# GIV011: Artificial Intelligence (AI) methods for Games

Dissertation Proposal

Supervised By: Dr. Aaron Bostrom

Yuvraj Singh Jadon, 100349186

August 25, 2022

## Abstract

Artificial Intelligence in games is a promising field for the future of games. The use of AI in games can help to create more realistic and engaging experiences for players. AI is also used to improve the experience of the player and their game experience.

Modern computer games widely use AI techniques to typically make NPCs ( Non-player characters) behave'intelligently' and aiming to outsmart the player(s). Three typical methods used for game AI are finite state machines (FSMs), goal-oriented action planning (GOAP) and behavior trees. Each of these methods has advantages and disadvantages.

In this project a set of test games are created that aim to illustrate the pros and cons of each of these methods. The challenges in this project are to thoroughly understand the strengths and weaknesses of the three AI methods and to identify and implement specific game AI to illustrate these strengths and weaknesses by comparison. These games are implemented using game engines Unity and Unreal with c# and c++ as scripting languages.

# Contents

# List of Figures

# Chapter 1

# Introduction

Artificial intelligence (AI) is a field of computer science that involves the creation and use of intelligent machines. AI has many applications in the field of robotics, but also in medicine, finance, transportation, and more. AI has been around since the 1940s, but it's only recently that it has truly become a viable option for businesses.

Although artificial intelligence has been around for decades, there are still some people who believe that it is impossible to create an intelligent machine without human intervention. This is because artificial intelligence can be defined as any system that uses algorithms to solve problems on its own. However, there are many other definitions of artificial intelligence that involve human input such as: "a machine that acts like a human," "a machine acting like a human," or "a computer program capable of learning."

Artificial intelligence in games is an important topic that has been studied for decades. It is often used to create artificial opponents for games, or to help the player in some way. This proposal will discuss how artificial intelligence can be used in games, and focus on their strengths and weaknesses.

The gaming business has developed from a small sector to a multi-billion euro industry in the previous several decades. We've witnessed a rocket-like surge in graphic and audio quality as the industry has evolved, but the field of artificial intelligence (AI) in games hasn't progressed at the same rate. Even while there has been considerable progress in the recent decade, many basic issues remain (Warpefelt et al., 2013). Many of the difficulties with NPC behaviour, as outlined by Johansson (2013) and Warpefelt (2013), originate from a lack of plausibility, particularly in their social elements. In order to be viewed as believable, NPC conduct must match the player's expectations (Loyall, 1997). However, for this alignment to work, the game must persuade the player of what is genuine in the game world, as well as create realistic landscapes, worlds, and circumstances in which the player may immerse themselves (Warpefelt, 2016).

## 1.1 Statement of the project

The project focuses on implementing various Artificial Intelligence algorithms in game engines for NPC's ability to make it real world and interactive for the gamer to enjoy, an immersive world which resembles real scenarios. The algorithm's implemented depends on the complexity and genre of game developed. This application will highlight the strengths and weaknesses of the algorithm and provide a clear understanding of the AI algorithms.

## 1.2 Background and motivation

In 2015, DeepMind published a paper describing the use of Artificial Neural Network and Machine Learning to train an npc which can play a lot of 2-D video games from 1970s and 1980s to a proficient level without any human intervention which was revolutionary(Gibney et al., 2015) which showed a great deal of application in many varied real life applications. Generally, the game development requires skills in programming and visual scripting languages such as c# and Blueprints. As a game enthusiast and a long time gamer, i was always baffled by the logic behind the functioning of non-player characters and their interaction with the game environment. The motivation to choose artificial intelligence in games is that it allows the game designer to create a more realistic experience. The artificial intelligence can be used to make the games more interactive, and it can also be used to make them more challenging and giving players the sense that they are interacting with a real person rather than a computer generated character.

## 1.3 Aim and Objectives

### 1.3.1 Aim

The aim of this project is to present a detailed idea of the techniques and methodologies used in developing NPC's and their implementation in Unity and Unreal Engine 5 with languages C# and Blueprints and present a comparison between Finite State Machines (FSMs), Goal Oriented Action Planning (GOAP) and Behaviour Trees as well as the strengths and weaknesses

### 1.3.2 Objective

There are two main objectives:

- To thoroughly understand FSM, GOAP and Behavior Tree algorithms and their strengths and weaknesses.
- To identify and implement these algorithms in a set of games for better understanding in comparison to their strengths and weaknesses.

