ADAPTIVE AND ADVANCED BEHAVIOUR OF ARTIFICAL INTELLIGENCE IN COMPUTER GAMES

A thesis submitted by

HARI SUBRAMANI

s5319307

In partial fulfilment of the requirement of the award of

MSc Computer Animation and Visual Effects

National Centre for Computer Animation

Bournemouth University

Submission Date: 22/08/2021

Word Count: 8220

ABSTRACT

In this thesis, we will be investigating and implementing the adaptive and advanced behaviour of artificial intelligence (AI) in video games. In addition to basic techniques such as pathfinding, adaptive behaviour is introduced through reinforced learning as well as complex decision-making based on the response of the playable character.

The first scenario is the implementation of a combat level between the playable character and an individual AI where the AI shows adaptive behaviour by reading the data of the playable character as well as making complex decisions. The second scenario will demonstrate the movement tactics and coordinated response of an AI squad against the playable character thus simulating advanced behaviour.

ACKNOWLEDGEMENT

I would like to thank my supervisor, Professor Jon Macey for his advice and guidance in driving this project forward and triggering my enthusiasm in other areas of computer graphics.

I would also wish to thank the rest of the teaching staff in the National Centre for Computer Animation for directly as well as indirectly aiding me over the year to hone my skills in various subject areas in and out of this course.

Finally, I wish to thank the unreal engine community for helping the development of this project at https://forums.unrealengine.com/c/community/12.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Computer games have evolved significantly over the years. One of the earliest games to use AI was Nim, a mathematical strategy game. Pac-man, another game released in 1980 uses pathfinding in AI to chase down the player. These two games along with countless others before the 21st century uses simple AI movement and response to make games an immersive experience with competitive AI. But these AI entities will follow a repetitive pattern with no complex decision making and not adapting to any style of play by the end-user or player. Recent games have employed different decision-making techniques to give programmed responses like patrolling the environment, taking cover while engaging the player character, etc. but there is less focus on adaptive AI with tactical movement, coordinated team response with a few exceptions.

This thesis will describe the design and the implementation of adaptive AI as well as its advanced behaviour. The project has two scenarios. The first scenario is about a single AI engaging with the player showing an adaptive response to the player's style of play. The second scenario is about an AI team, working in a coordinated way along with moving tactically to engage the player character.

Chapter 2 covers previous work and examples in designing Artificial Intelligence with complex behaviours and implementing them in various genres of video games.

Chapter 3 introduces concepts of Behaviour tree, Perception, Environment query system (EQS) used in the game engine Unreal engine with references to tactical and military-like movement and response by team AI employed against the player.

Chapter 4 describes the design and implementation of the project detailing the scenarios about the individual as well as the team AI with the use of in-engine tools and programming along with details of the development process.

Chapter 5 covers the results and observations including the technical difficulties encountered during the development of the project.

Chapter 6 covers the conclusion and future work for the project described in this thesis.

Appendix A contains the source code for this project written in C++ programming language.

CHAPTER 2

PREVIOUS WORK

2.1 Advanced AI

Modern video games use in-engine tools in combination with programming to design and evoke complex behavioural responses from AI entities. These responses are pre-defined depending on the environment, situation, and genre of the game. For example, Grand Theft Auto V (Rockstar Games, 2013) is a big, open-world game with numerous non-playable characters (NPC) populating the vast area of the map. All these NPCs are of course AI entities each programmed to perform different operations depending on the role and also interacting with each other on occasions. For example, if the player hits an NPC's vehicle who happens to be an ordinary citizen, it will evoke a response of either cursing and attacking the player or flee away but if that vehicle belongs to the police NPC, then a chase and bust response is triggered against the player where even helicopters and big armoured trucks are called in as backup as shown in Figure 2.1. These responses are pre-programmed and designed to work based on the type of NPC in the world and most of them are often repetitive confining to a particular type (Ashwin et al. 2007).

Another example is the enemy AI entities in games like Assassins creed Valhalla (Ubisoft, 2020) where different types of enemies on the battlefield will have their own set of attack tactics against the player where each set is pre-programmed for a particular type of enemy as shown in Figure 2.2. These responses are often disorganized with no coordination between the type of enemies to make it more challenging for the player. But these improvements are significant when compared to the simple chase and attack movement in early video games.

Figure 2.1 Police chase response of AI in Grand Theft Auto V

Figure 2.2 Different enemy types in Assassins Creed Valhalla

2.2 Adaptive AI

Examples mentioned in 2.2 is largely about pre-programmed behaviours of AI in various games but with a few notable exceptions, almost all the AI used in games have a set of pre-programmed responses which would not adapt to the player's style of play. Adaptive AI is an area largely unexplored due to the notion of missing the element of fun in video games. People play games to have a good, fun time rather than tussling with an adaptive, tough AI and not make progress. But this notion is kept in the mind of developers for a long time since the inception of game development and the evolution of the mindset of gamers who are willing to explore the unknown, will pave the way into making more adaptive AI entities to give a realistic, immersive experience in the future. It may well have applications beyond gaming, like training military personnel to increase their combat skills when having a virtual fight against an AI enemy adapting to your moves.

There are also attempts in making AIs in games adaptive. Middle Earth: Shadow of Mordor game has a system called Nemesis in which if a player is defeated by a type of enemy AI, it will rose through the ranks of the enemy hierarchy as well as remembering the scars inflicted upon it and battle tactics of the player. This will provide a new level of immersion when the next time the player encounters that same AI, the responses are enhanced thus giving a stern challenge.

The Design of adaptive AI needs to meet computational requirements such as speed, effectiveness, robustness, and efficiency as well as functional requirements such as clarity, variety, consistency, and scalability (Pieter et al. 2006).

Figure 2.3 Nemesis System for AI in Middle Earth: Shadow of Mordor

CHAPTER 3

TECHNICAL BACKGROUND

3.1 Behaviour Tree

Decision-making is pivotal in allowing AI to respond to different situations. The described project uses Unreal engine, a powerful game engine with a lot of in-engine tools for development. To make decisions depending on different actions, Behaviour Tree is used. This behaviour tree asset is used to execute different branches of logic for different needs. It also relies on another asset called Blackboard which is touted as the "Brain" for the behaviour tree where it can have many user-defined Keys to hold and use in the behaviour tree logic. Some of the key types including but not limited to are bool, int, float, vector, object which can be declared in the blackboard and modified in any AI controller class using the behaviour tree as shown in Figure 3.1. The typical workflow for this setup is to create a blackboard asset, add any number of blackboard keys, and use them in the behaviour tree that uses the blackboard asset. These keys are used in the behaviour tree to switch between different logic or can be a part of a task-driven inside the logic. The behaviour tree can also have any number of tasks and services to carry out the decision-making logic.

Apart from the pre-programmed ones, customized tasks and services can be created through blueprints or C++ programming to achieve the desired results. Typically, a behaviour tree executes logic from left to right but it is also possible to handle concurrent behaviour through Simple Parallel nodes, services, and decorators.

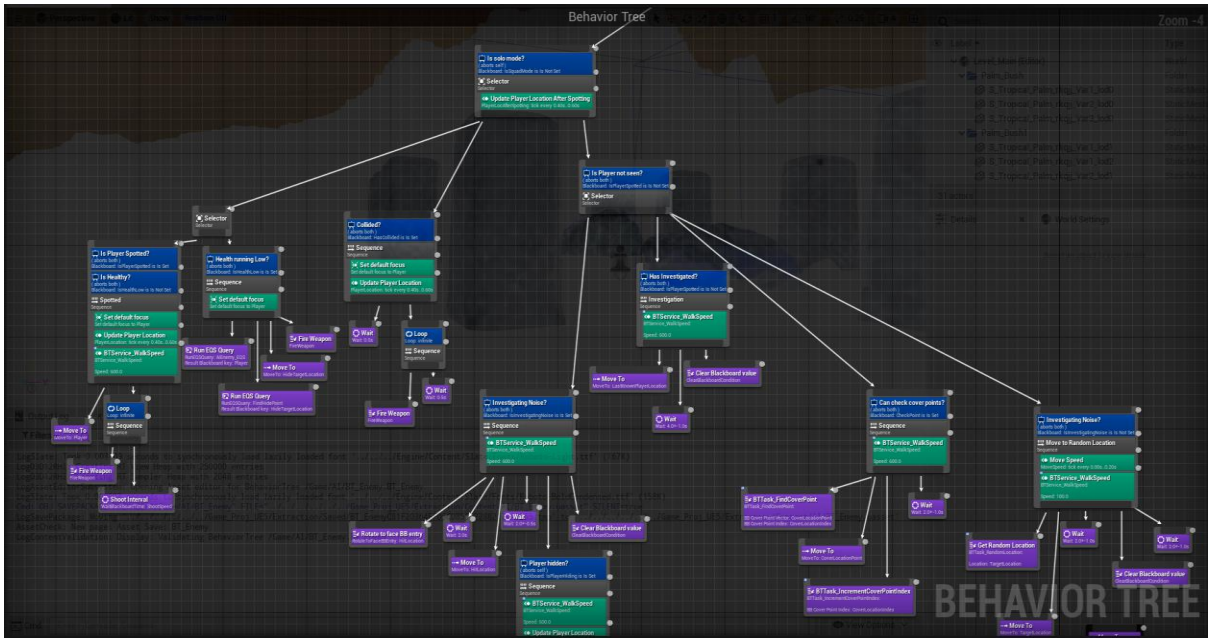

Figure 3.1 Behaviour Tree asset used by Enemy AI for decision making

3.2 Perception

Realistic human response to events like sight, touch, hearing, etc. can also be simulated in the game world through modern tools and techniques. This response is collectively called Perception. When perception can be added to AI, it would greatly enhance the simulation of interaction in the game world. Unreal engine's AI Perception is used in this project to respond to events such as sight, hearing, and touch. Perception components are added to the enemy AI controller class and each component can have a particular dominant sense of stimulus source. For example, if sight perception needs to be configured, a perception component is added to the controller class and the dominant sense is set to AI sight config. A few parameters such as sight range, peripheral vision angle are also configured. If there is no blockage to the line of sight, the sight perception is updated after spotting an actor in the scene as shown in Figure 3.3.

This method of configuration works for hearing (see Figure 3.2) as well as touch perception events. The response to any configured perception is obtained and utilized in the controller class to set a customized variable or even setting the blackboard keys to be used in the behaviour tree to drive a certain logic.

