INFERNAL BATTLE:
AN AI-BASED ARCADE FIGHTING GAME USING UNREAL ENGINE AND C++

A project report submitted in fulfilment of the requirements for the Mid-Semester Small Project in

ARTIFICIAL INTELLIGENCE AND EXPERT SYSTEMS

Submitted by

Anubhav Sharma and Arjun Choudhry
{2K18/IT/029, 2K18/IT/031}

Under the Guidance of

Ms. Reena Tripathi
Department of Information Technology
Delhi Technological University
New Delhi-110042

DELHI TECHNOLOGICAL UNIVERSITY (FORMERLY DELHI COLLEGE OF ENGINEERING)

ACKNOWLEDGEMENTS

The successful completion of this project wouldn't have been possible without the help and support of many, and we would like to extend our sincere gratitude to them.

We are highly indebted to Ms. Reena Tripathi for being a source of constant motivation, and for helping us improve the project to achieve its present shape. We would also like to thank her for giving us valuable insights on AI search algorithms like A* algorithm and min-max algorithm.

It would be unfair, should we miss out on thanking our fellow classmates, for some wonderfully insightful discussions on search algorithms, for working around us, and for making the subject even more interesting and for sharing many unique ideas, both on and off the subject.

Last but not the least we would like to thank our parents for always being there, and guiding us through this journey.

CONTENTS

ABSTRACT

AI algorithms used in games, or “game AI”, refers to an extremely broad set of computer algorithms ranging from robotics, computer graphics, control theory (both by the user and the computer) and computer science in general. However, contrary to popular belief, game AI is often not made up of "true AI", i.e., the techniques used do not necessarily help the computer “learn” or other standard criteria, but only constitutes "automated computation" or a predetermined and limited set of responses to a predetermined and limited set of inputs.

In recent years, with the rampant growth in the field of game AI, a large number of game developers developing games often tend to use the term “smart” or “AI”, when describing their game. Many in the field of AI have argued for and against the motion that “game AI” is not true AI, but an advertising buzzword used to describe computer programs that use simple sorting and matching algorithms to create the illusion of intelligent behavior. While this is up for debate, in this project, we focus on using these AI algorithms in our game “Infernal Battle”, that we have created using Unreal Engine, visual scripting and a C++ backend.

The first AI-based game, Nim, was made in the early 1950s, and was seen as a groundbreaking use of AI. But it wasn’t until the 1970s that single-player games were popularized. In 1997, IBM’s Deep Blue computer, specifically used for AI games like chess and checkers, was able to defeat Garry Kasparov, a Russian Chess grandmaster. This was seen as a major breakthrough in the field of game AI, and led to a massive amount of research being done in the field of game AI subsequently.

1 INTRODUCTION

In computer and console games, game AI is used to generate responsive, adaptive and intelligent behaviours primarily in non-player characters (NPCs), or the characters controlled by the computer, in a manner which would closely mimic human intelligence.

Games have been an area of research in the field of AI, right from the very beginning, and the first AI-based game, Nim, was made in 1951, and officially published in 1952. Since then, game AI has seen a massive surge, both in research and implementation. In the 1960s and 1970s, the single-player games with the computer as the enemy started appearing.

Over the years, with the increasing availability of game development platforms with defined classes and frameworks for easy access to the developers, games incorporating AI have seen a massive commercial surge. Unreal Engine by Epic Games, which was developed by Tim Sweeny, is one such platform based on the C++ programming language, which is extremely popular for the development of games.

In this project, we have created a single player, AI-based, Arcade fighting game called Infernal Battle. This is a human versus computer game in which our character, Bryan, is tasked with fighting a demon, Skin Hunter. The latter is controlled by the computer, hence is an NPC.

The game has been built using Unreal Engine 4.25.4, built using a C++ backend. The main workflow has been created using visual scripting, with the code further optimised and cross-checked by editing the code. Our project uses around 70 classes for function calls which are either pre-defined or custom designed.

2 METHODOLOGY

2.1 Prior Knowledge Required

1. **Workflow Designing using Visual Scripting**

2. **Knowledge of C++,Blueprinting and Classes**

3. **Unreal Engine:** We chose Unreal Engine over Unity due to its significant use of visual scripting and the use of a C++ backend. The major challenges in developing a game using Unreal Engine are developing the controls of the player and the computer-controlled character.

4. **AI Search Algorithms:** Prior knowledge of AI search algorithms was very important for proper control of the computer-controlled character. We chose a modified A* algorithm for the same.

2.2 State Space Search

2.2.1 Agent

Our agent is a problem-solving goal-based agent which searches for the protagonist character of the game (Bryan) using AI Search-based techniques, and tries to reduce Bryan's health to zero after finding him.

2.2.2 States

All the positions in the arena for battle in the game "Infernal battle" are states in our project.

2.2.3 Initial State

The initial state of the agent is situated near the centre of the arena.

2.2.4 Goal State

The goal state keeps changing according to the inputs given by the user to change the position of Bryan in the game.

2.2.5 Action

The agent can move left, right, in front, backward, and diagonally as well from its current position. The agent can punch to hit the target to reduce its health and achieve the objective of the game.

2.2.6 Transition Model

The change in position and the fighting pattern of the agent with respect to the player's position constitute the transition model in our project.

2.2.7 Goal Test

When the agent detects the player, that state of the agent is the goal state for that instant of time till the player moves again and the goal state changes again.

2.2.8 Path Cost

The cost of movement is delay, as a movement of the agent in the wrong direction would give the player time to attack the agent and ultimately win the game. Even an attack movement results in some delay as the agent cannot move while it is attacking.