## 1.4 Project Overview

This project involve a few key stages as shown in chapter 4. Firstly, a thorough understanding of the strengths and weaknesses of each AI method is achieved by the resources available. After this, The FSM, GOAP and Behavior Trees are implemented on NPC's in different games and enhancing their functionalities in unity and unreal engine with the languages C# and Blueprints and deriving a conclusion on its strengths and weaknesses.

The second stage is to create a set of experiments to verify some of our strengths and weaknesses of these AI algorithms

## 1.5 Designing Tools

The designing tool which are going to be used:

- Unreal engine and Unity Engine as the game design engine.
- C++ for Unreal engine and C# for Unity engine.
- Blueprints as visual scripting Language.

# Chapter 2

# Literature Review

Artificial intelligence, or AI, is a science that aims to make machines smarter and more self-reliant than humans. It is used not just in the creation of physical machines like robots (the Curiosity rover, Sophia, Alexa, and so on), but also in games such as RTS (Real Time Strategy) and FPS (First Person Shooter) games, allowing them to respond to human input and change the scenario.

Alan Turing, a British mathematician, wrote in one of the first articles in this topic in 1950, addressing the question of machine intelligence explicitly in connection to the modern digital computer. Computing Machinery and Intelligence is still relevant in its examination of the reasons for and against the creation of an intelligent computing machine. He proposed The Turing Test(Livingstone, 2006), which compares an apparently intelligent machine's performance to that of a human being, arguably the best benchmark for intelligent conduct. The Turing test usually takes the form where an interrogator in one room uses a computer terminal to play a game of question and answer with two subjects who are located in another room. One of the subjects is human while the other is a machine; the task of the interrogator is to determine which is which. If the interrogator is unable to tell, then the machine must be considered intelligent.(Livingstone, 2006).

AI can be applied to most aspects of game development and design including automated content creation, procedural animation, adaptive lighting, intelligent camera control, and perhaps the most obvious example, controlling the Non-Player Character s(NPCs)(Lucas, 2009). In the early days, it was mostly used to add difficulty and challenges to a game. Games that featured AI-controlled characters were typically difficult to beat, but they added a layer of strategy and skill that many players enjoyed.

As time went on, developers began to realize that AI could be used to make games more fun by adding elements of storytelling. Games such as Half-Life 2 are known for their extensive use of scripted scenes and dialogue with characters who react according to their own personalities(Yildirim and Stene, 2010). This type of AI has become more prevalent over the years—it's now commonplace for games to have multiple AIs interacting with each other in order to create new experiences for players.

The evolution of artificial intelligence in games has also had an impact on human-to-human interactions within these games. As we've seen with popular titles like Portal 2, developers can create AIs that make players feel like they're interacting with real people instead of just computer programs or animated avatars.

The first use of AI in games was in real-time strategy games such as Command & Conquer and Starcraft II where units have personalities and can be trained to perform specific tasks.

In these types of games, players are able to control their armies by issuing commands that the unit will then execute based on its personality traits.

In another example, artificial intelligence has been used in adventure games such as Monkey Island and The Sims (Laird, 2002) where characters must make decisions based on their own personality traits or how they were trained by their owners. This allows players to interact with characters that display emotions like anger or happiness which can affect their gameplay experience significantly.

## 2.1 AI history of gaming

The earliest roots of AI in games can be found in 1951 mathematical strategy game called NIM, in which two players take turns to remove objects from piles.It was able to beat human players regularly." At the University of Manchester, a game of checkers and a game of chess were written using the AI 'Ferranti Mark 1 machine' the same year(Bigdata, 2021). The use of AI in game design emerged in the 1970s, when Space Invaders, a game famous for its increasingly difficult levels and unique patterns of movement, incorporated the use of AI.Pac-Man introduced AI patterns into its maze in 1980. Fighting games soon started using AI, such as 1984's Karate Champ, and in 1988, the action role-playing game First Queen introduced characters controlled by the computer's AI (News, 2021) . Tools like finite state machines were used in new video game genres that emerged in the 1990's. For example, AI was used in real-time strategy games for path finding, making real-time decisions, and economic planning.