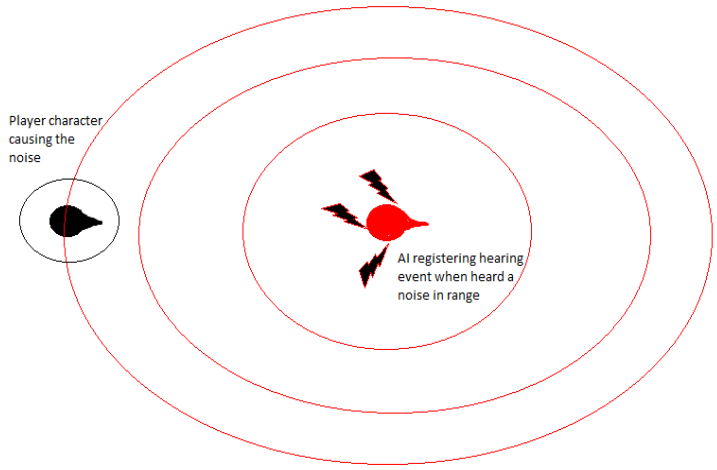

Figure 3.2 Illustration of AI hearing perception event

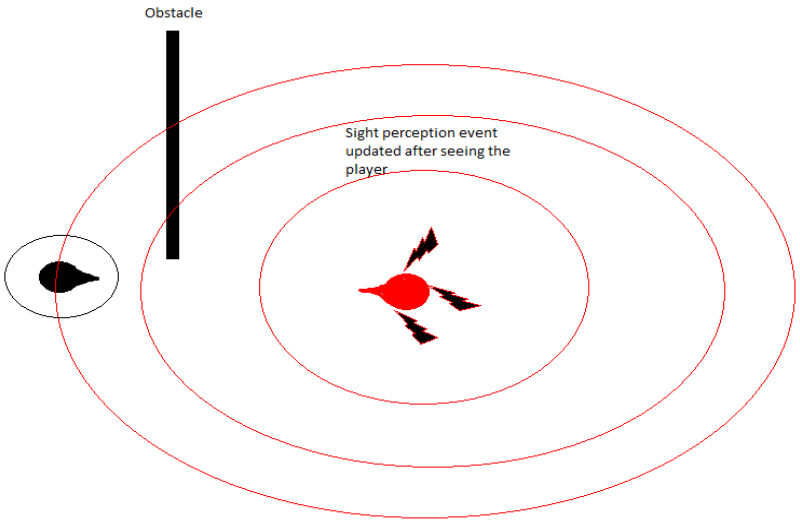

Figure 3.3 Illustration of AI sight perception where the perception event is updated after having a clear line of sight to the character.

3.3 Environment Query System

Unreal engine comes with tools and features for AI development. One such feature is called Environment Query System (EQS) which can be used to collect information about the environment inside the game level.

The data collected from the environment is then fed to a Generator in which the system can query the data with questions and user-defined tests and returning the best possible result to be used in a behaviour tree task.

For example, if the AI needs to find cover points away from the player, an EQS query is run in the behaviour tree where the query system collects data from the environment through points of weighted/filtered results drawn inside the navigable bounds. The data is then queried against different user-defined tests added to the generator. Some examples of the tests include finding the distance between the querier and any context added (e.g., Player), tracing to the added context, finding the dot product between the querier and the context added, etc. as shown in Figure 3.4.

All the added tests can be run to filter and score cover points placed in the environment and finally the best possible cover point satisfying the tests in the generator is filtered and returned. This point can be used as the best possible location to take cover from the player.

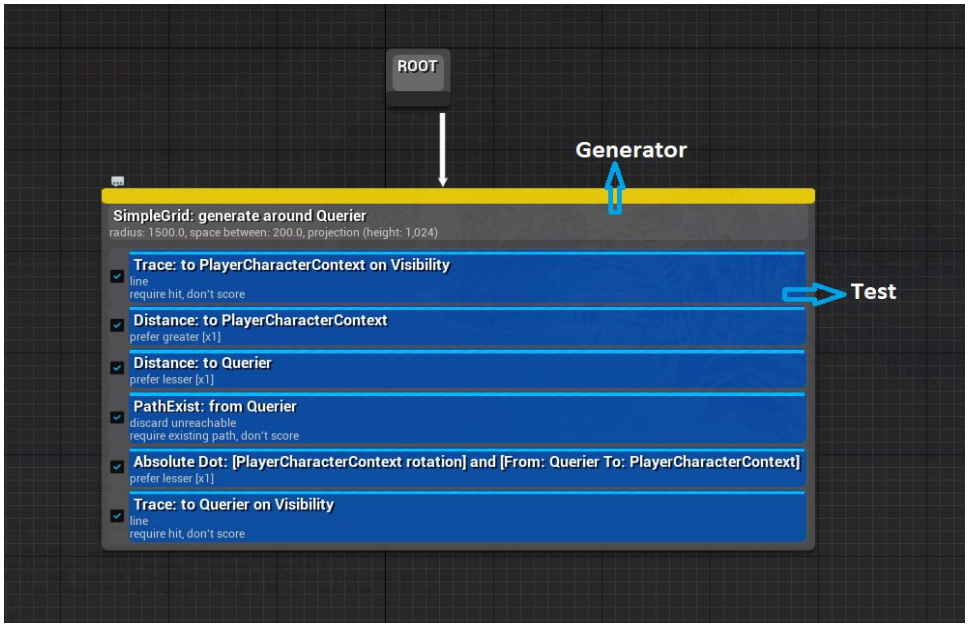

Figure 3.4 An EQS generator having multiple tests to filter and score the best possible result.

Figure 3.5 Preview of an EQS query by filtered results placed in the game environment in the form of spheres.

The results of the query can be previewed inside the viewport editor of the game engine to adjust the test parameters as well as add new custom tests to the generator by observing real-time calculations of scores by weighted/filtered results in the form of spheres placed in the environment as shown in Figure 3.5.

CHAPTER 4

DEVELOPMENT

4.1 Objectives

The project described within this thesis has two scenarios.

1. Scenario 1 is to design and implement an individual AI with adaptive as well as decision-making skills and place it against the player in a game level
2. Scenario 2 is to create a team AI demonstrating tactical movement and coordinate response against the player placed in another level.

4.2 Scope

The project described in this thesis focuses only on the technical aspects as well as the working of the artificial intelligence entities within the game.

The quality of level design, HUD, 3D models, animation, visual effects, and rendering are not considered during the development of the project and are outside the scope.

To demonstrate the working of the artificial intelligence inside the game environment, Unreal engine 5 application is used rather than developing a system from scratch. This enabled the ease of implementing the project with good visual detail. The tools for AI in the game engine are also exploited to provide a complete solution to appeal visually.

4.3 Character design

4.3.1 Player Character

This is a C++ class written based on the ACharacter class provided in the engine. This class acts as a base class to design a blueprint class inside unreal engine called BP_PlayerCharacter which is to be used for the player. Movement controls, health, and damage events are implemented inside the class to be derived by the blueprint class. The source code for this PlayerCharacter class is presented in Appendix A.

4.3.2 Enemy AI

4.3.2.1 BP_AICharacter

A blueprint class which has the PlayerCharacter class as the base class but acts as the enemy AI when spawned inside the level. This will retain the same movement controls, health, and damage events but when spawned inside the level, acts as the enemy AI. The blueprint class contains a controller class called Enemy, a C++ class that has the AAIController class as the base class. The source code for the Enemy class is presented in Appendix A. This controller class enables the character to behave like an AI when placed inside the level along with the player. The enemy class contains all the decision-making, tracing as well as blackboard key setting logic for the behaviour tree declared in this class. The adaptive features are also designed and implemented in addition to other C++ classes for the functioning of the enemy AI. This enemy class acts as the base for the blueprint class called BP_Enemy which will implement perception and handling the inputs of blackboard keys to be used in the behaviour tree.

4.3.2.2 BP_AICharacterAlt

This character blueprint class is designed the same way as the BP_AICharacter class but it is designed to work in the enemy AI team for the demonstration of the second scenario. This class does not have adaptive capabilities and it also has its own blueprint enemy controller class called BP_EnemyAlt derived from the Enemy class for customizing individual actions. It also has its own behaviour tree and blackboard assets to execute actions required to work as a team player.

4.3.2.3 BP_AIFlanker

This is another character blueprint class designed the same way as the BP_AICharacterAlt class to work in a team environment. It also has its own blueprint enemy controller class called BP_EnemyFlanker derived from the Enemy class along with its own behaviour tree but shares the same blackboard asset as the BP_Enemy controller class.

4.4 Implementation

The two scenarios mentioned in the objectives are implemented in two different levels and they are explained as follows.

4.4.1 Scenario 1 – Player vs AI (Solo)

In this scenario, the player is placed against an AI which is the BP_AICharacter actor. Different responses are evoked through decision making in the behaviour tree and adaptive responses are also implemented which are detailed as follows.

4.4.1.1 Decision making

a) Patrolling:

When the game starts, initially the player is spawned at a distance not within the sights of the AI and at this stage, the AI starts the patrolling task defined in the behaviour tree (see Figure 4.1) by picking random location points (three-dimensional vectors) and moving back and forth of selected random locations. These events will run in sequence until the AI encounters a different event needing a different response.

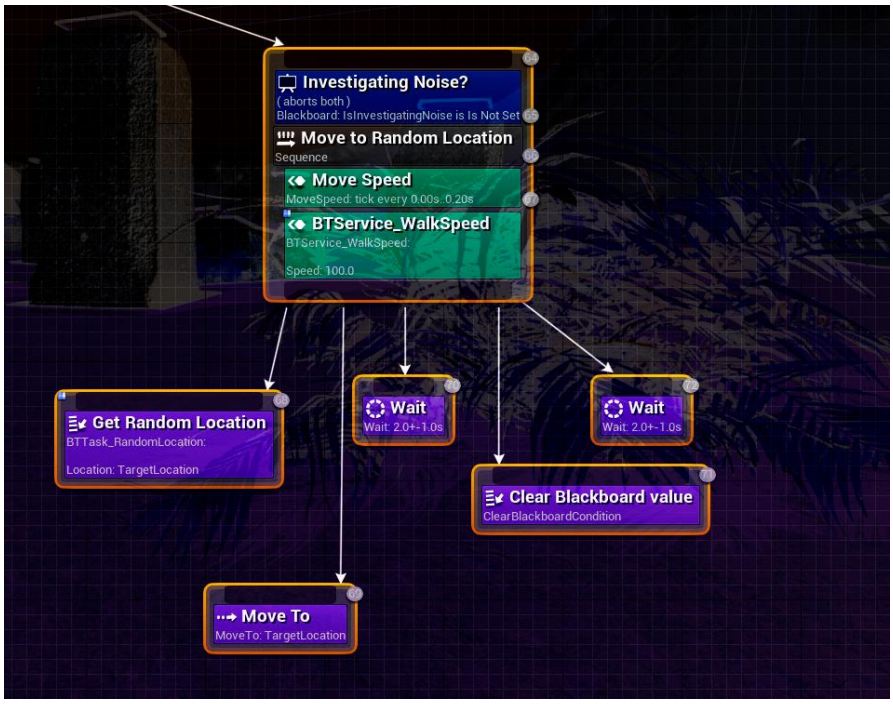

Figure 4.1 Behaviour tree sequence of events for AI patrolling task.

Boolean blackboard keys are used to facilitate such conditional response and these keys are used in the blackboard decorator placed on the task. For example, in Figure 4.1, the sequence "Move to Random Location" has a blackboard key IsInvestigatingNoise in a decorator called "Investigating Noise?" where the decorator will decide whether that sequence can be started or not. It is done by setting the blackboard key IsInvestigatingNoise which can be accessed and changed in the controller class e.g., BP_Enemy.

b) Engaging player on sight:

While patrolling, when the player comes in the range and angle of the sight perception, a stimulus event is registered (see Figure 4.2), and the response is recorded to a blackboard key which will start the sequential response of moving towards the player and fire weapon to cause damage as shown in Figure 4.3.

