2/19/2018

Project Lance

Design Document

**Team Members:** Miles Dreisbach, Corey Dues, Matt Lempa, Tom Abraham, Kyle McHenry, Connor Loftus

**Version:** 1.0

**Table of Contents**

**1. Introduction 3**

1.1 Document Purpose 3

1.2 Project Overview 3

1.3 Definitions & Acronyms 4

**2. Problem Solving Approaches Considered 5**

2.1 Technology Choices 5

2.3.1 Unreal Engine 4 vs. Unity 5

2.3.2 C++ vs. Blueprint Scripting 6

**3. Game Screens 7**

3.1 Main Menu 7

3.2 Options Menu 8

3.3 Game Play 9

3.3.1 Practice Mode 9

3.3.2 Local Multiplayer 10

3.3 Character Selection 11

3.4 End Game Screen 12

3.5 Game Screen Flow 12

**4. Core Game Components 13**

4.1 Level Components 13

4.2 Level Loading Algorithm 14

4.3 Death Wall Trigger 17

4.4 Characters 18

4.4.1 Character Superclass 18

4.4.2 Character Specific Functionality 18

4.5 Single Player Game Play 19

4.6 Local Multi Player Game Play 20

**5. Class Structure 21**

5.1 Proposed UML Diagram 21

**6. Mid Assessment Details 22**

6.1 Goals 22

6.2 Member Tasks 23

**7. Prototype Details 25**

7.1 Concerns & Goals 25

**1. Introduction**

*1.1 Document Purpose*

This Design Document describes the overall functionality of the project being developed, the individual components that it is comprised of, and how they fit together. In addition, the document describes the decision process in picking the technologies our team is utilizing. The document will include terminology related to using Unreal Engine 4. This is to ensure accuracy in the components being described. Included in section 1.3 of this document is a definitions section. This document is intended for Rowan staff and students of the senior project class to see a high-level view of the components and structure of the project.

*1.2 Project Overview*

The project outlined in the design document is a third person action platformer being designed and developed in C++ with a game engine known as Unreal Engine 4. The title of the game is Project Lance. The goal of the project is to create a fully playable game mode where the main feature is a continuous level where the level segments are dynamically loaded through code. Each time the game is played the level will be unique as modular level elements will appear in a random order. The game is being designed with modularity in mind for ease of expansion to level components and characters.

The game being described within this document contains two game modes. Both are endless runners where the goal of the player is to keep moving forward while a cloud of smoke continues to pick up speed and approach the player. If a player fails to outrun the cloud that character will die, and it will be game over for them. Each character will have special action abilities to assist in navigating through the level. Single player mode (practice mode) will allow the player to see how far they can make it through the increasingly difficult level. Local multi-player mode will allow two to four players to compete to be the last one standing.

*1.3 Definitions & Acronyms*

Blueprint – Visual scripting system in Unreal Engine 4 that allows the creation of game elements and definition of object oriented classes within the game.

Game Engine – A software framework that aids in the development of video games.

GUI – Graphical user interface

HUD – A head-up display is like a GUI except it is overlaid over a game screen.

Local Multi-Player – Allows multi-player game play on one local computer.

Traces – A method for creating a line segment between two points to and collision between the two points.

Trigger – An element in unreal that allows for detection of collision between objects. Like an ActionListener in a programming language such as Java.

Unreal Engine 4 – The game engine being utilized for this project.

**2. Problem Solving Approaches Considered**

***2.1 Technology Choices***

***2.1.1 Unreal Engine 4 vs. Unity***

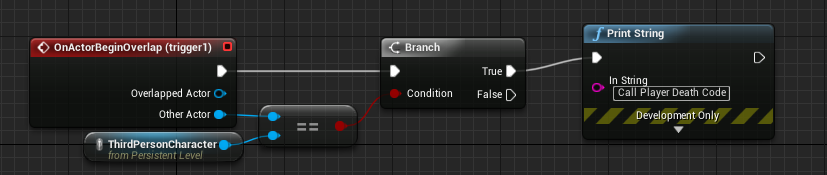
Before the start of our project our team had to decide on the game engine to use for development. Our team came to two choices based on research and team experience. The two engines considered were Unity and Unreal Engine 4. Overall our team had more firsthand experience in Unity. However, the research led to our team find significant benefits of Unreal Engine 4 that led our team to picking it. Listed below are the benefits of Unreal Engine 4 over Unity that our team found:

* Higher quality of graphics
* Uses C++ code for development of games
  + This is the industry standard for game development
  + Unity uses C#
* Unreal Engine offers their full source code
  + Allows for customization and extension of Unreal
* Blueprint Visual Scripting (see section 2.1.2)
  + Allows for quick scripting capabilities
  + Large amount of built in functions available within Unreal
  + Allows for extension directly with C++
* Found examples of well-developed games using Unreal
  + For example: Dead Island 2, Borderlands, Gears of War, Bioshock
* Ports to different systems easily (PC, Xbox, PS4)
* Does not depend on asset store for serious functionality.
* Large number of resources and learning material found online

***2.1.2 C++ Code vs. Unreal Blueprint Scripting***

One core component of Unreal Engine 4 that pushes it ahead of other game engines is the Blueprint Visual Scripting. This is a visual scripting tool that allows for easy creation of game logic for simple components. Additionally, Unreal Engine allows for the functionality of games to be coded in C++. Our team looked at both technologies and decided that using a combination would be most beneficial and efficient. This is because our team will not need to spend time coding simple components from scratch that our already available within the engine. However, our game requires more advanced functionality so C++ is needed for our project. Unreal Engine is designed to allow Blueprint and C++ to function side by side and even combine seamlessly. C++ code can be called inside of a Blueprint and C++ code can reference a Blueprint to run the functionality defined within it. Due to this our team is planning to utilize both C++ code and Blueprint scripting within our project. Blueprint works similar to designing circuits digitally and can be compared to using a program such as Active-HDL (taught in Principals of Digital Computers Lab at Rowan University). Below in figure 2.1.2.a is an example Blueprint.

Figure 2.1.2.a: Blueprint showing a print statement executing when a character overlaps an object named trigger1.



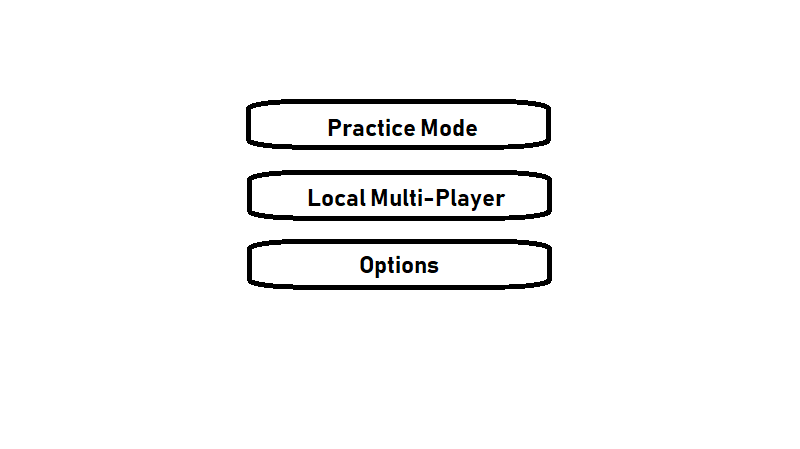
**3. Game Screens**

***3.1 Main Menu***

The main menu will consist of the following screens:

* Practice Mode (see section 3.3.1)
* Local Multiplayer (see section 3.3.2)
* Options (see section 3.2)

Figure 3.1.a: Layout of the Main Menu screen.