2.3 AI Search Algorithms Used

2.3.1 A* Search Algorithm

It is a heuristic search technique or an informed search technique. The most important advantage of A* search algorithm which separates it from other traversal techniques is that "it has a brain". This makes A* very smart and pushes it ahead of other conventional algorithms.

In our game, we have used A* algorithm code with a slight modification. In Unreal Engine program, each node is a grid in the arena and as we are constantly changing the destination node, the agent needs to revise the destination node.

The A* algorithm selects the path that minimizes $f(n)=g(n)+h(n)$.

In our game, the agent continuously searches for the player to quickly kill him. For this quick path-finding, A* search algorithm served the purpose best. Other search algorithm would not have been as time efficient to be able to constantly change the destination in real time. However, using A* algorithm, by keeping our heuristics for the player node updated each time it moves, the agent is able to quickly find and attack the agent.

The costs were calculated along with the heuristics and pushed onto the priority queue.

The algorithm for A* implementation is as follows:

1. Take input from the user and update the player's position
2. Based on the updated position of the player, calculated the heuristic values of the nodes in the arena and the associated costs.
3. Take the nearest node with lowest heuristic value and push it in the closed list from the open list. Then, remove it from open list.
4. If the destination node is reached, break the loop.
5. Try to update node position for the current edge to make paths more precise unless heuristic is not admissible (overestimates), in which case, we have to use constant values for given node to avoid creating cycles.
6. If the node is already in the open list and the new result is worse, skip.
7. If the node is already visited and processed, and the new result is worse, skip.
8. Repeat the steps 3 to 7 till the open list becomes empty.

In our project, we have used a slightly modified A* algorithm. It differs in the state being a right-angled triangle instead of a square, thereby resulting in a more optimized searching.

2.3.2 Min-Max Algorithm (Originally intended)

Infernal battle is a human-vs-computer game and can make use of different AI algorithms. So, for making the AI agent faster, we intended to use min-max algorithm. The AI agent takes into account the fact that the player is playing optimally and plays intelligently using recursion by creating a game tree and traversing it backward.

The AI agent finds the moves that the opponent might choose to win the game and hence doesn't follow the player blindly following it everywhere. Instead, while going through the edges, the AI agent considers that this would result in a loop as the speeds of both are the same, it acts intelligently considering the fact and guessing the path that the player would follow.

The pseudocode for min-max algorithm used in the game is as follows:

```
1.      if depth ==0 or node is a terminal node then

                return static evaluation of node

2.      if MaximizingPlayer then        // for Maximizer Player

                maxEva= -infinity and

                for each child of node do

                        i) eva= minimax(child, depth-1, false)

                        ii) maxEva= max(maxEva,eva)        //gives Max of the values

                return maxEva

3.      else                        // for Minimizer player

                minEva= +infinity

                for each child of node do

                        i) eva= minimax(child, depth-1, true)

                        ii) minEva= min(minEva, eva)        //gives Min of the

                values

                return minEva
```

This increased the difficulty level of the game, making it eventually unplayable and hence was subsequently detached.

2.4 Implementation Details

Programming Languages Used	C++ (direct and via Blueprinting)
Operating System	Created in Windows, but packageable for Windows, Linux, MacOS, Android
Library, Packages or APIs Used	Unreal Engine Modules and Libraries, Standard C++ classes
Interface Design (GUI)	Classes with parent class Unreal Engine Viewport Widget

Table 1: Details of the implementation environment of the program, to ensure stability and compatibility across systems and operating systems.

3 GAME DESIGNING

3.1 Character Setup

We have created some assets that will be used by our characters, Bryan (player character) and Skin Hunter (enemy character).

Firstly, we have created Character Blueprint Classes, which inherit from Character class. These handle most of the affairs of the characters, including movement logic, attack logic, damage logic and death logic.

Secondly, we have created Animation Blueprint Classes (we will populate these later in this process), which will be used to drive the animations of the character.

The next step was developing a skeletal mesh, for which we used the SK_Mannequin_Skeleton Skeletal Mesh template and few opensource third party assets.

Character Setup

Character movement: To create the character movement, several parts are needed, each working together to make the character move around in the game.

We needed a character, inputs for that character, how the character responds to inputs from a player, animations for the character moving, transitions to and from animations, and the logic behind the transitioning between animations.

Using visual scripting and function code, we built a character that can walk, run, crouch and crouch-walk, go prone, jump from a stationary standing position, or jump while moving.

3.2 Game Setting

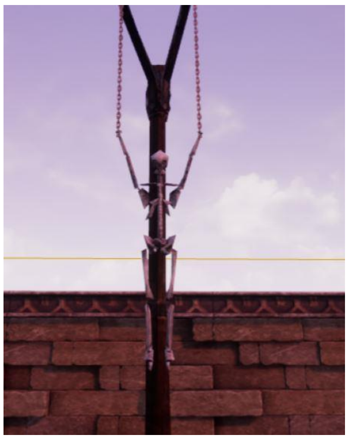
A Game Engine environment is a specialized development environment for creating video games. Users build the game with the game IDE, which may incorporate a game engine or call it externally. Game IDEs are typically specialized and tailored to work with one specific game engine.

3.2.1 Story

The storyline behind the game is that we have a player controlled by the user (Bryan), with an enemy controlled by the computer (Skin Hunter). Bryan and Skin Hunter are engaged in combat and the first one to lose all of their health due to attacks from the other player dies. Thus, the player must cautiously choose to attack and defend oneself, so as to win the game.