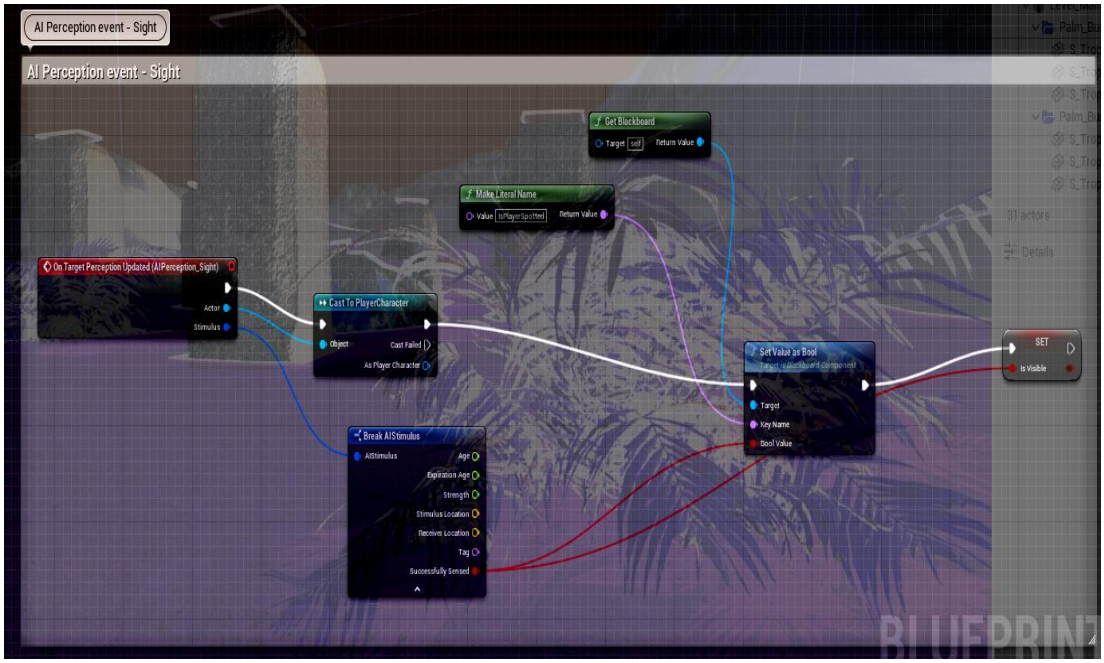

Figure 4.2 AI Perception event handling for AI sight in BP_Enemy blueprint class.

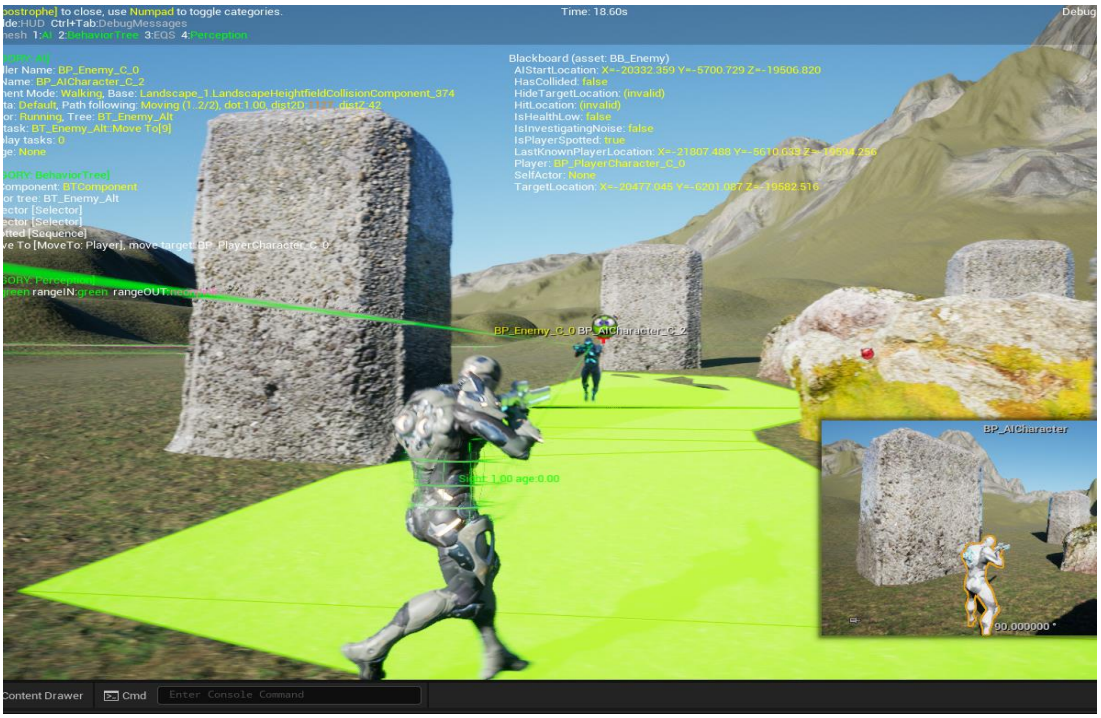

Figure 4.3 Enemy AI engaging with the player after having the player in its sights.

c) Investigation:

Two types of investigative events occur in this encounter. One is for investigating noises obtained through hearing perception events and the other is investigating the last known player location after losing sight of the player.

When the player fires a weapon and the bullet impact sound is within the hearing range of the AI, the hearing perception event is updated triggering the sequential event in the behaviour tree (see Figure 4.4) through a blackboard key where the AI will move to the location of the bullet impact and investigate.

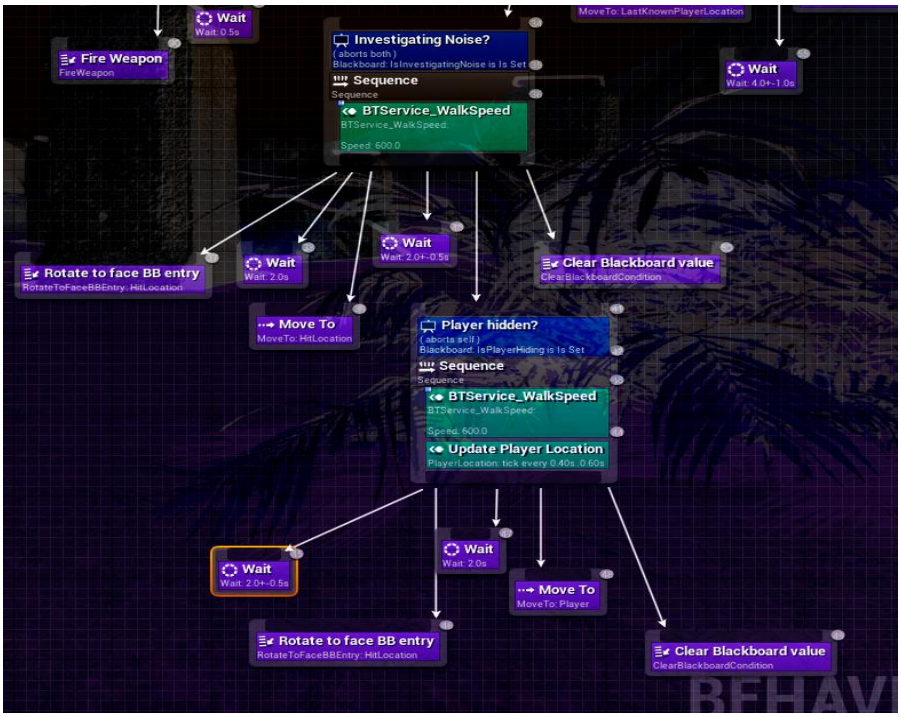

Figure 4.4 Behaviour tree sequences for AI investigative tasks of hearing perception event.

If the AI loses sight of the player, it will move to the last seen location of the player (see Figure 4.5) and investigate. If it has the player again in its sights nearer to the last seen location, it will re-engage.

Figure 4.5 AI investigating the last seen location of the player.

4.4.1.2 Adaptive AI

AI entities that adapt to the player's style offer a different immersive experience and, in the project, there are two tasks where the AI can adapt to the player's style of play.

a) Shoot Intervals:

The player character has a weapon attached as a component and the weapon class has logic for pulling the trigger, tracing, and hit events. Every time the player pulls the trigger, a clock is activated to find the difference of time in seconds between the first shot and the next time the player pulls the trigger for another shot. This time difference is then added to an array called ShootIntervalData and the array values are then written to a file. This is presented in the source code for the Weapon class in Appendix A.

The system keeps on writing the ShootIntervalData's values to the file till the player plays the same level thrice. When the gameplay starts for the fourth time, the Enemy class will read the data from the file, store it to an array and check for different interval times. This is presented in the source code for the Enemy class in Appendix A. Whichever interval has the most entries, it will then be fed to a blackboard key called ShootSpeed into the respective behaviour tree sequence thus updating the shooting speed of the enemy AI as shown in Figure 4.6.

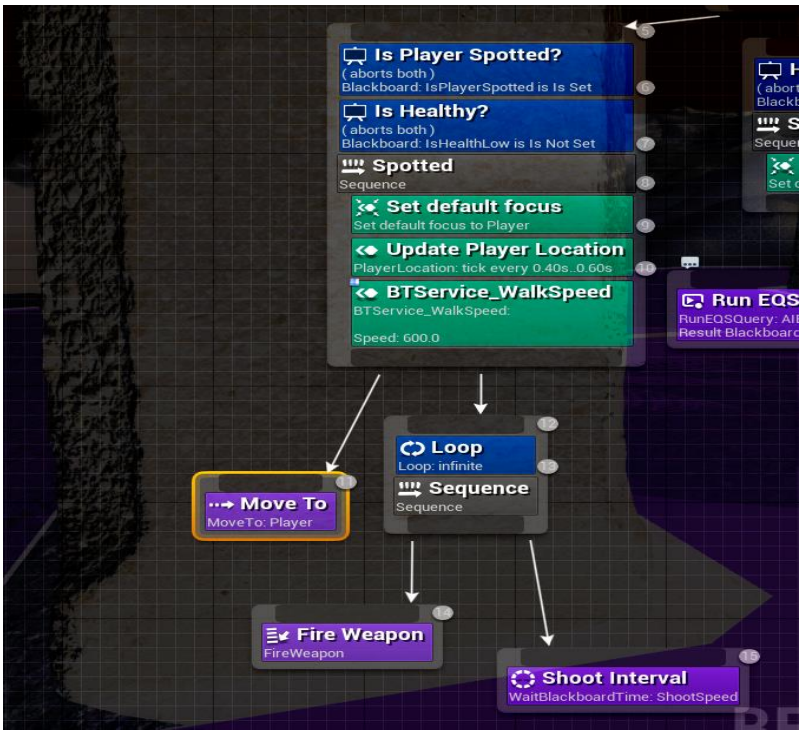

Figure 4.6 Behaviour tree sequence with the Shoot Interval task having ShootSpeed key.

b) Checking possible cover locations:

The AI starts engaging the player once it has the player in its sights. When it loses sight for the first time and then regains it, the location of the player at which the sight is regained is added

to an array called PlayerLocationData and the array values are then written to a file. This logic is presented in the source code for the Enemy class in Appendix A.