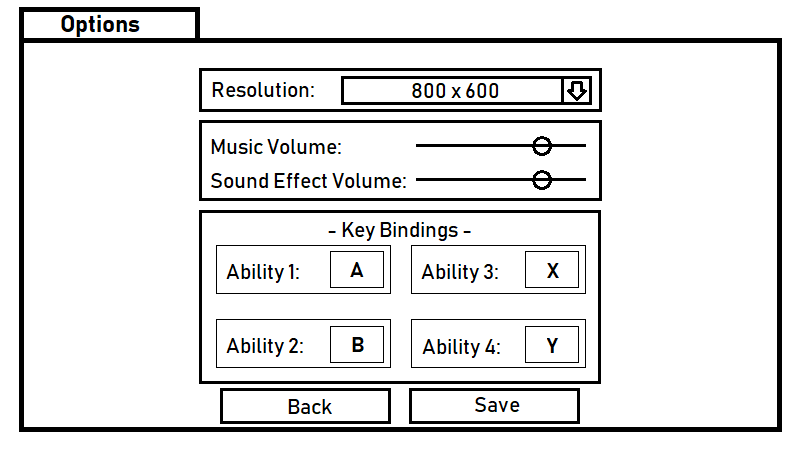


***3.2 Options Menu***

The options menu will consist of several settings to allow the player to customize game play. The settings that will be included are:

* Screen Resolution – Will be implemented using a dropdown menu.
* Music & Sound Effect Volume – Will be implemented using sliders that correspond with an integer variable.
* Key Bindings – Will be implemented using buttons. Once the button is clicked the next key the user presses will assign the variable responsible for that key bind to the key pressed.

Figure 3.2.a: Layout of the options menu.



***3.3 Game Play***

***3.3.1 Practice Mode***

Practice mode will allow the player to compete by themselves to see how far they can make it. The game view will take up the entire screen with a head-up display (HUD) in the top left corner. This HUD is illustrated in figure 3.3.1.a. The HUD contains the following components:

* Player icon – Icon unique to each character.
* Distance from cloud – Displays the distance from the player to the incoming cloud of smoke. This value will update with every frame.
* Ability cooldowns – Displays each of the characters four abilities. When an ability is on cooldown the icon will be faded out.

Figure 3.3.1.a: Layout of the Practice Mode game screen.

****

Player Icon

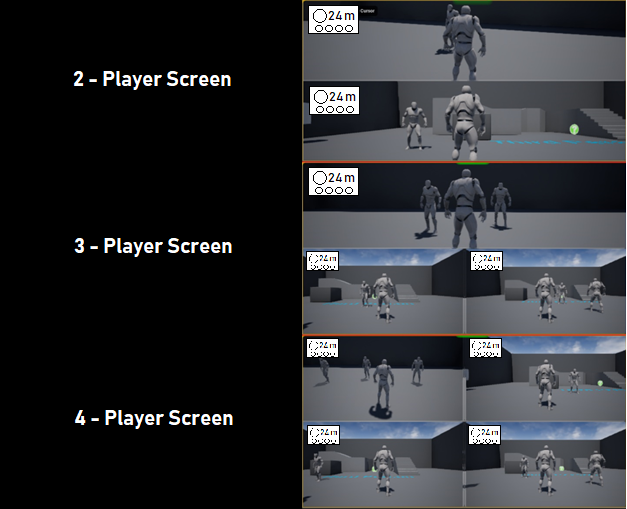
Ability Cooldowns

Distance from cloud

***3.3.2 Local Multiplayer***

Local multiplayer will allow for multiple people to play at once utilizing a split screen. Each player will have a screen identical to Practice Mode with the three HUD components defined in section 3.3.1. Figure 3.3.2.a displays the layouts for each of the possible number of players.

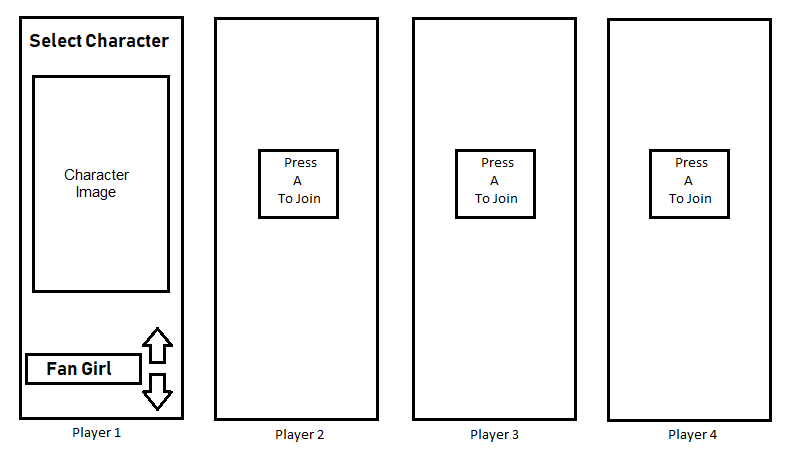
Figure 3.3.2.a: Layout of each of the Local Multiplayer screens depending on number of players.