## 2.2 Human Brain Project

The Human Brain project is an attempt to replicate the human brain using artificial intelligence. The project was started in 2015, and it is being led by Dr. Henry Markram at the Swiss Federal Institute of Technology in Lausanne, Switzerland.

The Human Brain Project aims to create a virtual replica of the human brain that can be used for research purposes. The project is being funded by the Canadian government, and it has so far received $ 1 billion from them. It will take about 10 years for the project to complete its goal (Salles et al., 2019), which is to create an artificial brain that functions like a real one does .

The first step in this process was creating a model of how neurons work within the human brain. This model was created using data from rodents and primates, as well as data from other animals' brains as well as humans' brains (including those who have died).

The second step involves creating a simulation of how synapses work within neurons. The goal here is to simulate how information flows through these synapses and how they function together within particular parts of the brain; this includes what happens when neurons are damaged or when they aren't working properly due to illness or injury.

These types of projects really works in the direction of artificial super intelligence and further evolves every sector they are incorporated.

## 2.3 Previous Related Work

There has been several papers written about the use of different techniques used for developing NPC's. One of the most notable work in this field with respect to FSM methodology is by (Lee, 2014) of explaining the games in this field while for GOAP and behaviour trees methodology the paper (Long, 2007) and (Lee, 2016) gives a detailed view of their implementation in a game. In the field of GOAP algorithms a major portion of the work is published by the developer of FEAR game. According to Orkin (2004), the GOAP architecture provides a good framework for the creation of human like intelligence in his game series FEAR.

## 2.4 Deterministic Versus Nondeterministic AI

Game AI techniques generally come in two types: deterministic and nondeterministic.

### 2.4.1 Deterministic

Deterministic performance or behaviour is predetermined and predictable. There isn't any ambiguity. A basic chasing algorithm is an example of deterministic behaviour.(Bourg and Seemann, 2004) It is a new way of making games that uses an algorithm to determine what the next move will be. This allows the game to be much more unpredictable and engaging, as it's not just following a set path, but reacting based on what has happened in the past. A FSM, Goap or Behavior tree is used in this type of implementation.

### 2.4.2 Nondeterministic

The antithesis of deterministic behaviour is nondeterministic behaviour. There is a level of uncertainty in behaviour, and it is quite unpredictable. it is a programming technique used in video games to generate several outcomes of a game based on randomness. A NPC learning to adapt to a player's response techniques is an example of nondeterministic behaviour(Bourg and Seemann, 2004). A neural network, a Bayesian approach, or a genetic algorithm could be used in this type of learning.

# Chapter 3

# Definitions

## 3.1 Game

A computer game, also known as a video game or a computer-controlled simulation of play, is a form of entertainment perhaps most familiar from video games. Computer games usually involve user input in the form of data generated by a player by using a controller device or alternatively by responding to instructions in response to control (directional pad, keyboard and mouse) inputs (Stenros, 2017).

For more than 50 years, computer games have been an important part of our culture. As technology has developed, they have become more complex and immersive. They can be played on computers, mobile devices and game consoles.

## 3.2 Game Engine

A game engine is a software that provides the basic functions of a game, like 2D and 3D graphics, physics simulation, sound and music playing, user input handling, network communication and so on. This engine is typically used in the development of video games For decades games were created in a very non-technical way, with programmers creating all components of the game themselves. This proved to be highly ineffective and time consuming, so dedicated game engine software (Unity, Unreal Engine, CryEngine etc) was developed and subsequently used by most studios/developers today. Game engines can be used to power single-player or multi-player games. A game engine may also be used to create non-interactive media, such as movies and advertising(Andrade, 2015). Game engines are often designed with common functionality, which means they can be used by multiple developers in the same way. The technology has been used in every major gaming platform since its invention, including PC, console, mobile devices and Internet browsers. The first game engine was developed by Ralph H. Baer at Magnavox in 1972. He called it the "Brown Box". Some of these early engines were designed only to display graphics, while others could play sounds and music (Andrade, 2015). Game engines were especially important for the gaming industry during this time period because they allowed developers to easily reuse code for different games; this helped reduce expenses for each title's development cycle.

