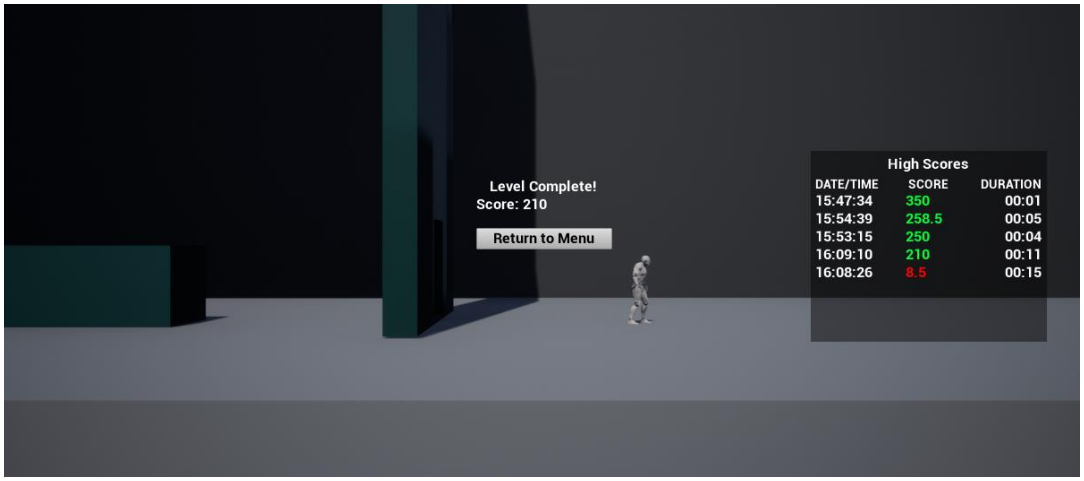

Figure 13: Completed level user interface

Audio

Each character in the game includes their own game sounds that get used during certain important events, such as getting hit. Each sound is implemented as a public `UPROPERTY` which allows for a sound to be set in a parent blueprint in the editor. This allows for easily swapping out the sounds with new ones when we wish. The characters will mainly use a sound when they: get hit and receive damage, once they have no health and die, when a projectile is fired, and when they jump. However, to prevent sound overload, when an enemy character receives a hit, their hit sound will have a small period of cooldown before being played again.

To differentiate between the player's character and enemy characters, each type has their own sounds. For example, the player's hit sound is different from the enemy character's hit sound. With this, it helps the player to tell the difference between themselves and the enemy.

Apart from the characters, the world also contains several sounds to inform the player about events. Below is a brief list of the events and the sound:

- Once the player completes a puzzle successfully, a positive "ding" noise is played.
- Whenever the player hits a breakable box, such as a power-up box, a muted "donk" sound plays.
- When the player picks up a power-up, they each play their own unique sound relevant to their effect.

All game sounds were generated using a free tool called SFXR (drpetter.se/project_sfxr). The tool is able to generate a wide range of 8-bit sounds that fit any role in a game, from collecting a power-up, to jumping and firing.