****

***3.3 Character Selection***

The character selection screen involves four panes for each player. This screen will be the same for both single player and multiplayer modes. However, with the difference that on single player mode only the first player slot will be able to select a character. On multiplayer mode the screen will default to the first player selecting a character (illustrated in figure 3.3.a). The player will be able to navigate through the possible characters by selecting up or down on their controller. The additional controller slots will be allowed to select the ‘A’ button to enter the character selection pane. Once each player selects a character the game will load.

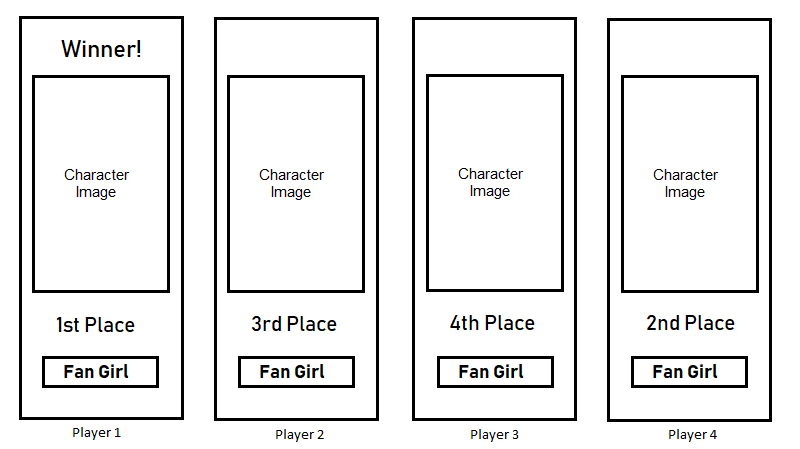
Figure 3.3.a: Layout of the character selection screen.



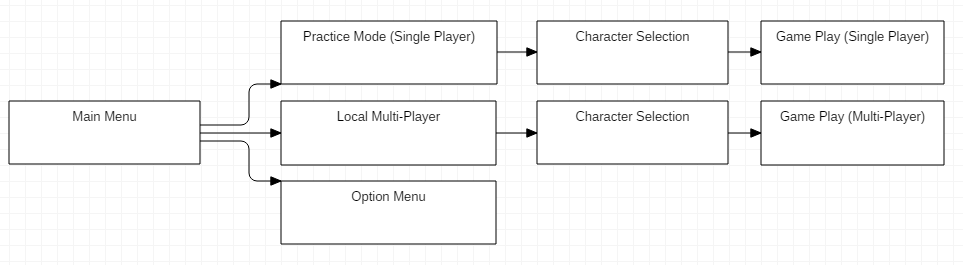
***3.4 End of Game***

The end of game screen will look like the character selection screen. The end game screen and character selection screen will share many of the same components to reduce the number of unique components needed to be coded. The differences are that the screen will show the place each player ended the game in.

Figure 3.4.a: Layout of the end of game screen.



***3.5 Game Screen Flow***

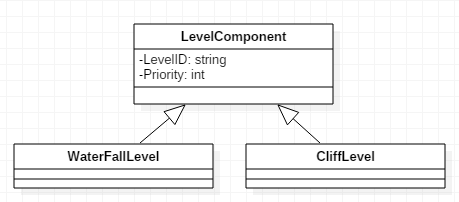


**4. Core Game Components**

***4.1 Level Components***

The levels will be made up in components that can be loaded in any order. These components are designed to be modular and fit together. The start and end of each level component will be the same width to ensure seamless transition from one until the other. However, the core design of each component will differ. Additionally, each level component will contain a unique ID to be used for loading the level. These reused variables and any additional reused functionality will be contained in a LevelComponent superclass.