### 3.2.1 Unreal Engine

The Unreal Engine is a game engine developed by Epic Games, a video game developer based in North Carolina. The engine was created in 1998 and it has since been used to develop games for over 20 years.

The Unreal Engine is a general-purpose engine that has been used for many types of games including first-person shooters, role-playing games, action adventure games and racing games.It has been used in the development of many popular games, including BioShock, Mass Effect, and Gears of War (Sanders, 2016). The engine has also been used in movies such as the film Jurassic Park III: Park Builder, where it provided the visual effects for dinosaurs. The engine supports both DirectX 11 and DirectX 12 to allow developers to use additional graphical features.

The Unreal Engine allows users to create their own content using the SDK (Software Development Kit), which is available online or on disc. This can be used in conjunction with the editor's Blueprint visual scripting language or C++ source code. The editor's Visual Scripting Language allows developers to create high quality game levels without needing programming knowledge (Sanders, 2016). Developers can also import custom assets into their games using the Content Browser system, which allows them to add new assets quickly without having to work directly with programmers or artists.

### 3.2.2 Unity

Unity is a powerful, open-source game development engine. It is easy to use, with a robust API and a large selection of free and paid packages that can be used to build almost any type of game. Unity is also highly customizable, so developers can create games that suit their specific needs.

Unity's core engine features include a built-in game editor that allows users to create their own games in no time at all; support for multiple platforms including iOS and Android devices (de Macedo and Formico Rodrigues, 2011); and access to online resources such as forums and tutorials to help new developers learn how to use the software. Unity supports both 2D and 3D development, with nearly every feature needed for creating high-quality games available within the engine itself.

Unity has been used successfully by companies like Zynga (makers of Farmville), Disney Interactive Studios (makers of Infinity Blade), Rovio Entertainment (makers of Angry Birds), Gameloft (makers of Asphalt 8: Airborne), Square Enix (makers of Final Fantasy XV), Ubisoft (makers of Assassin's Creed Syndicate).

## 3.3 Blueprints

A blueprint is a model that describes the properties of an object in Unreal Engine 5. It can be used to control the behavior of objects in Unreal Engine 5.

Blueprinting is a way to create and program game characters, objects, and environments without having to write code. Blueprint provides a visual editor that allows you to drag and drop nodes onto each other to make up their behavior.(Sanders, 2016) This means you can create complex systems by combining small pieces into large ones.

Blueprints are created by placing nodes on top of each other in a graph editor. You can use these nodes as building blocks for your character or environment. This also means that you can reuse your blueprints across projects so that if you make changes to one, they will update everywhere it's used instead of having to remake everything from scratch every time.

The blueprint in unreal engine for npc Artificial intelligence has many benefits such as:

1) It helps us understand what our game needs from an AI perspective and allows us to focus on implementing it correctly without worrying about implementation details

2) It gives us a solid foundation on which to build upon later when we need more advanced capabilities

## 3.4 C#

C# is a language that is used to create applications and games for the Microsoft Windows operating system. It is a statically-typed, object-oriented programming language that supports functional programming constructs. The .NET framework is an implementation of this language designed specifically for use with the Common Language Runtime (CLR) (de Macedo and Formico Rodrigues, 2011). Unity is a game development platform from Unity Technologies that allows users to build high-quality 3D games using a drag-and-drop interface for creating content.

The Unity scripting language (Scripting) allows developers to write scripts in C# that can be compiled into native code for execution by the Unity engine. Scripts can also be written in JavaScript or Boo.js which are supported by both Mono and MonoDevelop, which are integrated into Unity's Asset Store repository.

Scripts can be executed from within the Unity editor window or from within MonoDevelop, as they are all built on top of MonoGame's ScriptableObjects functionality.