3.2.2 Blockout

We have used 3ds Max 2014 and basic 3D primitives such as boxes and cylinders for placeholder models. Because primitives are easy to make, it's a good idea to do different versions of the scene to see what works the best. We have also imported different objects into the scene – such as rocks, trees and characters – to get a better idea of the scale.



Blockouts

Unreal Engine 4 (UE4) supports a variety of lighting features. It is possible to mix and match most of UE4's lighting features.

In "Full HDR Lighting with per-pixel lighting from the Sun" tier, all of the HDR lighting features are taken into account. This tier has some post processing features and has the same advantages and recommendations as other tiers. One exception here is that the developer can add a single Directional Light to the scene that automatically uses per-pixel lighting for higher quality. This feature gives access to Static Lighting and Global Illumination features.

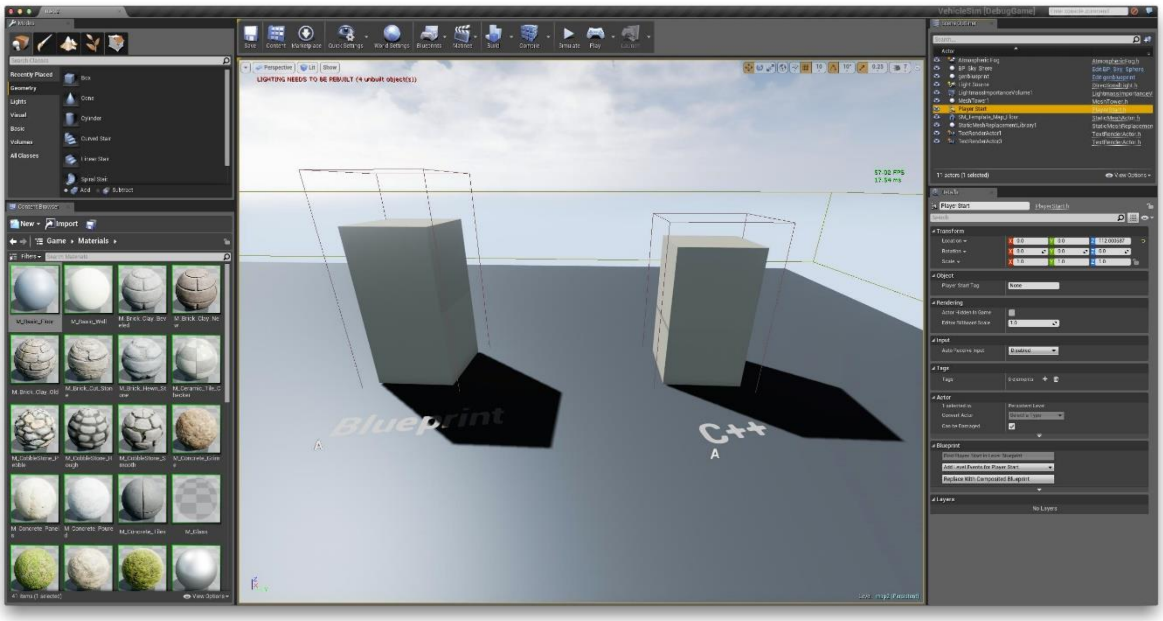
The game settings in Unreal Engine include weather adjustment, fog, clouds, and sky colour for better depiction of notion of the character, and to make it somewhat challenging for player to navigate through the map.

3.3 Game Environment

Game Environment

3.3.1 Static Meshes

Static Mesh is a piece of geometry that consists of a set of polygons that can be cached into video memory and is rendered by the graphics card. This allows the static meshes to be rendered efficiently, meaning they can be much more complex than other types of geometry such as Brushes. Since they are cached in video memory, static meshes can be translated, rotated, and scaled, but they cannot have their vertices animated in any way.