At the start of play, possible cover locations such as the bushes placed inside the level are also added to the array and written to a file. When the player starts the level for the fourth time, instead of using random location vectors in the game environment, the AI will check and move to each cover location vector read from the file where the location data is written as shown in Figure 4.7. This will allow the AI to increase the possibility of finding and engaging the player.

Figure 4.7 AI checking possible cover points including bushes after reading recorded data.

4.4.2 Scenario 2 – Player vs AI (Team)

In this scenario, the player is put against an AI team comprising of the characters BP_AICharacter, BP_AICharacterAlt, and BP_AIFlanker. The focus in this scenario is to demonstrate tactical movement and coordinated response of the AI team.

4.4.2.1 Approaching the Player:

The level for this scenario has a lot of objects to take cover and these cover objects are used to approach the player in a tactical way. The first challenge is to establish a line of sight towards the player and find the appropriate cover locations to approach the player. This is done using the function LineTraceMultiByObjectType. A trace line is established between the AI and the player (see Figure 4.8) and whichever static objects the line hits on the way, the information about those objects including the object type, tag, location, impact points, etc. are then stored in a variable of type FHitResult. The logic is presented in the source code for the Enemy class in Appendix A.

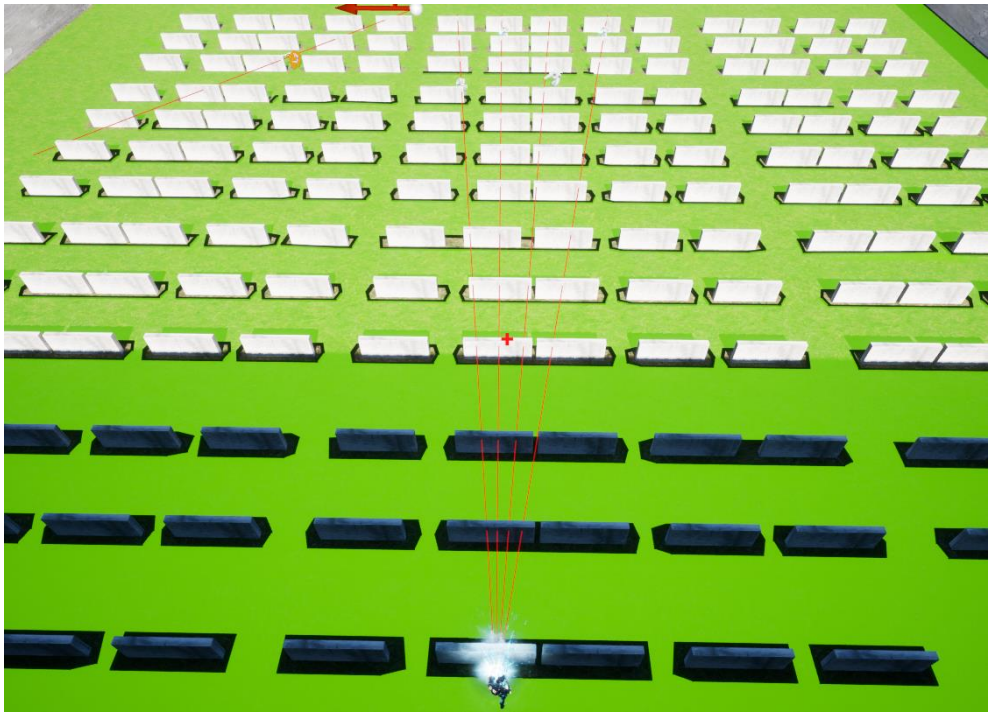

Figure 4.8 AI characters establishing line of sight towards the player.

The impact point of an object where the trace line hits is taken and stored in an array called CoverObjectLocations. The logic is presented in the source code for the Enemy class in Appendix A. Each element of the array is then accessed and iterated through a behaviour tree task, and it can be used as a cover location by the AI thus facilitating the approach with the chance of taking less damage by the player as shown in Figure 4.9.

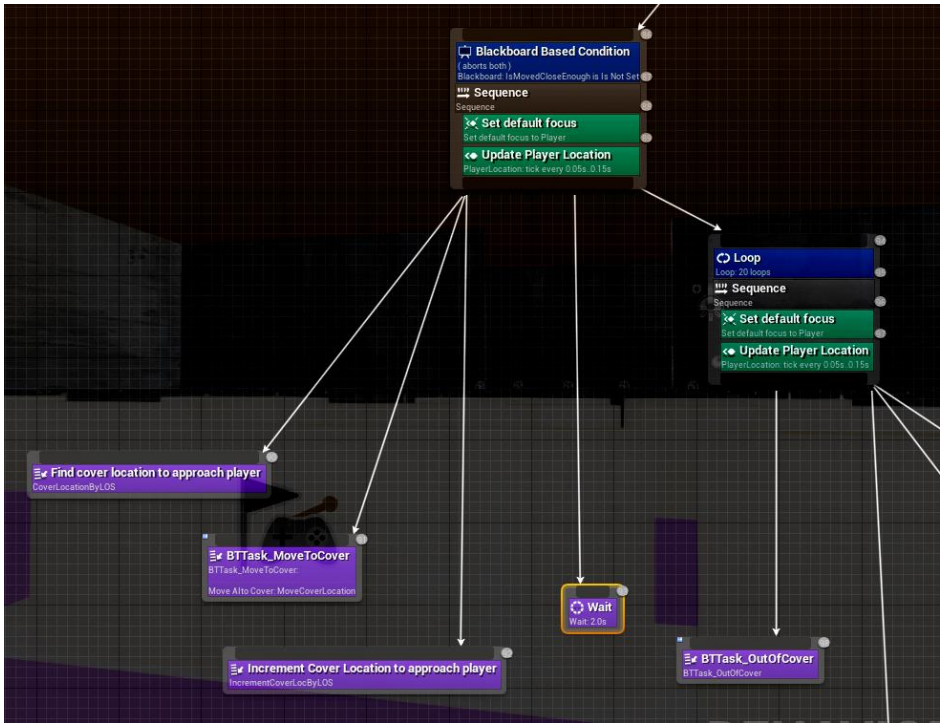

Figure 4.9 Behaviour tree sequence of AI finding and moving to cover points incrementally.

4.4.2.2 Tactical movement and Coordinated response:

Good coordination between the AI characters will not give any breathing room for the player to counterattack. After establishing the line of sight, the characters, BP_AICharacter and BP_AICharacterAlt will move in pairs while approaching the player.

When the first pair of BP_AICharacter type moves to a cover location, the second pair of BP_AICharacterAlt type will keep firing towards the player location to keep the player pinned down. When the first pair reaches a cover location, it will come out of cover and keep firing towards the player location while the other pair will start moving to their respective cover locations as shown in Figure 4.10.

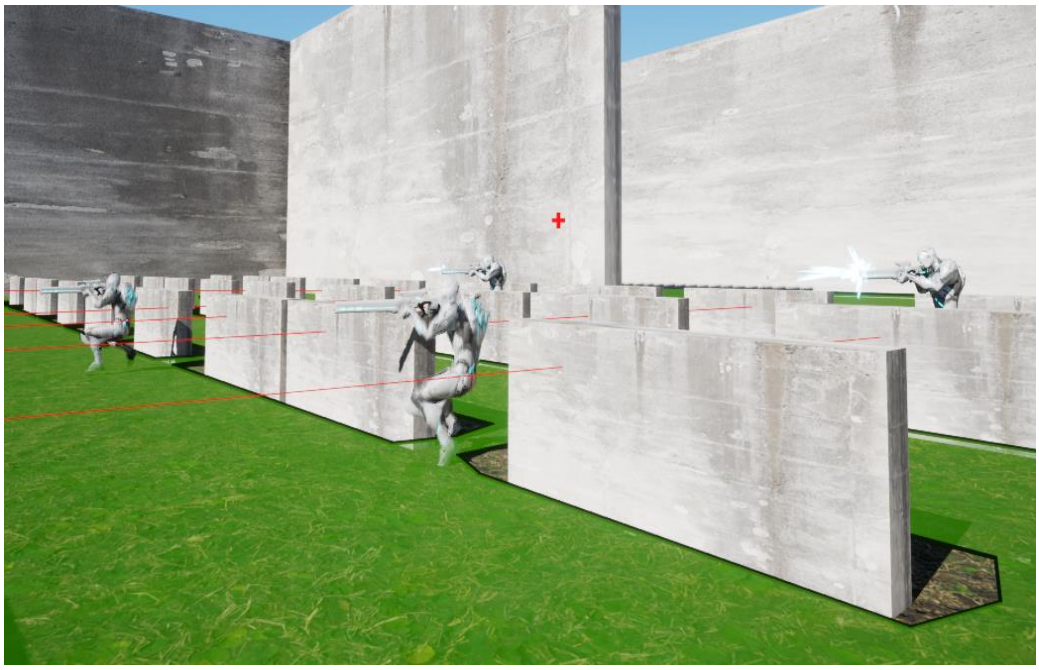

Figure 4.10 AI characters demonstrating tactical movement and coordinated response.

This type of tactical movement and coordinated engagement will pose a stern challenge for the player to defeat the AI team. The AI characters will re-trace the line of sight every 11 seconds towards the player to cover the possible condition that the player moved to a different location and recalculate the cover locations to approach the player. This is presented in the source code for the Enemy class in Appendix A.

4.4.2.3 Flanking:

While the AI characters BP_AICharacter and BP_AICharacterAlt work in pairs to engage with the player, another AI character called BP_AIFlanker will try to flank the player by moving away from the direct line of sight and approach the player sneakily while the other AI pairs keep the player busy as shown in Figure 4.11.

Instead of establishing a line of sight directly towards the player, it will shoot a trace line away from the player by adding the vector value of the player location and another vector called Flank vector which is declared in its controller class BP_EnemyFlanker.