## 3.5 NPC

Non-player characters (NPCs) are a staple of any video game. They are the most important aspect of the game, as they serve as a means for the player to interact with the environment and other characters. NPCs can also serve as tools for the player to use in their quest. Non-player characters (NPCs) are often used in games to make the players feel more like they're part of a larger world (Diller et al., 2004). This can be done through various methods: they can act as a guide or friend, a character with whom the player can interact and learn about the game's world, or even as an obstacle that prevents players from progressing further. In general, there are three main parts to an NPC: its data store, its environment, and its behavior (Diller et al., 2004). The data store contains information about the environment and about itself as well as what it should do in accordance with its environment and other aspects. The environment contains all the things that will be affecting the behavior of an NPC through time such as weather conditions or other events that might occur during game play such as shooting enemies on sight or not shooting enemies on sight depending on whether or not there is a threat present (a threat being something harmful). The behavior refers to what happens when negative stimuli are present such as being hit by an enemy or the NPC.

# Chapter 4

# Methodology

In the following dissertation, we will go through main aspects of our work: research, design and the assumptions underpinning each phase. We begin by elaborating our research aims, by focusing on different categories of games NPCs that AI systems might be applied to, and we describe how these apply to game-specific design considerations such as game genres and player types. We then move on to describe our tooling for designing AI systems for games according to specific game. Next we outline our design decisions for modelling skill acquisition, which are influenced by past work on different RPG based games.

## 4.1 Methods Used in AI Games

The 3 types of Methods used in different types of games are

1. Finite State Machines (FSMs)
2. Goal-Oriented Action Planning (GOAP)
3. Behavior Trees

### 4.1.1 Finite State Machines (FSMs)

A finite-state machine, or FSM, is a computing paradigm based on a hypothetical machine with one or more states. Because only one state may be active at a time, the machine must transition from one to the other to accomplish distinct activities (Bevilacqua, 2013).It is a discrete-time automaton that has a finite number of states, a finite set of inputs, and an output. The FSM can be used to model computer programs and can be used to design NPC artificial intelligence (AI) systems. In a game, the FSM represents the AI's knowledge of the game's rules, which it uses to make decisions. The FSM that describes the game's rules is called its "state machine". The machine has a finite number of states and these states are represented by variables. Transitions between states represent actions that can be taken. Events are usually software/hardware interrupts that cause the machine to change its internal state. The event triggers a transition. Inside each state

The earliest work in this field was conducted by Carsten S. Hallermann and his colleagues around 1980 at MIT's Artificial Intelligence Laboratory (AIL). They were interested in creating a programmable computer that could play checkers against itself, using only a few bits of memory to store game states and rules. Their solution was very simple: they created a machine that consisted of two parts, an input-output processor (IOP) that controlled the board, and an IOP2 that controlled itself. The IOP2 would use its memory to track states for itself and its opponent, with each state being represented as two bits: one bit for the position of each piece on the board, and another bit indicating whether the piece had moved from its original position or not. The IOP2 would also keep track of its own moves, so that it would know when it had made a legal move.

The function of a machine's state is to determine what actions it will take when given certain inputs. In order for a machine to play a game well (that is, without making mistakes), its state must represent all possible states that are reachable from its current one in accordance with the rules of the game.

## Design of the game

The game is designed using Unreal Engine 5. It was chosen because of fast workflow and is easily manageable which results in much faster and efficient game development for a solo developer. Blueprints were used during the development of the game as the visual scripting language along with C++ which gave a good dynamic workflow. In order to properly implement FSM into an enemy AI, an RPG game is developed. The game is developed in an 3-d environment. the player in the game is facing the first enemy which is created by using finite state machine algorithm. The First enemy AI stands and guards the castle. once the player reaches in the range of the enemy it stops guarding and starts running towards the player and once it reaches player it starts attacking.