Figure 4.1.a: UML diagram for LevelComponent sample class and example sub classes.



***4.2 Level Loading***

The level components will be loaded in a random order to allow for a dynamic new experience every play through. Due to the nature of the game being an infinite runner (if players never die) the components cannot all be loaded at the start of the game as the computer would not have enough memory. Even loading many components would be inefficient. Due to this the level components will be loaded as the players progress from component to component. To mitigate the risk of players reaching the end of a component before another is loaded the game will load two components ahead of the last component a player is at. Additionally, level component object will be required to be deleted after they are not being used to prevent the game from slowing down due to many objects being loaded. Level components will be deleted after the last player is two components ahead of it. Therefore, there will be two components loaded before the player in first play and two components behind the player in last place.

Figure 4.2.a: Diagram showing the number of components that would be loaded given players are spread out between three components.

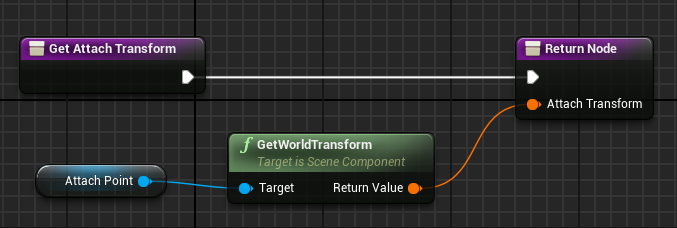
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | X  Player 4th | XX  Player 3rd  Player 2nd | X  Player 1st |  |  |
| Component next to be deleted |  |  |  |  |  | Component last to be loaded |

The level components will load randomly however a priority algorithm will be implemented to reduce the chance of duplicate component being loaded in a row. Once a component is loaded it will be given a loaded priority for the future.

The design of loading level components dynamically works in four segments. They are illustrated below in Unreal Blueprint.

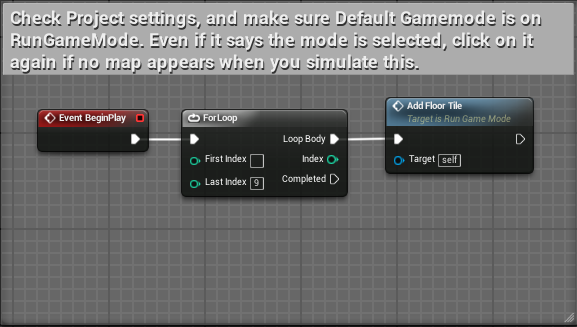
1. GetAttachTransform

Figure 4.2.a: This screenshot shows the blueprint that will, when called, attach a tile to the attach point (or origin, if it's the first tile). It is then attached to the attach point created on the map tile. This blueprint function is duplicated across all map tiles.



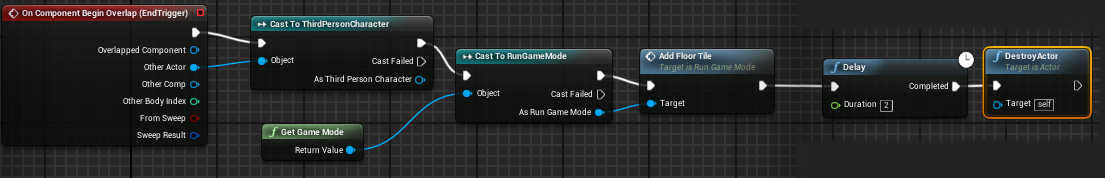
1. RunGameMode

Figure 4.2.b: When the game begins, a for loop is started. However, there is no end criteria, so this works more as a while loop. Every time the loop recurs, the AddFloorTile blueprint is called, which will add another map tile from the pool. This will continue until the simulation is terminated. The Index fields indicate that only 10 total tiles (index 9 is item 10, like a normal array) can be on the field at any time, so it is necessary to remove tiles you pass. (The comment mentions that when trying to run the game in this test mode while making it, if you do not go into the project settings and set it to RunGameMode from its default, then the game will not run properly. This will be fixed later.)