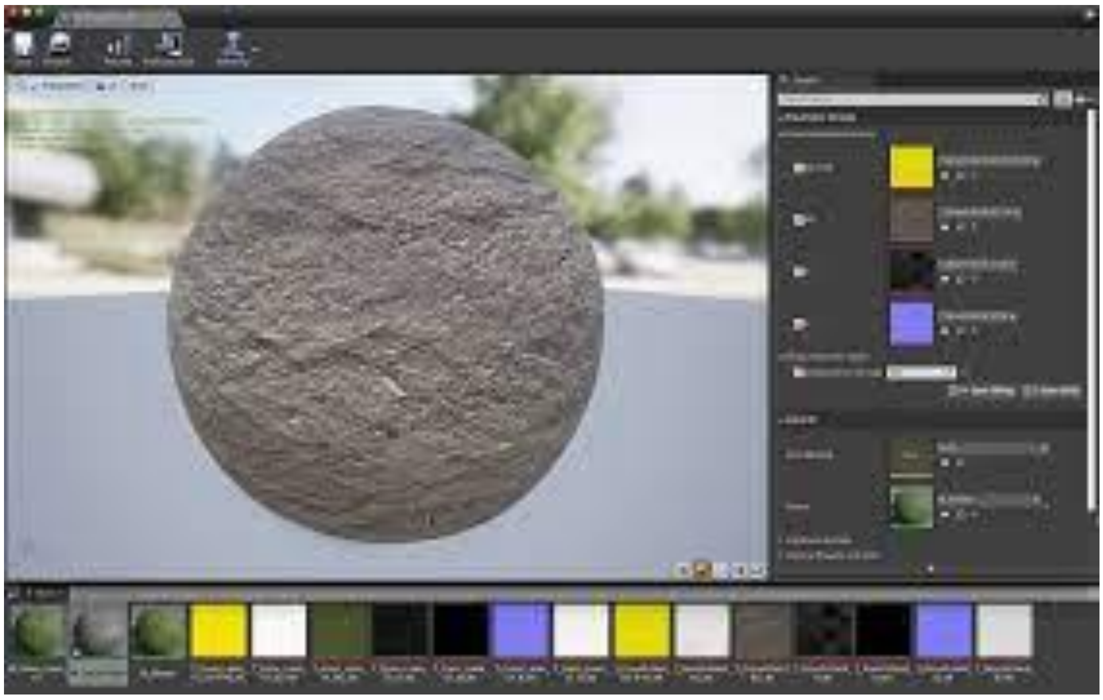


Static Meshes

In our game play, we have used meshes rocks, walls and the statue along the wall. They appear to be illuminated because we have fitted small light cylinders there.

3.3.2 Textures

Initially, the meshes are white in colour. However, to make the game play more attractive and engrossing, we have given the meshes some textures including colour, pattern and roughness, so that it gives a more realistic look.

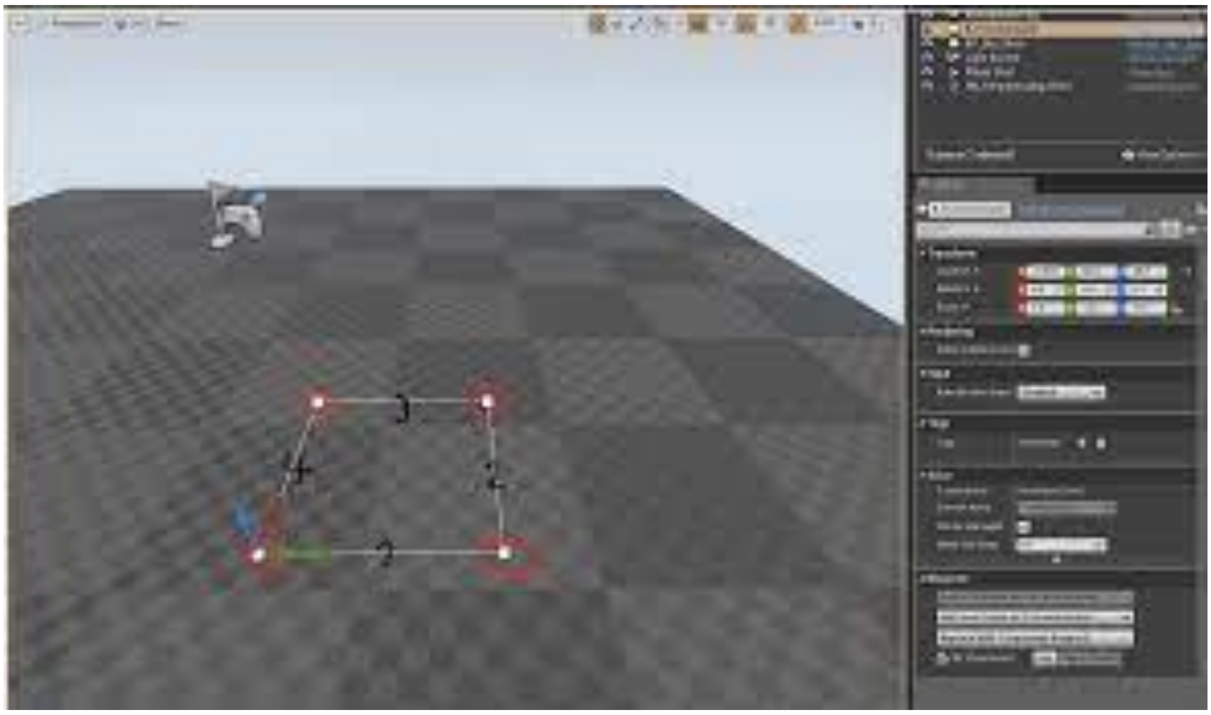


Textures

Textures are images that are used in materials. They are mapped to the surfaces the material is applied to. Textures can be applied directly - for example, for Base Colour textures - or the values of the texture's pixels (or texels) are used within the material as masks or for other calculations. In some instances, textures may also be used directly, outside of materials, such as for drawing to the HUD. For the most part, textures are created externally within an image-editing application, such as Photoshop, and then imported into Unreal Editor through the Content Browser.

3.3.3 Floor component

A running ground with defined parameters is needed to constrain the movement of the player. It is also advisable to redesign the ground to give it a better look.