## System Architecture

The system architecture of finite state machines in AI NPC games. Finite state machines are useful for organizing the flow of logic in a program. In general, they make it possible to describe the relationships between states and transitions. It is also possible to represent any logic function with a finite state machine. A good example of a finite state machine is an NPC. FSM are generally Designed by using graphs where nodes are represented as states and edges as transition lines. The FSM states represent an action or a state of our NPC (Jagdale, 2021). Every state is connected to one or more than one state via transition lines. The transition from one state to another Is approved when certain conditions are met or fulfilled. In FSM it should be noted that no state should be unreachable from any other state. The looping in the game keeps updating by checking is that transition lines fulfils the condition or not. I have used fsm To my first enemy AI. The structure for the designed FSM is below.

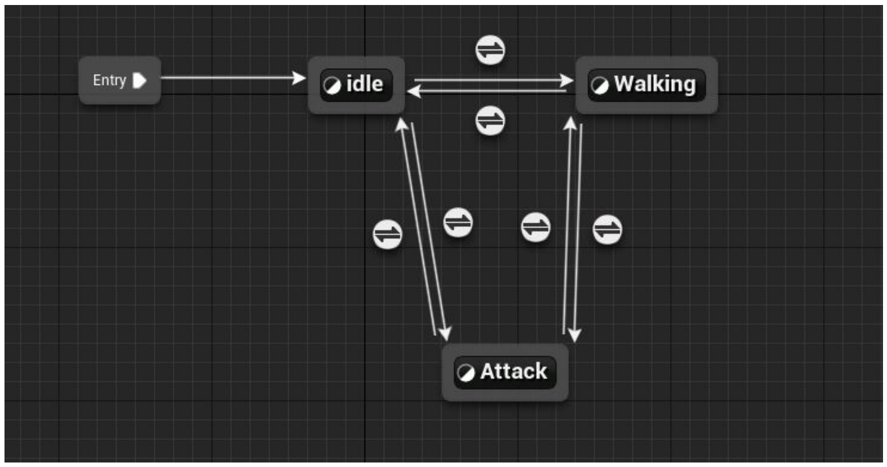
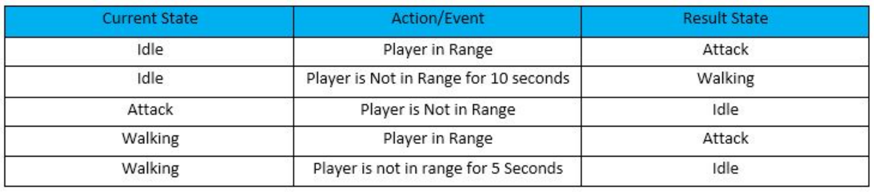


Figure 4.1: FSM implemented in Game

In this architecture initially the enemy is an ideal state. As soon as the player reaches in its range the state changes from ideal to chasing and the enemy AI starts chasing the player. If the player is close enough the AI will start attacking the player and cause damage. The AI will keep chasing the player and attacking Until it died or the player is out of its range. This loop will keep on repeating until either of these conditions are fulfilled. Above state diagram can also be written in the state transition table. The table contains 3 columns where the first one is the current state, the second one is the action or an event, and the third one is the resulting state. Below is the table representing FSM design to a state transition table.



| Current State | Action/Event | Result State |
|---|---|---|
| Idle | Player in Range | Attack |
| Idle | Player is Not in Range for 10 seconds | Walking |
| Attack | Player is Not in Range | Idle |
| Walking | Player in Range | Attack |
| Walking | Player is not in range for 5 Seconds | Idle |

Figure 4.2: FSM State Transition Table

### Strengths of using FSM

- FSMs are easy and quick to code so they are simple to debug as the game agent's behaviour is divided down into small bits (states). Non-programmers, such as level designers, game producers, and others, may easily debate the design of our AI using easily digestible pieces.
- They have a minimal computational cost since they effectively obey hard-coded rules as .
- They also allow you to create additional states and rules. As a result, we may simply change, edit, or broaden the scope of the agent's actions.
- They provides a stable foundation upon which other approaches, such as fuzzy logic or neural networks, can be built and sustained.

- FSM's allow for better testing. A developer can test their game by playing it exactly as they would if it were finished, which provides them with a good idea of how the game will play out when the player actually plays through it. This helps ensure that there are no bugs that could be fixed at this stage of development, but instead must be found during further playtesting later on down the line when players actually start playing through it themselves.