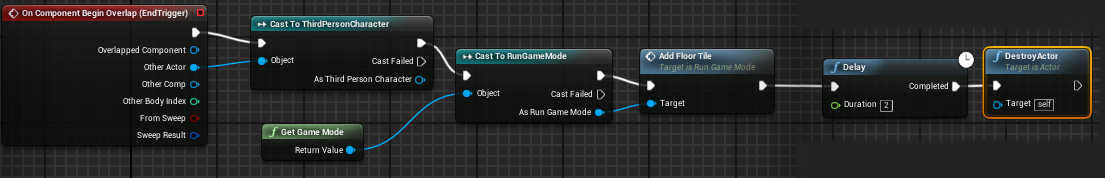
1. FloorTileBP

Figure 4.2.c: There is a trigger set on an area at the end of each tile, and when it detects a player passing through it (it is set to only detect a player), it will call to add a new floor tile at the end of the set, and after a delay set of 2 seconds, the tile attached to that trigger is destroyed, freeing up the last index in the loop to place a new tile from the pool.



1. AddFloorTile

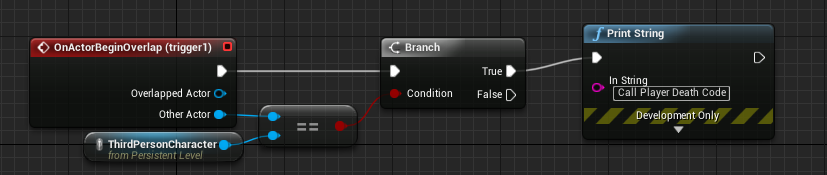
Figure 4.2.d: An array is created of all available tiles. When AddFloorTile is called, it takes a random number from index 0 to the last index of the array of Floor Tiles and moves that tile to be spawned at the next spawn point, which is an attach point. The spawned actor is cast to an instance of FloorTileBP, and then calls GetAttachTransform, and then that tile's attach point is set as the next spawn point for the next tile when this function is called again.



***4.3 Death Wall Trigger***

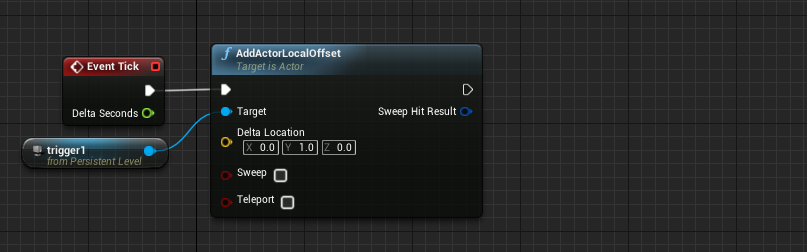
The component in our game known as the death wall is an invisible wall that moves along the generated levels giving the players a reason to keep advancing forward. The wall will be created using a trigger box. A trigger box is an element in Unity that is used to detect collision between another element. In our example the collision being detected is with another character. Code will then be executed once collision is detected between the trigger box and the player.

Figure 4.3.a: Displaying the flow of the trigger detecting collision with a player. Print String node to be replaced with call to Player Death code.



A second element of the trigger is that it will be constantly moving throughout the game. This will be accomplished using the “AddActorLocalOffset” function that is provided by Unreal Blueprint. This function allows the position to be updated by a given amount based on the last position of the object. This function will be called on the Event Tick of the game which is called every frame. Added to the “AddActorLocalOffset” function will be additional C++ code to increase the amount offset over time. This will increase the difficulty of the game as players live longer as the trigger wall will gradually increase in speed.

Figure 4.3.b: Displaying flow of the box trigger being offset each frame of the game.