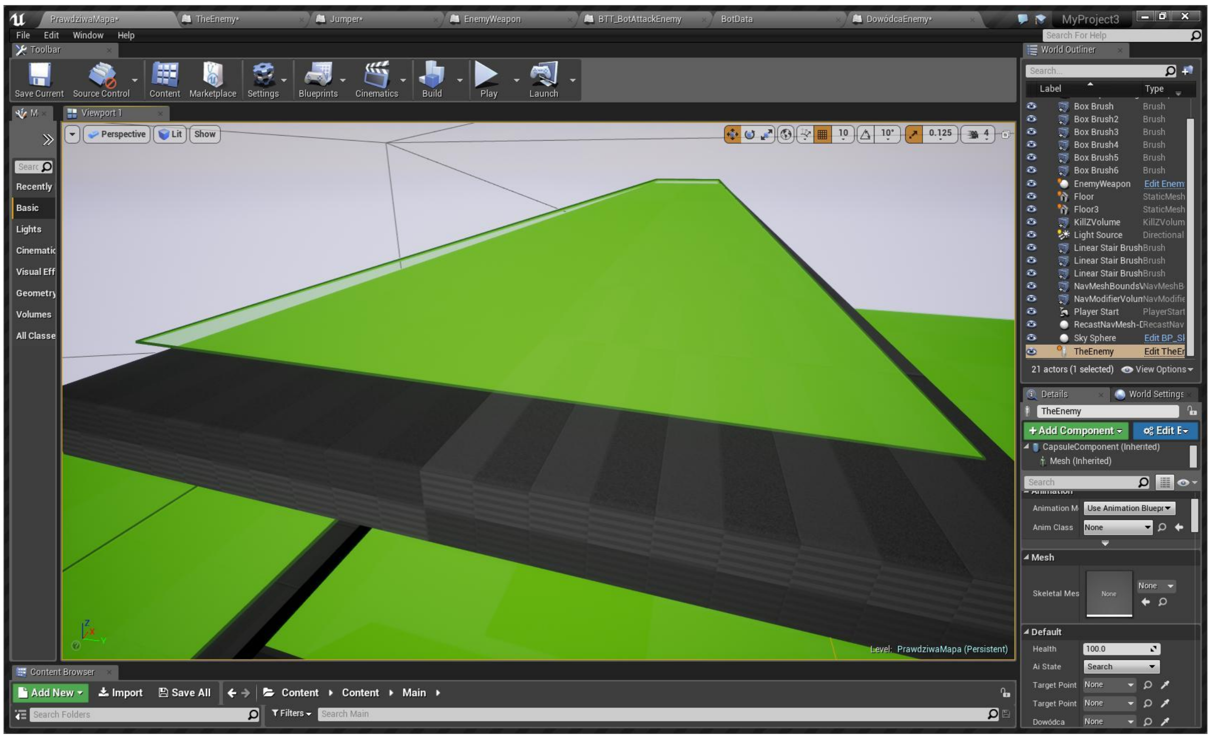


Floor Components

3.3.4 NavMesh Bounds Volumes

This volume only blocks the path builder - it has no gameplay collision. A NavMesh is a grid of triangles which are placed (automatically, in UE4) in the 3D environment. The area within the triangle is walkable and we have 3 edges that we can move to from each part.

By default, pretty much everything is taken over by Unreal Engine. We need to make sure there is a nav mesh around (this is done with a nav mesh volume and can verify it by pressing "P" while in viewport which toggles a visualization of it).



NavMesh Bounds Volume

“dd”, is the NavMesh module as a public dependency in MyProject.build.cs - this links in any of the Recast/Detour libraries in the project, the sub-class ARecastNavMesh in C++ and declares the constructor.

Within DefaultEngine.ini, we add a section to tell the engine to use our Navmesh Implementation. We create a second class below that, which sub-classes FRecastQueryFilter (we can also use INavigationQueryFilterInterface, but if extending from RecastNavmesh, there are a lot of assumptions needed in the existing code that can lead to a crash).

In this second class, we override virtual functions to allow changes in the cost of a path. Next, we declare a variable of our second class-type, and define the AMyRecastNavMesh constructor. Later, we set the filter implementation for the DefaultQueryFilter shared pointer in our constructor that can be seen in the following few classes:

MyProject.build.cs

```
using UnrealBuildTool;

public class MyProject : ModuleRules
{
    public MyProject(TargetInfo Target)
    {
        PublicDependencyModuleNames.AddRange(
            new string[] {
                "Core",
                "CoreUObject",
                "Engine",
                "InputCore",
                "Navmesh"
            }
        );

        PrivateDependencyModuleNames.AddRange(new string[] {  });
    }
}
```

DefaultEngine.ini

```
[/Script/Engine.NavigationSystem]
+SupportedAgents=(Name="CustomNavmesh",NavigationDataClassName=/Script/MyProject.MyRecastNavMesh)
```

MyRecastNavMesh.h

```
#pragma once

#include "AI/Navigation/PImplRecastNavMesh.h"
#include "AI/Navigation/RecastNavMesh.h"
#include "MyRecastNavMesh.generated.h"

class MYPROJECT_API FNavFilter_Example : public FRecastQueryFilter
{
    // Override any functions from INavigationQueryFilterInterface/FRecastQueryFilter here
};

UCLASS()
class MYPROJECT_API AMyRecastNavMesh : public ARecastNavMesh
{
    GENERATED_BODY()

public:
    AMyRecastNavMesh(const FObjectInitializer& ObjectInitializer);

    FNavFilter_Example DefaultNavFilter;
};
```