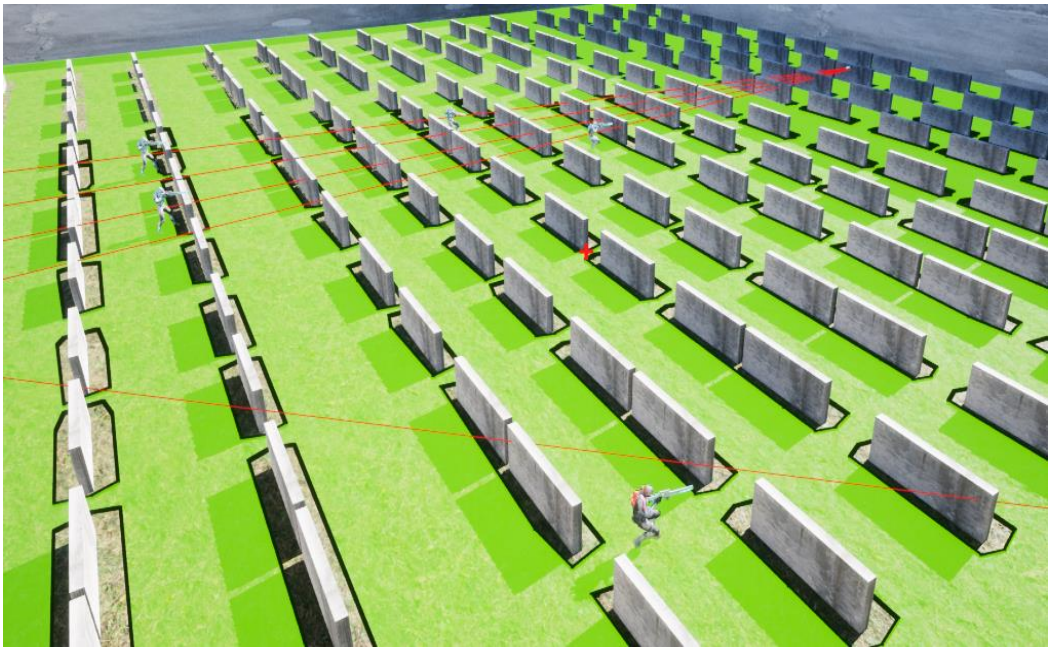

Figure 4.11 AI Enemy flanker seen at the bottom of the image moving away from the direct line of sight towards the player.

After moving away to a position considerably from the line of sight, the AI will move in a straight line parallel to the player's position and, once in line with the player's position, it will proceed to move towards the player and fire its weapon to damage the player. If the player comes near the flanker while it is moving, it will immediately engage and fire at the player and also gives chase if moved away. If the player hides away from the flanker's sight, it will proceed to investigate the last see position of the player.

Scenario 2 demonstrates that it is possible to design an AI team each having different tactics work together as a team to keep the gameplay immersive and pose challenges to the player.

CHAPTER 5

OBSERVATIONS

5.1 Performance:

5.1.1 AI in Scenario 1:

The AI is able to handle decision-making really well and carries out all the responses swiftly. Reading data and utilizing it for adaptive behaviour also works without any errors. When the health of the AI is low, it will abort other tasks and tries to flee from the player by breaking the line of sight. This requires finding the appropriate hide location and Environment Query System is used for that task.

It seems to be doing well by keeping the AI always away from the player by moving to the nearby hide location preferably behind static objects. However, when the player exhibits unpredictable movement in the approach while the AI is in cover, it can sometimes move towards the player's direction thus exposing itself to damage. When the AI reaches the hide point, if the player does not move closer to the AI, instead of remaining static, it keeps on moving back and forth for a short distance just around the static object which is also prone to exposure Currently, only three different speed intervals for shooting by the AI character are implemented where they can change depending upon the player's speed. This is presented in the source code for the Enemy class in Appendix A. After starting a level more than three times, the AI will adapt to the data read from the files and it will not revert to its default behaviour until all the entries in the file having the data for the number of start times are cleared manually. This applies to all the data needed to be read from files having the player's location as well as the interval between different shoot times. When the player is hidden inside a bush and the AI detects collision with the player, it has to fire at the player even in cover, but it will go into a frozen state until the player shows some movement away from the bush. Overall, the demonstration of the complex decision-making skills and adaptive behaviour of the AI in Scenario 1 proves to be successful.

5.1.2 AI Team in Scenario 2:

The AI team carries out the movement and responses alternatively thus simulating a combat manoeuvre approach towards the player. However, when coming close to the player, all the members are clustered behind a single cover object and sometimes hindering movement between them. This will be corrected when the player proceeds to move towards them where they will split up and engage with the player. Even though it is timed to abort shooting when one pair is already out of cover and shooting in the direction of the player, the other pair can come into the line of fire when near the player location but damage to each other is not considered to ease the implementation.

The AI flanker works as intended by choosing to move away from the direct line of sight and move towards a location parallel to the player location. However, the direction in which the flanker can move away is determined by the vector variable, Flank vector which is not set dynamically.

5.2 Technical difficulties during development:

Although the EQS system provides a way of querying the environment and retrieving data for utilization, the results are not always accurate leading to inconsistent performance. EQS was initially considered to use in scenario 2 to approach the player but due to inconsistent results, line of sight functionality is chosen which is far more accurate in finding cover locations between the player and the AI character.

Only one AI character was considered initially for usage throughout all the scenarios with varying roles but some of the coordinated response and movement was not possible even though all the blueprint controller classes of the respected character use the same base C++ class. Having different character classes each having its own AI controller class proved to be advantageous and was easy to add more modules and maintain.

CHAPTER 6

CONCLUSION AND FUTURE WORK

This project can be considered a successful one as it was able to demonstrate the adaptability, complex decision making, and coordination with tactical movement and response mentioned in the objectives. The framework is a base for further development and can be improved to add more features to the existing AI characters as well as introducing new techniques and character types. The process of the AI character engaging the player could have been improved instead of moving and firing in scenario 1 such as cover to cover fire using the static objects, ducking and diving with swift movement to evade line of fire, etc. to make the gameplay completely immersive. More events can be added in scenario 2 to force the player to come out of cover while pinned down by continuous fire. For example, throwing a grenade while providing continuous fire towards the player will force the player to move thereby increasing the chance of defeat. All these mentioned possible events and more scenarios can be added to the system which is also scalable and easy to maintain.

Observations between the packaged version of the game application from the in-engine development version and errors encountered:

a) The packaged version has the gameplay demo in an executable file which can throw fatal errors at times when either the player is defeated, or the player defeats the AI in both scenarios.
b) Unable to check the possible cover locations in the packaged version whereas, in the development version in the engine, the AI was able to check after the player re-plays the level more than three times. The number of times the player starts the level is written to a file StartCount and read in the Enemy class. In the packaged version, this count is not incremented.
c) Two user interface (UI) classes, one for the intro level and another UI widget class for displaying player info are appearing in an overlapped way inside the intro level. The UI widget class for player info is designed to appear only on the gameplay levels i.e., levels for scenarios 1 and 2.

All these issues are hoped to be resolved with further work in the future.

Possible future areas of work are listed as follows:

a) Introduce new ways of reading data other than file system such as gameplay videos to train the AI into adapting the player's style of play.
b) Combining Environment Query System and traditional line of sight methods to find the best possible approach to flank the player as well as improving the tactical movement and coordination between AI.
c) Introduce other AI entities which can switch roles at any level when required for an immersive engagement.