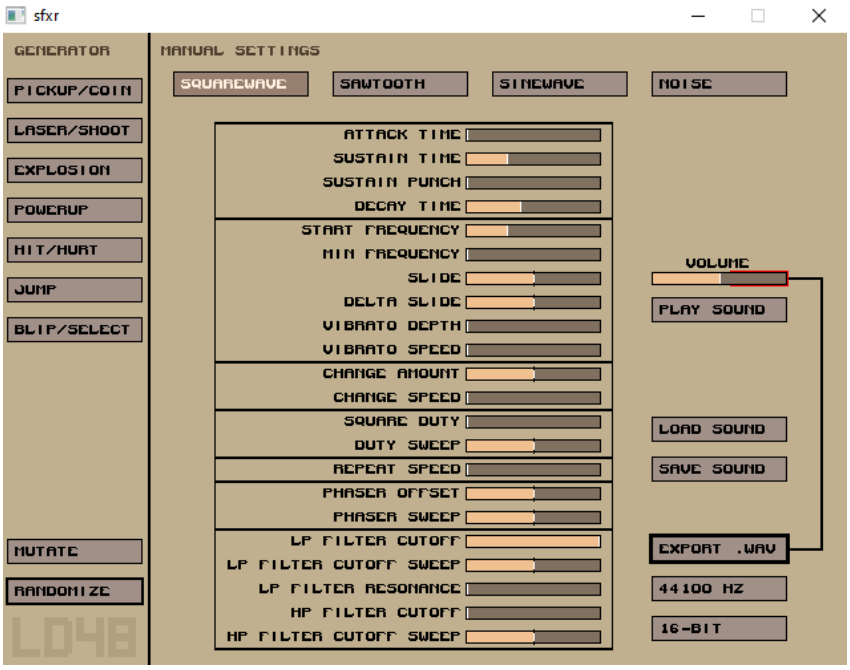

Figure 14: Screenshot of the SFXR program used to generate game sounds

Level Design/Creation

The level design initially created in the design document was to style the level to best create combat zones specifically tailored to each enemy. For example, for the shooting enemy, we would design an area with many cover areas, as well as having mostly flat or small hilled terrain, with a handful of platforms the player can jump on. This enforces the player to engage the shooter combat and utilize the cover to defeat them.

As the enemies were a core aspect of the level design, the main level of the game was not designed until at least one enemy was implemented. This was so we had a frame of reference in terms of scale, speed, and enemy lifetime of how the enemy characters would perform. The final level is split up into 5 main section and reveals a new enemy once the player enters a new section.

- Shooter Enemy: Contains cover, long straight terrain, some platforms to jump above enemies.
- Punching Enemy: Many platforms with multiple layers of travel. Close combat area with tight areas.
- Chasing Enemy: Long corridors, multiple layers of terrain. Contains holes that player/enemy can jump over. Contains small puzzle to unlock coin bonus

The 4 sections reveal an enemy, whereas the last section reveals the final boss. The player is blocked from leaving the boss area once entered and blocked from leaving until the boss has been defeated or when the player runs out of lives.

Figure 15: Overview of all sections of the final level

Save Game & High Scores

The save game is a simple file that contains a history of the player's previous games. The game data that is tracked includes the player's time, score, and a ISO8601 string of the date and time that the game was completed. ISO8601 is a standard format across date/time formatting which allows the string to be easily used regardless of the parsing language. The score and time formats are stored as floats, with the time being stored in the number of seconds the player took to complete the game. The save game also includes the top five high scores from the player's game history. Once a new game history gets added, a validation check is performed on the new game to check if it should be placed within the top 5 high score games. If the new game has a higher score, regardless of time to complete, then the game is considered a better high score. As the save game is limited and only contains these two lists, the save game file size will be kept small and only contain the required information for us to reuse in the future, where necessary.

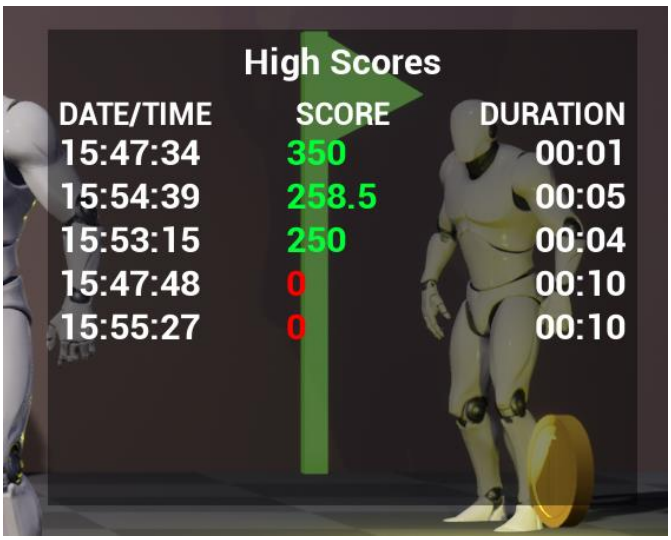

Figure 16: The High Score leader board on the main menu

Further Learning

Learning: Nav Mesh Links

The enemies in Quinn's Escape use the navigation mesh for path finding. One issue that arose using the navigation mesh in a 2D game is the ability to navigate over holes or wells in the world. However, after looking online and through documentation, Unreal Engine contains a method of being able to connect sections of the navigation mesh together using a navigation mesh link. The link allows for setting a 'bridge' during the editor that connects the generated meshes together. However, by default the link does not launch or jump a character if the link destination is higher. For this, I created a class called `VelocityProxyLink` that inherits from `NavMeshLink` and implemented the ability for a character to be launched using the `ACharacter::LaunchCharacter()` method.