MyRecastNavMesh.cpp

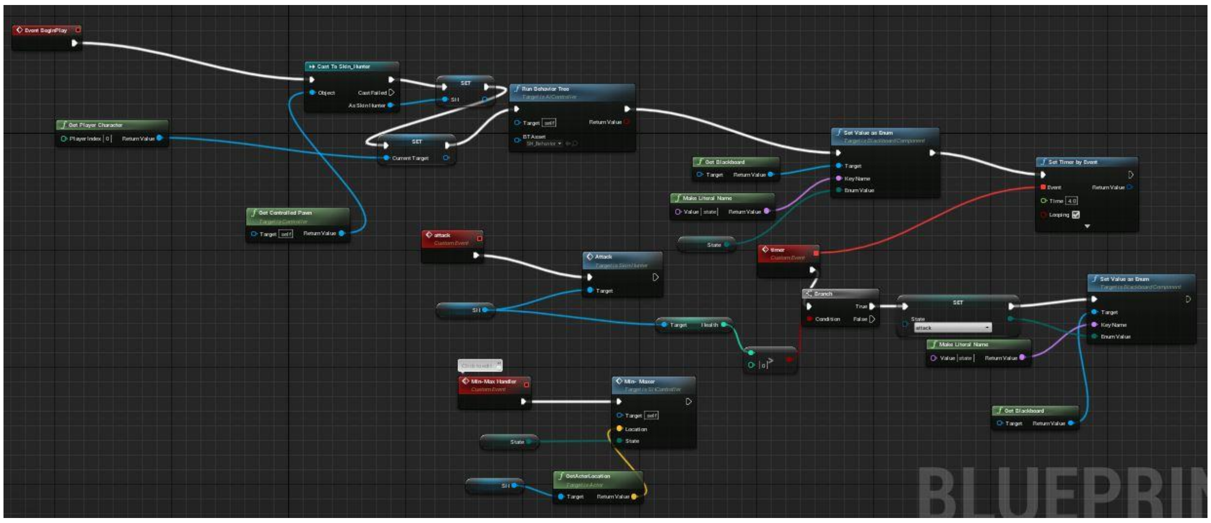
```
#include "MyProject.h"
#include "MyRecastNavMesh.h"

AMyRecastNavMesh::AMyRecastNavMesh(const FObjectInitializer& ObjectInitializer)
    : Super(ObjectInitializer)
{
    if (DefaultQueryFilter.IsValid())
    {
        DefaultQueryFilter->SetFilterImplementation(dynamic_cast<const INavigationQueryFilterInterface*>(&DefaultNavFilter));
    }
}
```

This will mean that every path-finding query, by default, will use this solution, unless a QueryFilter has been chosen on PathFind. To define a query filter, one can take a look at URecastFilter_UseDefaultArea. For example, the Move To node in Behavior Trees allow you to specify a move filter. If none is specified, it will use the default.

4 GAME FLOW OF CONTROL

4.1 SH_controller Class



Blueprint for SH_controller class

SH_Controller class is a child class of AI_Controller class, which is a pre-existing class in Unreal Engine. It is responsible for giving out instructions to the AI (Skin_Hunter) class to carry out its written logic behind movement, action, etc. We have programmed the class to carry out some basic functions whenever instantiated:

1. Set the current target of AI: This sets up the current target variable (vector) to player (Bryan), which will be used in the Behaviour Tree later.

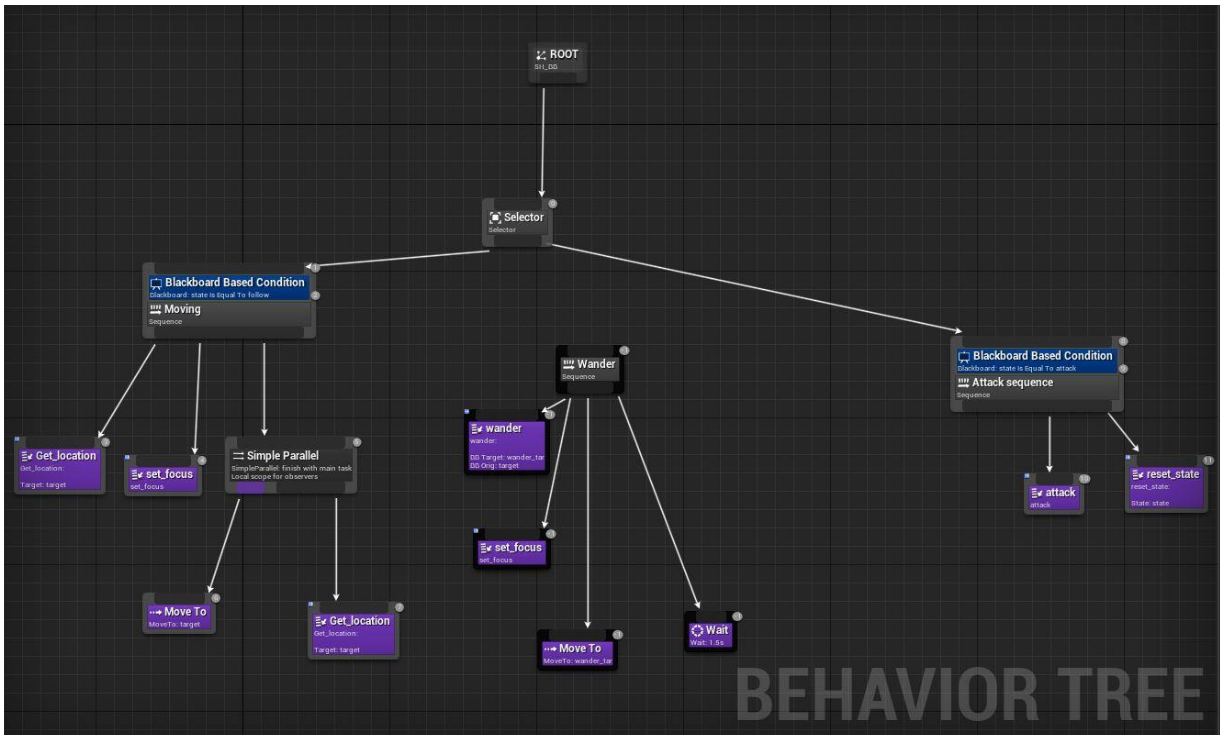
2. Call Behaviour Tree: Behaviour Tree is run through this class.