IncreaseSpeed

***4.4 Characters***

***4.4.1 Character Superclass***

The Character superclass will contain functionality that will be a part of each individual character. The following components will be a part of the Character superclass:

1. Ledge Grab and Climb

Every character will have a ledge grab and side movement while hanging onto the ledge. This works with what Unreal Engine calls "traces". Traces work by creating an arrow that sticks out in front of your player, and the arrows are used to get feedback from the level on what/if there is anything present along the line segment created by the arrow. The way the ledge grab/shuffling along/climbing/climbing around corners works is we have traces along the sides, and in front of the character that look to get feedback from a certain subset of walls that respond to the traces. If the traces detect that there is a wall that responds to the traces, and the player is close enough an animation plays, and the player grabs on to the ledge. It works the same for shuffling/turning corners etc.

2. Aiming System

Every character will have a similar design for aiming for their skill shots- where if they hold down the button for a skill shot the player will be able to aim their shot and upon release of the button they will make the shot. Like smart casting in league of legends but with press and release of a button.

The way this works is just manipulation of the position of the camera, and the drawing of a Widget (this one specifically is a UI element). The player hits the aim button, the camera's position is moved closer to give that zooming in effect, and a crosshair is set to appear on the screen. When the player releases the same button, the opposite happens: the camera is moved back out, and the crosshair is set to be invisible again.

***4.4.2 Character Specific Functionality***

Character specific functionality to be added to this document in the future.

***4.5 Single Player Game Play***

The single player game will allow a player to practice by themselves. The game will play the same as multiplayer except the single player continues until they are in contact with the death wall trigger and get game over. Several steps need to happen when a game is loaded.

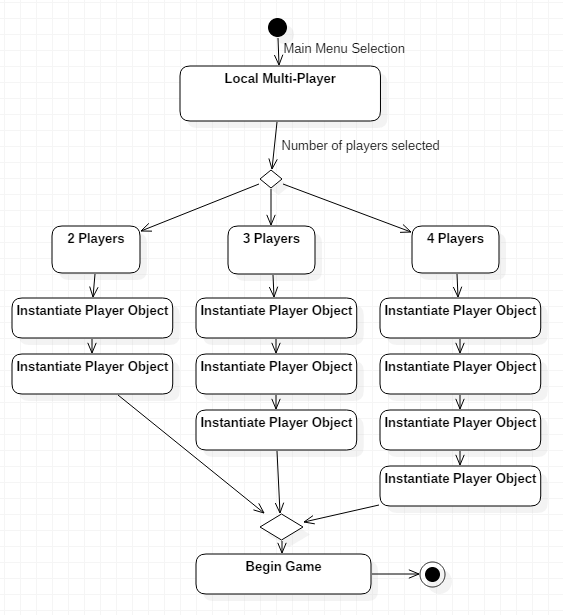
1. Load initial level segments for the player to start on.
2. Load the death wall trigger.
3. Load the HUD elements for the player and reset all the values to default.
4. Load the player’s character.
5. Connect each of the components of the game to the HUD to allow for dynamic updating.
6. Begin countdown and start the algorithms for loading new level segments and increasing difficulty over time.

The above steps will be done during the instantiation of a player object. This is because it will allow for easy generation of multiplayer (see section 4.6).

***4.6 Local Multi Player Game Play***

Local multi-player will involve a split screen for between two to four players. This game mode will allow the players to compete in a racing type game mode. However, instead of racing to a finish line it will be the last player who survives that wins. Creating a multi-player game mode in Unreal involves instantiating a new player object for each player that will be in the game. Each player object will contain all the components included practice mode. That will allow each player to have a separate HUD and when a player dies it will not affect the other objects. After the number of players is selected in the character selection screen a parameter will be passed to the class that loads the game. The class will then run a function to instantiate the number of player objects as the parameter passed in.

Figure 4.6.a: Activity diagram showing the sequence of steps taken from the main menu to starting a game for local multiplayer.