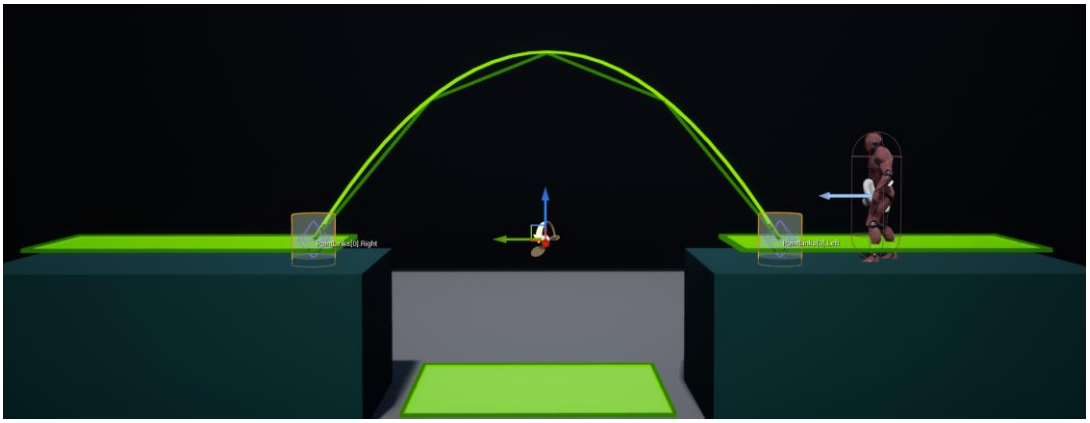

Figure 17: `VelocityProxyLink` in the editor, linked between two navigation mesh sections

Contribution Reflection

I feel my contributions to the project were quite substantial in progressing the game, as well as planning out implementations between the two of us on the project. We used a share Kanban board with multiple columns to easily split up out tasks and see where the other person is at. We each had our own "To Do" column which contains all elements that were next to implement, as well as a "In Progress" column and a "Done/Complete" column. Since I was in control of implementing the player and how they control, I completed this task quickly since it is one of the core elements needed before creating and adjusting other elements.

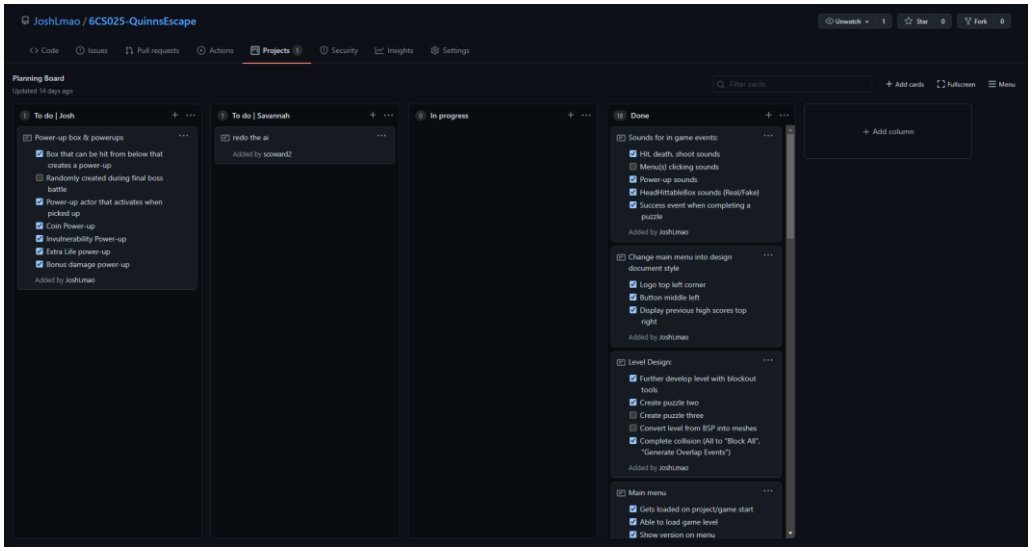

Figure 18: Project tasks split between the two of us in the project

During development, we had two test maps, one for each person. The maps were used for developing our own elements separately. Once we had implemented more elements and came close to finalizing them, I then created an initial level with everything discussed in the design document.

Our contributions to the project remained separate during the start of the development which allowed us to create the elements in isolation of each other. However, when coming to merge the elements together, we then also had to fix issues that had appeared. In hindsight, when creating and testing our elements, we should have also included our previously implemented implements as this would help to detect the issues sooner. However, none of the issues appear in the final game which is a good thing.