3. Reset State: Once the behaviour tree has been run, the enum 'State' is set to 'follow'. It is set to attack in the following cases:

   a. The player is in the attackable radius (the AI does not have to move to attack).
   b. The threshold time limit (4s) has passed since last attack.

4. Attack function: This is a member of the class but is called only via the behaviour tree. This further calls the attack function in Skin_Hunter class, which has the primary attack logic.

4.2 Behaviour Tree



Blueprint for Behaviour Tree

Once the SH_Controller class shifts flow of command to the Behaviour Tree, a custom conditional statement Selector decides the direction of the flow of command.

If the condition of the Moving Sequencer (Sequencer is another custom conditional statement) is met, i.e., the value of enum is set to 'Follow', then the Moving Sequencer runs it constituent blocks from left to right.

The location of Bryan is retrieved and stored in the Target variable. Set_focus moves the direction of Skin Hunter towards Bryan. Simple Parallel block alternates between its constituents for small intervals of time, thus moving Skin Hunter towards Bryan a little, then updating the direction in case Bryan has moved. This ensures the optimal direction is taken by Skin Hunter to move towards Bryan.

If the condition of the Moving Sequencer is not met and the Attack Sequencer is met, i.e., the value of enum is set to 'Attack, then the Attack Sequencer runs it constituent blocks from left to right.

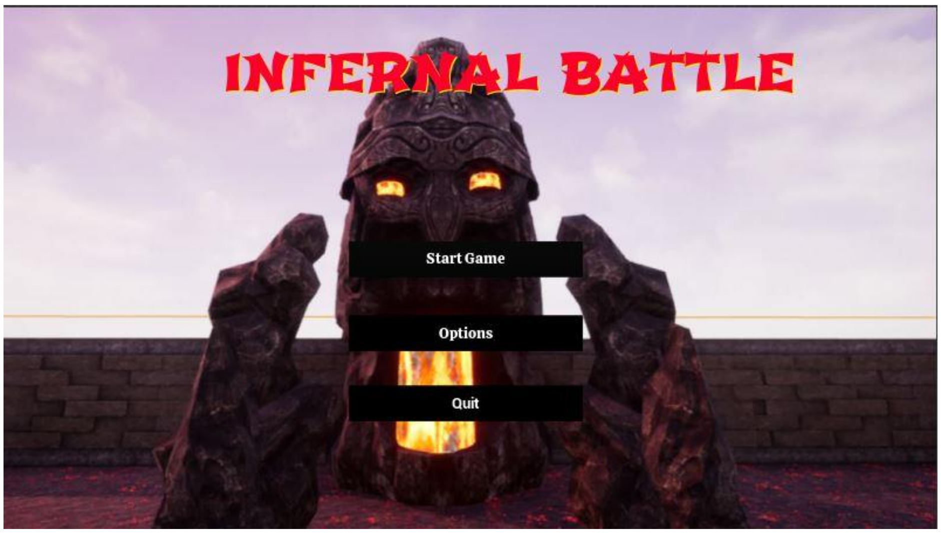
The attack function calls the attack function of the Skin Hunter Class, prevents Skin Hunter from accepting any moving commands, displays the attack animontage, and gives back control for another attack/movement to Skin Hunter after 2 seconds since attack function is called. The reset state sets the value of the enum back to 'Follow', and the flow of command goes back to the SH_Controller blueprint.

4.3 Collision Detection Mechanism

We have devised an intricate mechanism for detecting whether the player (Bryan) or the agent (skin hunter) have been hit by the other player in the game. For detecting this, we have put invisible box collision elements at both left and right arms of the players while constructing the character blueprint classes.

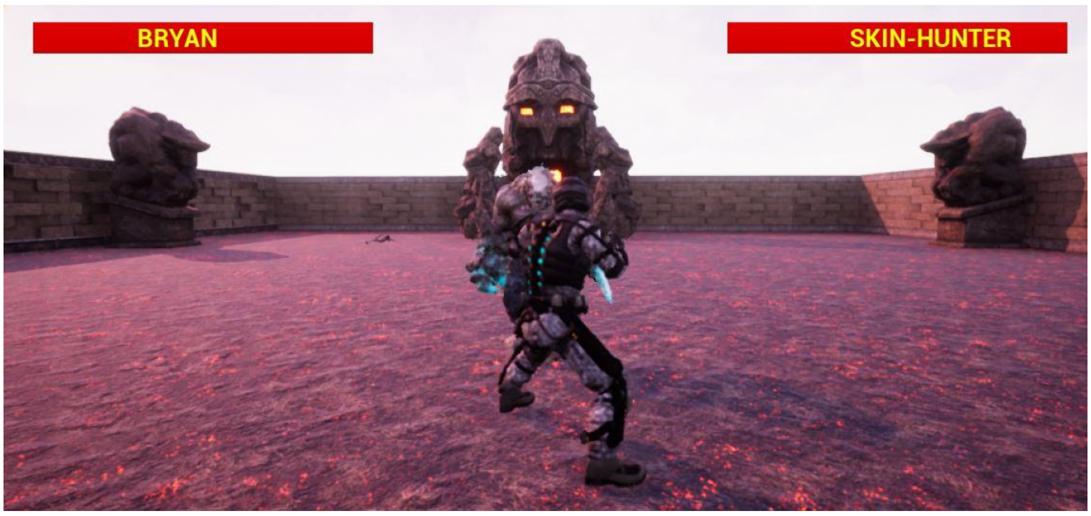
The invisible boundaries keep a check so that nothing violates it. This is managed by the function begin_overlap(), which checks the same. Whenever the invisible boundaries of the players get violated, a bool check variable called is_punching is checked to verify if the other player was punching at that time. If that was indeed the case, then the health bar (initiated with 100 health points) gets deducted by a value of 10 health points each time a hit is encountered.