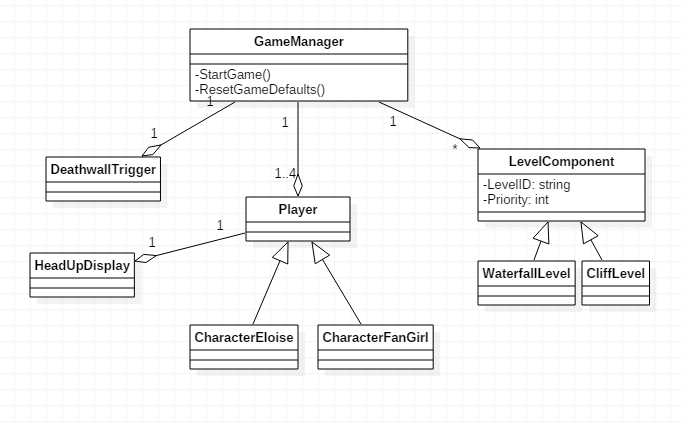


**5. Class Structure**

***5.1 Proposed UML Diagram***

Outline in the section is a diagram of the proposed classes and how they will relate to each other. The classes currently contain minimal attributes and operations. This section will be updated to include more detail in the future and is subject to change.

Figure 5.1.a: UML outline of the proposed classes and how they relate in the project.



**6. Mid Assessment Details**

***6.1 Goals***

Our goals for the mid assessment is to create a fully functioning version of our game with all the core functionality components defined in section 4 of this document. Our focus for up to the mid assessment will be functionality that is core to the game and applies to all characters. Our team will leave specific design of levels, characters, and user interface elements for after the mid assessment unless each of our goals are reached. Listed below are the specific goals our team plans to accomplish and deliver at the mid assessment:

* Dynamically generating levels that load continuously
* Priority algorithm implemented for loading of level segments
* At least three placeholder level segments to demonstrate functionality of priority algorithm
* Fully functioning menu system
  + Look will not be a priority, but flow will
* Single player and multi-player functionality both playable
* Implementation of death wall trigger and algorithm to increase the speed over time
* Implementation of character superclass functionality with a sample model that is playable
* Placeholders for HUD over gameplay screen
* Playable game with the above functionality

***6.2 Member Tasks***

Miles –

* Working on character specific functionality (subclasses)
* Level design
* Building of modular levels

Corey –

* Working on character specific functionality (subclasses)
* Key bindings and options functionality
* GUI/Menu functionality development

Tom –

* Creation of multiplayer game mode
* Creation of dynamically loading levels
* Level design
* Building of modular levels

Connor –

* Creation of priority algorithm for level loading
* Creation of GUI/HUD elements for gameplay screen
* Menu creation
* Development of death wall trigger and algorithm for increasing the speed

Matt –

* Working on character specific functionality (subclasses)
* Combining separate elements of the game into one unified level

Kyle –

* Development of aiming functionality for all characters
* Development of character superclass to cover overall functionality of each character
* Animation

**7. Prototype Details**

***7.1 Concerns & Goals***

For our prototype our team plans to focus on delivering the main functionality of the game with a focus on the core functionality. The look will be bare bones as adding assets will be easier once all the functionality is implemented. One main goal of our prototype is to address concerns our team has about delivering on some of the more complex aspects of our game or concepts that our team does not have previous experience with. Below is a list of concerns that our team has identified during the design phase of our project.

* Creating a dynamically loading level. Our team previously only has game development experience where static levels have been created.
* Having the components of our dynamic level load in a random order.
* Creating a multiplayer mode for our game.
* Allowing our level to increase in difficulty over time.

Our team plans to show two prototypes which each address separate concerns. To show and address concerns related to the main game functionality our team plans to use a single player prototype with the following functionality shown:

* A dynamically loading level consisting of three sample level segments. This will load continuously and allow the player to keep running until they die.
* The level components will be loaded at random to show the concept for a unique level being loaded each play through.
* Implementation of the moving death wall trigger that ends the game on contact with the player. Plan to show that the speed of the wall can increase in speed over time. This will address the concern of having our level increase in difficulty over time.
* Having a character with at least one special ability playable.
* A simple main menu to demonstrate our ability to add menus to the game.

In addition to this single player prototype our team wants to demonstrate our ability to create a multiplayer game as that is a goal for the final product. A simple multiplayer game will be shown as a prototype to demonstrate our team’s ability to have this functionality implemented in the final version.