Module Reflection

Over the course of the module, I have learnt a lot of new elements of Unreal Engine that I would not otherwise had the chance to dive in and learn, such as behaviour trees, blackboards, as well as other key algorithms and techniques like projectile angling and action predictions. I do not feel like I had any major challenges with the module content, since I felt like I was able to easily follow the module's workshops and lectures.

I initially wanted to take these and implement them within Quinn's Escape and set out to do so. However, after implementing projectile angling to create a more realistic arcing shooting angle, it did not seem to fit in with the gameplay, so I removed it.

Quinn's Escape Post-mortem

The following is a post-mortem of Quinn's Escape which covers 3 areas of the game that I feel were good and 3 aspects of the game that were not as good and could be improved or reworked.

What Went Well

Design and Game Idea

As we set out to create a design document for the game before beginning any development, we were able to fully set out and specify a lot of elements of the game beforehand. Usually, developers will have an initial idea of the game they wish to create but run straight into developing the game and creating gameplay elements as they get around to implementing them. However, since we spent the time to clearly set out our core elements, it helped to not add any additional features that can creep in during development, as well as helped to set out a clear order of which elements require implementation. For example, the puzzles require the level being created before implementation, and the level requires enemies before creation, and so on.

Code Structure & Integration

The majority of the game was written in C++ classes as opposed to Unreal Engine's blueprints. This helped since managing code is easier than blueprints through source control such as Git. However, the other advantage is being able to easily view code and create a chain of inheritance through classes. When I initially set out implementing the player, I implemented it all in one character class. However, when creating an example NPC, I then created the shared base class between the two which was simply a case of copy and pasting the relevant code and creating the necessary call backs.

I feel the integration of these classes is also nicely integrated with the front-end user interface of the editor, inside the blueprints. The C++ classes' variables, which are variables to tweak or tune its behaviour, were publicly modifiable in blueprints to allow for easily tuning the characters behaviour, either at runtime or during development. If an enemy acted to fast or dealt too much damage, I could easily open its blueprint and change that variable. The helped speed up the development of the game.

Resulting Game

I also feel the resulting game is a strong point since being able to finish and polish any project that I have been a part of is a big achievement in itself. The final game has a simple menu which shows the player's previous high scores and buttons to play or quit. While the menu did not have a big scope such as containing rebind able controls or an options menu, it is very minimalistic and does not overwhelm the user, giving a good first impression. Players will be able to easily navigate through the application and play with ease as a whole since the interface is basic with minimal distractions.

What Went Not So Well

Enemies

I feel the enemies in the game could be much better and have a lot more room for improvement. Since Quinn's Escape is a single player game, the AI and enemies are the main focus

for the player, aside from navigating the level and solving puzzles. Since the game only contains a couple of the designed enemies, the game can feel empty or repetitive in nature due to the limit of different enemies. With more time to develop and work on the game, I feel that we should spend more time on the AI and enemies, such as using behaviour trees and blackboards inside Unreal Engine to assist in planning and executing their behaviour.

In the design document, we also designed the final boss to incorporate all elements of the NPC enemies in the level. The level was designed with a final boss in mind; however, the game lacks the final tough enemy which can feel like a bad pay off for the player.

Overall Scope

One other downside of Quinn's Escape is the overall scope for the game. On one hand, the scope had to be limited due to the time limit and workload, however with more time I think we could create a bigger and more alive world within the game or create several more levels for the player. Since we designed the game in the style of Mario, a 2D side scrolling game, we knew the game could be kept relatively small while including minute details and refinement for the gameplay within that area. However, playing the final game, the player could potentially finish the game in around 3 minutes, if they ignored puzzles and jumped over/avoided any enemies where possible. While the purpose of the game is to get the highest score, players are discouraged from doing this, while it still is possible.

With more time, I think we could have added a couple more levels to flesh out the game. For example, we could have separated each level to introducing a new type of enemy, starting with the most basic one. This method would give the player more chances and combat opportunities with each enemy, allowing them to become familiar faster in preparation for the boss enemy.

Models and Environment

Finally, the game only contains simple models and primitive shapes to block out the world. For each enemy, we utilized the same model as the player but with a tweak in its colour. While the player can differentiate between the enemy and Quinn, players who could be colour-blind would not be able to tell the difference which could cause issues. With more time, we would find a different enemy model that has animations to replace the current enemies.

The environment is mostly created using the Unreal Engine's block out tools which is ideal for a game in development and for initially creating levels. However, since we finalized out level's design, with a dedicated 3D modeller we would want to swap out these models with custom-made models in the style of a destroyed clothing shop to match the game's theme.