APPENDIX A : C++ Source Code

```
//-----------------------------------------------------------------------------------------------------------------------------------------------------------

/// @file Enemy.h

/// @brief This header file contains the function declarations and variables for all the features of a working AI character through the blueprint controllers BP_Enemy, BP_EnemyAlt, and BP_EnemyFlanker.

/// @version 1.0

/// @date 24/07/21

/// @class AEnemy

//-----------------------------------------------------------------------------------------------------------------------------------------------------------


#pragma once

#include "CoreMinimal.h"

#include "AIController.h"

#include "Enemy.generated.h"


class APlayerCharacter;

UCLASS()

class EXTRACTION_API AEnemy : public AAIController

{

	GENERATED_BODY()

public:

	AEnemy();

	~AEnemy();

protected:

	virtual void BeginPlay() override;

public:

	UPROPERTY(EditAnywhere, BlueprintReadWrite)

	TArray<FVector> BushLocations;


	UPROPERTY(EditAnywhere, BlueprintReadWrite)

	float CloseLocationLimit = 1000.0f;


	UPROPERTY(EditAnywhere, BlueprintReadWrite)

	FVector FlankVector;


	UPROPERTY(EditAnywhere, BlueprintReadWrite)

	bool IsAIEnemyCrouching;


	UPROPERTY(EditAnywhere, BlueprintReadWrite)

	bool IsSquadMode;
```

```
	UPROPERTY(EditAnywhere, BlueprintReadWrite)
	bool CanMoveAlternatively = false;

	UPROPERTY(EditAnywhere, BlueprintReadWrite)
	TSubclassOf<AActor> AICharacterClass;

	UPROPERTY(EditAnywhere, BlueprintReadWrite)
	TArray<AActor*> AllAICharacters;

	UPROPERTY(EditAnywhere, BlueprintReadWrite)
	TArray<FVector> CoverObjectLocations;

	TArray<FVector> CoverObjectLocationsForFlanker;

	UPROPERTY(EditAnywhere, BlueprintReadWrite)
	bool IsCollided;

	UPROPERTY(BlueprintReadWrite)
	TArray<FVector> LocationDataVector;

	int32 NumberOfStartsInitiated;

public:
	bool IsDead() const;
	virtual  void Tick(float DeltaTime) override;
	void FlankerStraightLineTrace(FVector& StraightLineTraceEnd);

private:
	UPROPERTY(EditAnywhere)
	float AcceptanceRadius = 200.0f;

	UPROPERTY(EditAnywhere)
	UBehaviorTree* EnemyBehaviorTree;

	UPROPERTY(EditAnywhere)
	bool IsReached = false;

	APawn* PlayerPawn;
```

```
	TArray<FString> LocationDataRead;

	TArray<FString> ShootIntervalData;

	FString PathtoFileforStarts;

	FString PathtoFileforShootInterval;

	FString PathtoFileforHiddenLocations;

	FString Loc;

	float HealthPercentage;

	int CountIntervalMode1;

	int CountIntervalMode2;

	int CountIntervalMode3;

	int flankTraceCount = 0;

	int count = 0;

	bool CanTrace;

	bool HasStoppedTracing;

	bool ObjTrace;

	bool ObjTraceForFlanker;

	bool CanFlankerTrace;

	void TracingByFlanker(FVector& DrivingVector);

};
```

```
//-----------------------------------------------------------------------------------------------------------------------------------------------------------

/// @file Enemy.cpp

/// @brief This code file contains the function definitions for all the features of a working AI character through the blueprint controllers
BP_Enemy, BP_EnemyAlt, and BP_EnemyFlanker.

/// @version 1.0

/// @date 24/07/21

/// @class AEnemy

//-----------------------------------------------------------------------------------------------------------------------------------------------------------


#include "Enemy.h"

#include "DrawDebugHelpers.h"

#include "PlayerCharacter.h"

#include "Kismet/GameplayStatics.h"

#include "Misc/DefaultValueHelper.h"

#include "BehaviorTree/BlackboardComponent.h"

#include "GameFramework/CharacterMovementComponent.h"



AEnemy::AEnemy()

{

 PrimaryActorTick.bCanEverTick = true;

 APlayerCharacter* ControlledCharacter = Cast<APlayerCharacter>(GetPawn());

 HealthPercentage = (ControlledCharacter->GetMaxHealth() * 30.0f) / 100.0f;

}


void AEnemy::BeginPlay()

{

 Super::BeginPlay();

 //Setting all blackboard keys used in the behaviour tree of the enemy AI.

 GetBlackboardComponent()->SetValueAsBool(TEXT("IsFlankerClose"), false);

 GetBlackboardComponent()->SetValueAsBool(TEXT("CanStartSLTrace"), false);

 GetBlackboardComponent()->SetValueAsInt(TEXT("MoveCoverIndex"),0);

 GetBlackboardComponent()->SetValueAsInt(TEXT("MoveCoverIndexForFlanker"),0);

 PlayerPawn= UGameplayStatics::GetPlayerPawn(GetWorld(),0);

 GetBlackboardComponent()->SetValueAsVector(TEXT("PlayerLocation"), PlayerPawn->GetActorLocation());

 GetBlackboardComponent()->SetValueAsObject(TEXT("Player"), PlayerPawn);


 if(EnemyBehaviorTree != nullptr)

 {

  RunBehaviorTree(EnemyBehaviorTree);

  GetBlackboardComponent()->SetValueAsVector(TEXT("PlayerLocation"), PlayerPawn->GetActorLocation());
```

```
 GetBlackboardComponent()->SetValueAsVector(TEXT("AIStartLocation"), GetPawn()->GetActorLocation());
 GetBlackboardComponent()->SetValueAsBool(TEXT("IsHealthLow"), false);
}

//Read the number of times the level has started from a file to start adaptive measures.
PathtoFileforStarts = FPaths::GameSourceDir();
PathtoFileforStarts.Append(TEXT("StartCount.txt"));
IPlatformFile& FileManager = FPlatformFileManager::Get().GetPlatformFile();
if(FileManager.FileExists(*PathtoFileforStarts))
{
 FString StartTimes;
 FFileHelper::LoadFileToString(StartTimes, *(PathtoFileforStarts));
 if(NumberOfStartsInitiated != FCString::Atoi(*StartTimes))
 {
  NumberOfStartsInitiated = FCString::Atoi(*StartTimes);
 }
}
//Read the values of shoot interval data from a file to match the speed of the player.
PathtoFileforShootInterval = FPaths::GameSourceDir();
PathtoFileforShootInterval.Append(TEXT("ShootInterval.txt"));
if(FileManager.FileExists(*PathtoFileforShootInterval))
{
 FFileHelper::LoadFileToStringArray(ShootIntervalData, *PathtoFileforShootInterval);
 for(int32 i = 0; i != ShootIntervalData.Num(); ++i)
 {
  float TempVal;
  TempVal = FCString::Atof(*ShootIntervalData[i]);
  if(TempVal < 1.0)
  {
   CountIntervalMode1++;
  }
  else if(TempVal < 2.0 && TempVal >= 1.0)
  {
   CountIntervalMode2++;
  }

else
  {
   CountIntervalMode3++;
  }
```

```
  }
 }
 //Read the location vectors from a file to search possible cover locations.
 PathtoFileforHiddenLocations = FPaths::GameSourceDir();
 PathtoFileforHiddenLocations.Append(TEXT("Locations.txt"));
 if(FileManager.FileExists(*PathtoFileforHiddenLocations))
 {
  FFileHelper::LoadFileToStringArray(LocationDataRead,*(FPaths::GameSourceDir() + TEXT("Locations.txt")));
  for(int32 i =0; i != LocationDataRead.Num(); ++i)
  {
   FVector Temp;
   Temp.InitFromString(LocationDataRead[i]);
   LocationDataVector.Add(Temp);
   Temp = Temp.ZeroVector;
  }
 }
 CanTrace = true;
 CanFlankerTrace = true;
 HasStoppedTracing = false;
}
//To check whether the enemy is killed or not to end the game level.
bool AEnemy::IsDead() const
{
 APlayerCharacter* ControlledCharacter = Cast<APlayerCharacter>(GetPawn());
 if(ControlledCharacter != nullptr)
 {
  return ControlledCharacter->isKilled();
 }
  return true;
}

void AEnemy::Tick(float DeltaTime)
{
 Super::Tick(DeltaTime);
 FTimerHandle StartTraceTimer;
 APlayerCharacter* ControlledCharacter = Cast<APlayerCharacter>(GetPawn());
 PlayerPawn= UGameplayStatics::GetPlayerPawn(GetWorld(),0);
 FVector Location = ControlledCharacter->GetActorLocation();
 TArray<FHitResult> OutHits;
 FCollisionObjectQueryParams CollisionTraceParams(ECollisionChannel::ECC_GameTraceChannel4);
```

```
if((ControlledCharacter== nullptr) && (PlayerPawn == nullptr))
{
 return;
}
float time = GetWorld()->GetTimeSeconds();
APlayerCharacter* Player = Cast<APlayerCharacter>(PlayerPawn);
if(Player != nullptr)
{
//Starting object trace for the enemy flanker type
 if(ControlledCharacter != nullptr && ControlledCharacter->ActorHasTag("Flanker"))
 {
 TracingByFlanker(FlankVector);
 float dot = FVector::DotProduct(ControlledCharacter->GetActorLocation().GetSafeNormal(),PlayerPawn->
GetActorLocation().GetSafeNormal());
 float angle = FMath::RadiansToDegrees(acosf(dot));
 float distance = FVector::Distance(PlayerPawn->GetActorLocation(), ControlledCharacter->GetActorLocation());

 //Checking the angle and distance to find whether the flanker is close to the player or not.
 if(angle < 100 || angle > 70)
 {
  if(distance < 500.0f)
  {
   GetBlackboardComponent()->SetValueAsBool(TEXT("IsFlankerClose"), true);
   CoverObjectLocationsForFlanker.Empty();
   GetBlackboardComponent()->SetValueAsInt(TEXT("MoveCoverIndexForFlanker"),0);
   CanFlankerTrace = false;
  }
 }
}

//Is Squad mode enables the BP_EnemyAI to act as a team when the level for scenario 2 is launched.
if(IsSquadMode)
{
 if(!ControlledCharacter->ActorHasTag("Flanker"))
 {
  if(Player->ActorHasTag("Player") && Player->Health > 0)
  {
   ObjTrace = GetWorld()->LineTraceMultiByObjectType(OutHits,Location,PlayerPawn->
GetActorLocation(),FCollisionObjectQueryParams::InitType::AllStaticObjects);
   float distance = FVector::Distance(PlayerPawn->GetActorLocation(),Location);
   if(distance < CloseLocationLimit)
```

```
{
 GetBlackboardComponent()->SetValueAsBool(TEXT("IsMovedCloseEnough"), true);
 CoverObjectLocations.Empty();
 GetBlackboardComponent()->SetValueAsInt(TEXT("MoveCoverIndex"),0);
 CanTrace = false;
}
if(time > 2.0f)
{
 //Shooting traces every 11 seconds to recalculate the cover locations between the player and the enemy.
 if((int)time % 11 == 0)
 {
  if((ControlledCharacter->GetVelocity() == FVector::ZeroVector) && (distance > 400.0f))
  {
   if(PlayerPawn->GetVelocity() == FVector::ZeroVector)
    {
     GetBlackboardComponent()->SetValueAsBool(TEXT("IsMovedCloseEnough"), false);
     CoverObjectLocations.Empty();
     GetBlackboardComponent()->SetValueAsInt(TEXT("MoveCoverIndex"),0);
     CanTrace = true;
    }
   }
  }
 }

if(CanTrace)
{
 if(ObjTrace)
 {
   CanTrace = false;
   DrawDebugLine(GetWorld(),Location,PlayerPawn->GetActorLocation(),FColor::Red,true,10);
   for(FHitResult HitR : OutHits)
   {
    if(HitR.GetActor()->ActorHasTag("CoverObj"))
    {
     //The impact point before a cover object is added as a cover location for the enemy to move.
     CoverObjectLocations.AddUnique(HitR.ImpactPoint);
     GetBlackboardComponent()->SetValueAsBool(TEXT("IsMovedCloseEnough"), false);
    }
   }
  }
```

```
    }
   }
  }
 }
}

if(ControlledCharacter->Health < HealthPercentage)
{
 GetBlackboardComponent()->SetValueAsBool(TEXT("IsHealthLow"), true);
}
//Setting the blackboard key value to check possible cover points.
if(NumberOfStartsInitiated > 3)
{
 GetBlackboardComponent()->SetValueAsBool(TEXT("CheckPoint"), true);
}
//Setting the blackboard key value to match the speed of shooting by the player.
if((CountIntervalMode1 > CountIntervalMode2) || (CountIntervalMode1 > CountIntervalMode3))
{
 GetBlackboardComponent()->SetValueAsFloat(TEXT("ShootSpeed"), 0.05f);
}
else if((CountIntervalMode2 > CountIntervalMode1) || (CountIntervalMode2 > CountIntervalMode3))
{
 GetBlackboardComponent()->SetValueAsFloat(TEXT("ShootSpeed"), 0.3f);
}
else
{
 GetBlackboardComponent()->SetValueAsFloat(TEXT("ShootSpeed"), 0.5f);
}
}

//This function is used only by the enemy type flanker to trace cover locations.
void AEnemy::TracingByFlanker(FVector& DrivingVector)
{
 const APlayerCharacter* ControlledCharacter = Cast<APlayerCharacter>(GetPawn());
 const APawn* PawnOfPlayer= UGameplayStatics::GetPlayerPawn(GetWorld(),0);
 const APlayerCharacter* Player = Cast<APlayerCharacter>(PlayerPawn);
 FVector Location = ControlledCharacter->GetActorLocation();
 FVector EndLocation = PawnOfPlayer->GetActorLocation() + DrivingVector;

 TArray<FHitResult> OutHitsForFlanker;
```

```
 if(Player->ActorHasTag("Player") && Player->Health > 0)
 {
  ObjTraceForFlanker = GetWorld()->
LineTraceMultiByObjectType(OutHitsForFlanker,Location,EndLocation,FCollisionObjectQueryParams::InitType::AllStaticObjects);
  if(CanFlankerTrace)
  {
   if(ObjTraceForFlanker)
   {
    CanFlankerTrace = false;
    DrawDebugLine(GetWorld(),Location,EndLocation,FColor::Red,true,10);
    for(FHitResult HitR : OutHitsForFlanker)
    {
     if(HitR.GetActor()->ActorHasTag("CoverObj"))
     {
      CoverObjectLocationsForFlanker.AddUnique(HitR.ImpactPoint);
      GetBlackboardComponent()->SetValueAsBool(TEXT("IsFlankerClose"), false);
     }
    }
   }
  }
 }
}

//This function is used by the enemy flanker type to move in parallel to the player's location
void AEnemy::FlankerStraightLineTrace(FVector& StraightLineTraceEnd)
{
 GetBlackboardComponent()->SetValueAsBool(TEXT("IsFlankerClose"), true);
 CoverObjectLocationsForFlanker.Empty();
 GetBlackboardComponent()->SetValueAsInt(TEXT("MoveCoverIndexForFlanker"),0);
 TracingByFlanker(StraightLineTraceEnd);
}

//At the end of play, the start time is incremented and written to a file.
AEnemy::~AEnemy()
{
 ++NumberOfStartsInitiated;
 FFileHelper::SaveStringToFile(*FString::FromInt(NumberOfStartsInitiated),*(FPaths::GameSourceDir() + TEXT("StartCount.txt")));
}
```

```
//-----------------------------------------------------------------------------------------------------------------------------------------------
/// @file PlayerCharacter.h
/// @brief This header file contains the function declarations and variables for all the features of a working player character.
/// @version 1.0
/// @date 15/07/21
/// @class APlayerCharacter
//-----------------------------------------------------------------------------------------------------------------------------------------------

#pragma once
#include "CoreMinimal.h"
#include "GameFramework/Character.h"
#include "PlayerCharacter.generated.h"
class AWeapon;

UCLASS()
class EXTRACTION_API APlayerCharacter : public ACharacter
{
	GENERATED_BODY()
public:
	// Sets default values for this character's properties
	APlayerCharacter();
protected:
	// Called when the game starts or when spawned
	virtual void BeginPlay() override;
public:
	UPROPERTY(EditAnywhere, BlueprintReadWrite)
	int SprintSpeed = 2;

	UPROPERTY(EditAnywhere, BlueprintReadWrite)
	int CrouchValue = 0;

	UPROPERTY(EditAnywhere, BlueprintReadWrite)
	bool isMoving;

	UPROPERTY(BlueprintReadWrite, EditAnywhere)
	bool CollisionDetection = false;

	UPROPERTY(EditAnywhere, BlueprintReadWrite)
	bool isCrouched;
```

```
UPROPERTY(BlueprintReadWrite, EditAnywhere)
float  JumpSwitch = 0.0f;

UPROPERTY(BlueprintReadWrite, EditAnywhere)
float  JumpTransition = 0.0f;

UFUNCTION(BlueprintPure)
float GetHealthPercentage();

UFUNCTION(BlueprintPure)
bool isKilled() const;

UPROPERTY(VisibleAnywhere)
  float Health;

// Called every frame
virtual void Tick(float DeltaTime) override;

// Called to bind functionality to input
virtual void SetupPlayerInputComponent(class UInputComponent* PlayerInputComponent) override;

virtual float TakeDamage(float DamageAmount, struct FDamageEvent const& DamageEvent, class AController* EventInstigator,
AActor* DamageCauser) override;

void Shoot();

float GetMaxHealth();

private:

UPROPERTY(EditDefaultsOnly)
TSubclassOf<AWeapon> WeaponClass;

UPROPERTY()
AWeapon* Gun;

UPROPERTY(EditAnywhere)
float MaxHealth = 100.0f;

void MoveForward(float AxisValue);
```

```
        void MoveSideways(float AxisValue);

        void Crouch();

        float DefaultWalkSpeed;

        float JumpTransitionIncrementor = 0.1f;

        FVector DefaultVelocity;

};



//-----------------------------------------------------------------------------------------------------------------------------------------------------------
/// @file PlayerCharacter.cpp
/// @brief This code file contains the function definitions for all the features of a working player character.
/// @version 1.0
/// @date 15/07/21
/// @class APlayerCharacter
//-----------------------------------------------------------------------------------------------------------------------------------------------------------

#include "PlayerCharacter.h"
#include "GameFramework/CharacterMovementComponent.h"
#include "Math/Vector.h"
#include "Engine/Classes/Kismet/GameplayStatics.h"
#include "Weapon.h"
#include "Components/CapsuleComponent.h"
#include "ExtractionGameModeBase.h"



APlayerCharacter::APlayerCharacter()
{
 PrimaryActorTick.bCanEverTick = true;
 isCrouched = true;
 DefaultWalkSpeed = GetCharacterMovement()->MaxWalkSpeed;
}
```

```
void APlayerCharacter::BeginPlay()
{
 Super::BeginPlay();
 Health = MaxHealth;
 DefaultVelocity = GetVelocity();

 //Attaching the weapon class component to the player to register firing, damage and health events.
 Gun = GetWorld()->SpawnActor<AWeapon>(WeaponClass);
 GetMesh()->HideBoneByName(TEXT("weapon_r"), EPhysBodyOp::PBO_None);
 Gun->AttachToComponent(GetMesh(),FAttachmentTransformRules::KeepRelativeTransform,TEXT("weapon_socket"));
 Gun->SetOwner(this);
}

// Called every frame
void APlayerCharacter::Tick(float DeltaTime)
{
 Super::Tick(DeltaTime);
 //To adjust moving animation based on movement
 if(GetWorld()->GetFirstPlayerController()->GetInputKeyTimeDown(FKey("W")) >= 0.1f
 || GetWorld()->GetFirstPlayerController()->GetInputKeyTimeDown(FKey("S")) >= 0.1f
 || GetWorld()->GetFirstPlayerController()->GetInputKeyTimeDown(FKey("A")) >= 0.1f
 || GetWorld()->GetFirstPlayerController()->GetInputKeyTimeDown(FKey("D")) >= 0.1f)
 {
  isMoving = true;
 }
 else
 {
  isMoving = false;
 }
 //For transitioning jump states
 if(GetCharacterMovement()->IsFalling())
 {
  JumpSwitch = 1.0f;
  JumpTransition += JumpTransitionIncrementor;
 }
 else
 {
  JumpSwitch = 0.0f;
  JumpTransition = 0.0f;
 }
```

```
}
// Called to bind functionality to input
void APlayerCharacter::SetupPlayerInputComponent(UInputComponent* PlayerInputComponent)
{
 Super::SetupPlayerInputComponent(PlayerInputComponent);
 PlayerInputComponent->BindAxis(TEXT("ForwardMovement"), this, &APlayerCharacter::MoveForward);
 PlayerInputComponent->BindAxis(TEXT("LateralMovement"), this, &APlayerCharacter::MoveSideways);
 PlayerInputComponent->BindAxis(TEXT("VerticalLook"), this, &APawn::AddControllerPitchInput);
 PlayerInputComponent->BindAxis(TEXT("HorizontalLook"), this, &APawn::AddControllerYawInput);
 PlayerInputComponent->BindAction(TEXT("Jump"), EInputEvent::IE_Pressed, this ,&ACharacter::Jump);
 PlayerInputComponent->BindAction(TEXT("Crouch"), EInputEvent::IE_Pressed, this ,&APlayerCharacter::Crouch);
 PlayerInputComponent->BindAction(TEXT("Shoot"), EInputEvent::IE_Pressed, this, &APlayerCharacter::Shoot);
}

//Function to receive damage from another actor inside the scene
float APlayerCharacter::TakeDamage(float DamageAmount, FDamageEvent const& DamageEvent, AController* EventInstigator, AActor*
                                   DamageCauser)
{
 float DamageValue = Super::TakeDamage(DamageAmount, DamageEvent, EventInstigator, DamageCauser);
 DamageValue = FMath::Min(Health, DamageValue);
 Health -= DamageValue;
 if(isKilled())
 {
  //When killed, the controller is detached and depending on the ending gameplay condition, further actions are evoked.
  AExtractionGameModeBase* BaseGameMode =GetWorld()->GetAuthGameMode<AExtractionGameModeBase>();
  if(BaseGameMode != nullptr)
  {
   BaseGameMode->PawnKilled(this);
  }
  DetachFromControllerPendingDestroy();
  GetCapsuleComponent()->SetCollisionEnabled(ECollisionEnabled::NoCollision);
 }
 return DamageValue;
}

void APlayerCharacter::Shoot()
{
 if(!isKilled())
 {
  Gun->PullTrigger();
 }
```

```
}

float APlayerCharacter::GetMaxHealth()
{
 return 100.0f;
}

float APlayerCharacter::GetHealthPercentage()
{
 return Health / MaxHealth;
}

bool APlayerCharacter::isKilled() const
{
 return Health <= 0;
}

void APlayerCharacter::MoveForward(float AxisValue)
{
 AddMovementInput(GetActorForwardVector() * AxisValue);
}

void APlayerCharacter::MoveSideways(float AxisValue)
{
 AddMovementInput(GetActorRightVector() * AxisValue);
}

void APlayerCharacter::Crouch()
{
 if(isCrouched)
 {
  ACharacter::Crouch();
  isCrouched = false;
 }
 else
 {
  ACharacter::UnCrouch();
  isCrouched = true;
 }
}
```

```
//-----------------------------------------------------------------------------------------------------------------------------------------------------------

/// @file Weapon.h

/// @brief This header file contains the function definitions and variables for a weapon component to be attached to the player or an enemy character.

/// @version 1.0

/// @date 15/07/21

/// @class AWeapon

//-----------------------------------------------------------------------------------------------------------------------------------------------------------


#pragma once


#include "CoreMinimal.h"

//#include "NiagaraComponent.h"

#include "GameFramework/Actor.h"

#include "Weapon.generated.h"


UCLASS()

class EXTRACTION_API AWeapon : public AActor

{

	GENERATED_BODY()

public:

	AWeapon();

	~AWeapon();

	void PullTrigger();


protected:

	virtual void BeginPlay() override;


public:

	virtual void Tick(float DeltaTime) override;


private:

	UPROPERTY(VisibleAnywhere)

	USceneComponent* SceneRoot;


	UPROPERTY(VisibleAnywhere)

	UStaticMeshComponent* WeaponMesh;


	UPROPERTY(EditAnywhere)

	USoundBase* WeaponMuzzleSound;
```

```
	UPROPERTY(EditAnywhere)
	USoundBase* BulletImpactSound;

	UPROPERTY(EditAnywhere)
	float LineTraceMaxRange = 100.0f;

	UPROPERTY(EditAnywhere)
	UParticleSystem* MuzzleFlash;

	UPROPERTY(EditAnywhere)
	UParticleSystem* ImpactEffect;

	UPROPERTY(EditAnywhere)
	float HitDamage = 10.0f;

	float TimePrevious = 0.0f;

	TArray<FString> ShootIntervalData;

	bool CanWeaponTrace(FHitResult &Hit, FVector &ShotDirection) const;

	AController* GetOwnerController() const;
};

//------------------------------------------------------------------------------------------------------------------------------------------------------
/// @file Weapon.cpp
/// @brief This code file contains the function definitions for a weapon component to be attached to the player or an enemy character.
/// @version 1.0
/// @date 15/07/21
/// @class AWeapon
//------------------------------------------------------------------------------------------------------------------------------------------------------


#include "Weapon.h"
#include "Components/StaticMeshComponent.h"
#include "Kismet/GameplayStatics.h"
#include "BehaviorTree/BlackboardComponent.h"
#include "Perception/AISense_Hearing.h"
```

```
AWeapon::AWeapon()
{
 PrimaryActorTick.bCanEverTick = true;

 SceneRoot = CreateDefaultSubobject<USceneComponent>(TEXT("Root Scene Component"));
 SetRootComponent(SceneRoot);

 //Adding the weapon mesh
 WeaponMesh = CreateDefaultSubobject<UStaticMeshComponent>(TEXT("Weapon Class"));
 WeaponMesh->SetupAttachment(SceneRoot);
 SetActorEnableCollision(false);
}

void AWeapon::BeginPlay()
{
 Super::BeginPlay();
 TimePrevious = FDateTime::UtcNow().GetSecond();
}

//This function is to trace the bullet path and can be used to spawn particle effects and sounds.
bool AWeapon::CanWeaponTrace(FHitResult& LineHit, FVector& ShotDirection) const
{

 AController* PawnController = GetOwnerController();
 if(PawnController == nullptr)
 {
  return false;
 }
 FVector Location;
 FRotator Rotation;
 PawnController->GetPlayerViewPoint(Location,Rotation);
 ShotDirection = -Rotation.Vector();
 FVector EndVector = Location + Rotation.Vector() * LineTraceMaxRange;
 FCollisionQueryParams CollisionParams;
 CollisionParams.AddIgnoredActor(this);
 CollisionParams.AddIgnoredActor(GetOwner());
 return  GetWorld()->LineTraceSingleByChannel(LineHit,Location,EndVector,ECollisionChannel::ECC_GameTraceChannel1, CollisionParams);
}
```

```
AController* AWeapon::GetOwnerController() const
{
 APawn* WorldPawn = Cast<APawn>(GetOwner());
 if(WorldPawn == nullptr)
 {
  return nullptr;
 }
 return  WorldPawn->GetController();
}

void AWeapon::PullTrigger()
{
 if(GetOwner()->ActorHasTag("Player"))
 {
  //To calculate the shoot intervals when the player presses the key to shoot.
  float TimeNow = FDateTime::UtcNow().GetSecond();
  float TimeDiff = TimeNow - TimePrevious;
  TimePrevious = TimeNow;
  ShootIntervalData.Add(FString::SanitizeFloat(TimeDiff));
 }
 UGameplayStatics::SpawnEmitterAttached(MuzzleFlash, WeaponMesh, TEXT("Muzzle"));
 UGameplayStatics::SpawnSoundAttached(WeaponMuzzleSound,WeaponMesh,TEXT("Muzzle"));

 FHitResult Hit;
 FVector ShotDirection;
 if(CanWeaponTrace(Hit, ShotDirection))
 {
  AActor* HitActor = Hit.GetActor();
  UGameplayStatics::SpawnEmitterAtLocation(GetWorld(), ImpactEffect, Hit.Location, ShotDirection.Rotation());
  UGameplayStatics::PlaySoundAtLocation(GetWorld(), BulletImpactSound, Hit.Location);
  UAISense_Hearing::ReportNoiseEvent(GetWorld(), Hit.Location,1,GetOwner(),2000.0f, FName("Noise"));

 //Damaging based on firing by different characters.
  if(HitActor != nullptr)
  {
   FPointDamageEvent DamageEvent(HitDamage, Hit, ShotDirection, nullptr);
   AController* PawnController = GetOwnerController();
   if((HitActor->ActorHasTag("Player") && (AActor*)this->GetOwner()->ActorHasTag("Enemy")) ||
   (HitActor->ActorHasTag("Enemy") && (AActor*)this->GetOwner()->ActorHasTag("Player")) ||
   (HitActor->ActorHasTag("EnemyAlt") && (AActor*)this->GetOwner()->ActorHasTag("Player")) ||
```

```
   (HitActor->ActorHasTag("Player") && (AActor*)this->GetOwner()->ActorHasTag("EnemyAlt")) ||
   (HitActor->ActorHasTag("Flanker") && (AActor*)this->GetOwner()->ActorHasTag("Player")) ||
   (HitActor->ActorHasTag("Player") && (AActor*)this->GetOwner()->ActorHasTag("Flanker")))
   {
    HitActor->TakeDamage(HitDamage, DamageEvent, PawnController, (AActor*)this);
   }
  }
 }
}
void AWeapon::Tick(float DeltaTime)
{
 Super::Tick(DeltaTime);
}

//Calculated Shoot intervals are written to a file to be read in the Enemy controller class.
AWeapon::~AWeapon()
{
 ShootIntervalData.Sort();
 FFileHelper::SaveStringArrayToFile(ShootIntervalData,*(FPaths::GameSourceDir() +
TEXT("ShootInterval.txt")),FFileHelper::EEncodingOptions::AutoDetect, &IFileManager::Get(), EFileWrite::FILEWRITE_Append);
}

//-----------------------------------------------------------------------------------------------------------------------------------------------------------
/// @file BTService_PlayerLocAfterSpotting.h
/// @brief This header file for a custom behaviour tree service contains the function and variable declarations needed to update the last known
player location every tick value.
/// @version 1.0
/// @date 10/08/21
/// @class UBTService_PlayerLocAfterSpotting
//-----------------------------------------------------------------------------------------------------------------------------------------------------------

#pragma once
#include "CoreMinimal.h"
#include "BehaviorTree/Services/BTService_BlackboardBase.h"
#include "BTService_PlayerLocAfterSpotting.generated.h"

UCLASS()
class EXTRACTION_API UBTService_PlayerLocAfterSpotting : public UBTService_BlackboardBase
{
        GENERATED_BODY()
```

```
public:
	UBTService_PlayerLocAfterSpotting();
	~UBTService_PlayerLocAfterSpotting();

protected:
	virtual void TickNode(UBehaviorTreeComponent& OwnerComp, uint8* NodeMemory, float DeltaSeconds) override;

private:
	TArray<FString> PlayerLocationData;

};



//------------------------------------------------------------------------------------------------------------------------------------------------------
/// @file BTService_PlayerLocAfterSpotting.cpp
/// @brief This code file for a custom behaviour tree service contains the function definitions needed to update the last known player location every tick value.
/// @version 1.0
/// @date 10/08/21
/// @class UBTService_PlayerLocAfterSpotting
//------------------------------------------------------------------------------------------------------------------------------------------------------

#include "BTService_PlayerLocAfterSpotting.h"
#include "BehaviorTree/BlackboardComponent.h"
#include "Kismet/GameplayStatics.h"
#include "GameFramework/Pawn.h"
#include "AIController.h"
#include "Enemy.h"
#include "Perception/AIPerceptionComponent.h"

UBTService_PlayerLocAfterSpotting::UBTService_PlayerLocAfterSpotting()
{
 NodeName = TEXT("Update Player Location After Spotting");
}

void UBTService_PlayerLocAfterSpotting::TickNode(UBehaviorTreeComponent& OwnerComp, uint8* NodeMemory,
                                                 float DeltaSeconds)
{
 Super::TickNode(OwnerComp, NodeMemory, DeltaSeconds);
```

```
APawn* PlayerPawn= UGameplayStatics::GetPlayerPawn(GetWorld(),0);
FActorPerceptionBlueprintInfo ActorInfo;
ActorInfo.Target = PlayerPawn;
ActorInfo.bIsHostile = true;
AEnemy* Enemy = Cast<AEnemy>(OwnerComp.GetAIOwner());

//Adding locations of the bushes in the scene to PlayerLocationData array to write to a file.
if(Enemy->NumberOfStartsInitiated <= 0)
{
 if(!Enemy->BushLocations.IsEmpty())
 {
  for(FVector BushLocation : Enemy->BushLocations)
  {
   PlayerLocationData.AddUnique(BushLocation.ToString());
  }
 }
}
if(PlayerPawn == nullptr)
{
 return;
}
if(OwnerComp.GetAIOwner() == nullptr)
{
 return;
}
//Adding last seen player locations to the PlayerLocationData array to write to a file later.
if(OwnerComp.GetAIOwner()->GetAIPerceptionComponent()->GetActorsPerception(PlayerPawn, ActorInfo))
{
 OwnerComp.GetBlackboardComponent()->SetValueAsObject(GetSelectedBlackboardKey(), PlayerPawn);
 PlayerLocationData.AddUnique(PlayerPawn->GetActorLocation().ToString());
}
else
{
 OwnerComp.GetBlackboardComponent()->ClearValue(GetSelectedBlackboardKey());
}
}
```

```
//When the gameplay ends, the values stored in the PlayerLocationData array is written to a file to be read and utilized later.

UBTService_PlayerLocAfterSpotting::~UBTService_PlayerLocAfterSpotting()

{

FFileHelper::SaveStringArrayToFile(PlayerLocationData,*(FPaths::GameSourceDir() +
TEXT("Locations.txt")),FFileHelper::EEncodingOptions::AutoDetect, &IFileManager::Get(), EFileWrite::FILEWRITE_Append);

}
```