5 GAME UI

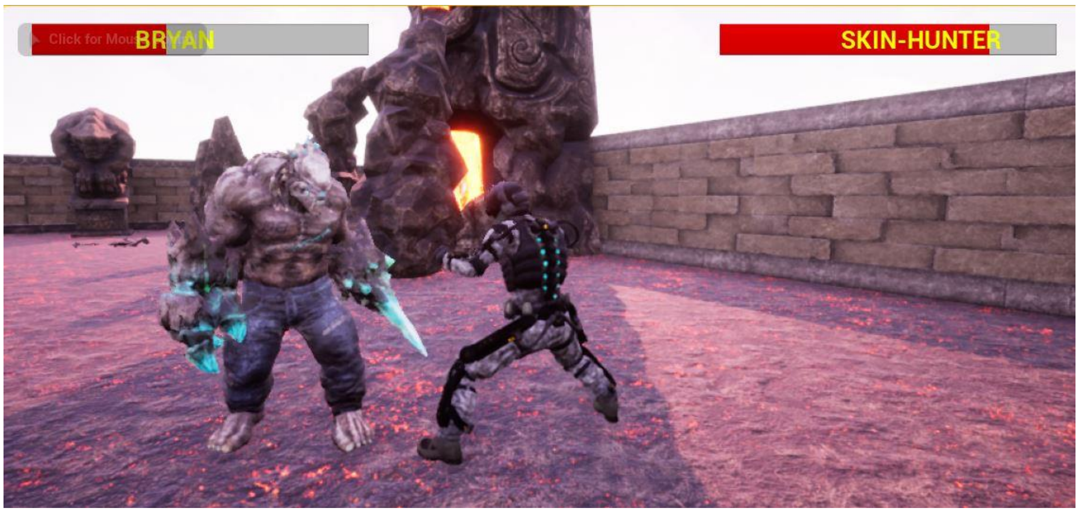


Main Menu

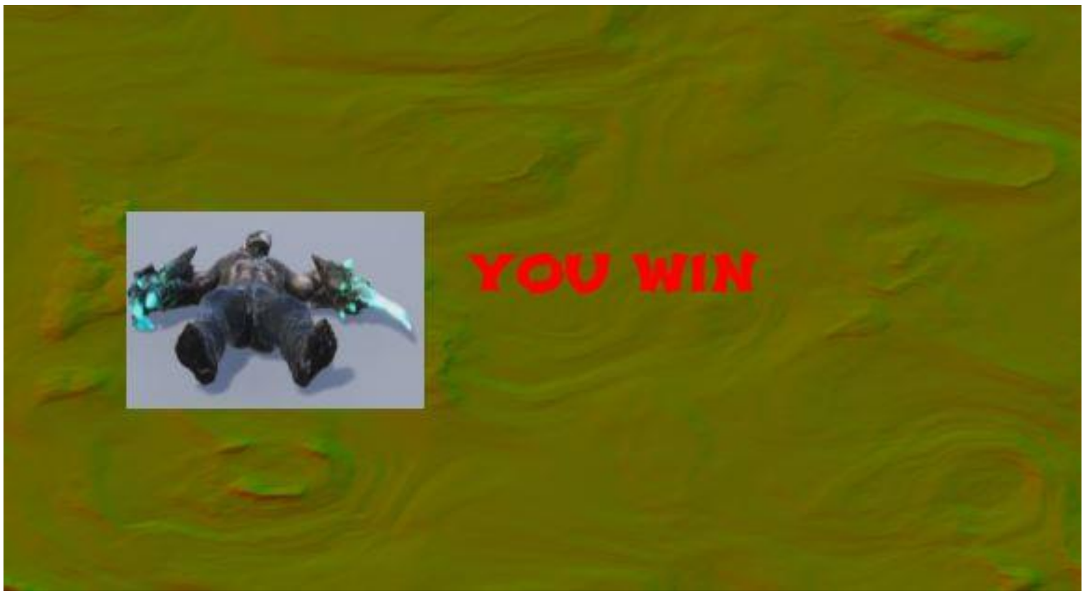
Game Arena



Gameplay



Gameplay



Winning Page

6 FUTURE WORK

We intend to take this project further, and some of the features in our roadmap include:

1. Adding multiple playable characters to the game.
2. Adding newer and trickier levels/maps to the game.
3. Refining AI of the game to incorporate 2-3 NPCs at the same time.
4. Add 'Special Move' feature to the game to make it more exciting.
5. Widen the number of players by making it playable over the internet.
6. Add more movement modes like flying and swimming.
7. Make the game environment more reactive to characters.
8. Better the in-game graphics.

7 REFERENCES

1. https://docs.unrealengine.com/en-US/AnimatingObjects/index.html
2. https://docs.unrealengine.com/en-US/Basics/Actors/index.html
3. https://www.oreilly.com/library/view/unreal-engine-4/9781784393120/ch04s03.html
4. https://www.nielsvandermolen.com/how-to-implement-custom-pathfinding-a-cc-in-unreal-engine-4/
5. https://medium.com/@mscansian/a-with-navigation-meshes-246fd9e72424
6. https://www.nielsvandermolen.com/how-to-implement-custom-pathfinding-a-cc-in-unreal-engine-4/