REFERENCES

Bigdata, 2021. History of AI Use in Video Game Design. *Big Data Analytics News* [online]. 24 March 2021. Available from: https://bigdataanalyticsnews.com/history-of-artificial-intelligence-in-video-games/ [Acce

Epic Games, 2021. *Unreal Engine.* Version five [computer program]. North Carolina: Epic Games.

Matuschek, M., 2020. Using Adaptive AI to Improve the Gaming Experience. *Mouser Electronics* [online]. 21 December 2020. Available from: https://www.mouser.com/blog/using-adaptive-ai-improve-gaming-experience [Accessed 25 July 2021].

Millington, I., and Funge, J., 2009. *Artificial Intelligence for Games*. Second Edition. Florida: CRC Press.

Ram, A., Ontanon, S. and Mehta, M., 2007. Artificial Intelligence for Adaptive Computer Games [online]. *In: Proceedings of the Twentieth International Florida Artificial Intelligence Research Society Conference*, Key West, 7-9 May 2007. Florida: AAAI Press. 22-29. Available from: https://www.aaai.org/Papers/FLAIRS/2007/Flairs07-007.pdf [Accessed 22 July 2021].

Rockstar Games, 2013. *Grand Theft Auto V.* [computer program]. New York: Rockstar Games.

Skinner, G., Walmsley, T., 2019. Artificial Intelligence and Deep Learning in Video Games A Brief Review [online]. *In: IEEE 4th International Conference on Computer and Communication Systems*, Singapore, 23-25 February 2019. Singapore: IEEE. 404-408. Available from: https://doi.org/10.1109/CCOMS.2019.8821783 [Accessed 23 July 2021].

Spronck, P., Ponsen, M., Sprinkhuizen-Kuyper, I. and Postma, E., 2006. Adaptive game AI with dynamic scripting. Machine Learning [online], 9 March 2006. Available from: https://doi.org/10.1007/s10994-006-6205-6 [Accessed 23 July 2021].

Taljonick, R., 2014. Shadow of Mordor's Nemesis system is amazing--here's how it works. *gamesradar+* [online]. 29 August 2014. Available from: https://www.gamesradar.com/uk/shadow-mordor-nemesis-system-amazing-how-works/ [Accessed 25 July 2021].

Thompson, T., 2014. Why AI Researchers Love Playing Pac-Man. *Tommy Thompson* [online]. 10 Feb 2014. Available from: https://medium.com/@t2thompson/ailovespacman-9ffdd21b01ff [Accessed 26 July 2021].

Ubisoft, 2020. *Assassin's Creed Valhalla*. [computer program]. Montreal: Ubisoft.

Warner Bros. Interactive Entertainment, 2014. *Middle-earth: Shadow of Mordor* [computer program]. California: Warner Bros. Interactive Entertainment.

Xia, B., Ye, X., Abuassba, A.O.M., 2020. Recent Research on AI in Games [online]. *In: 2020 International Wireless Communications and Mobile Computing (IWCMC)*, Cyprus, 15-19 June 2020. Limassol: IEEE. 505-510. Available from: https://doi.org/10.1109/IWCMC48107.2020.9148327 [Accessed 28 July 2021].