**Weakness of using FSM**

- FSMs have the greatest disadvantage due to their nature. Because FSMs can only be in one state at a time and there is no actual reasoning involved other than an if-then-else procedure, building complicated behaviours may be difficult and necessitates the introduction of several states or hierarchical state machines. It is demonstrated from the below graph that the more we increase the number of states into our state machine the more it gets complicated for the user to keep track of the actions implemented.

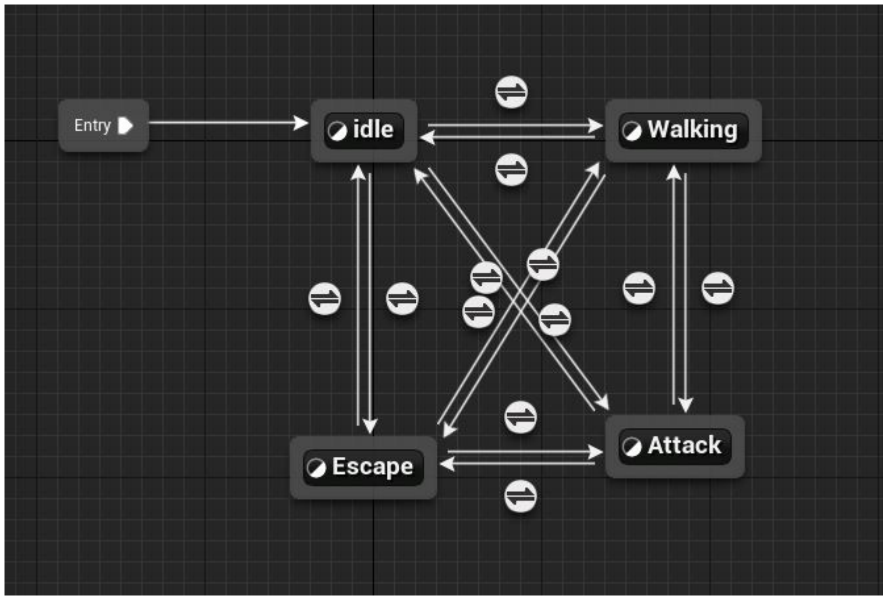


Figure 4.3: Complicated FSM State Example

- FSMs also have a high proclivity for becoming unmanageable, and they are unable to record related behaviours or profit from behaviour inheritance. As a result, they wind up repeating the same behaviour over several states.

- The FSM model uses decision trees to determine whether the current state is valid and how the system should change state. Each time a transition is made, new actions are performed or data is checked. Therefore, this type of model consumes time as it moves through its states (Andrea et al., 2014). They can be slow to learn and require a lot of memory space. This makes them less suitable for games that need fast reactions from their AI characters.

- Another problem with finite state machines is that they don't take into account human psychology or social interactions between players and their characters. If a player interacts with their character in an unexpected way, then the AI will not respond properly.

- They don't allow for autonomous behavior, which means that they don't allow the program to make decisions on its own without being told what those decisions should be by the user (i.e., by pressing buttons on a controller). This can make it difficult for developers who want to create games where their characters are allowed to roam around freely without being controlled by another player or another character within the game itself.

### 4.1.2 Goal Oriented Action Planning(GOAP)

Goal Oriented Action Planning is a field of classical AI that has been prominent for a very significant amount of time and has been widely explored in the scholarly world. GOAP is based on an alteration of the STRIPS which is characterized by utilizing objectives and operators (Burfoot et al., 2006). STRIPS objectives depict a few craved state of the world to reach, and administrators are characterized in terms of preconditions and impacts. An operator only executes in the event that all its preconditions are met, and each operator changes the state of the world in a way it impacts the character. It is a design methodology that has been used to create artificial intelligence (AI) systems that can play video games. GOAP was developed by Andy van Dam and Yann LeCun, two researchers at New York University who had previously worked together on New York University's AI program. The name "GOAP" was chosen to reference the main idea behind their project: building an AI system that could defeat players in games.

The first iteration of GOAP was published in 1993, but it wasn't until 1996 that the concept became popularized by van Dam and LeCun themselves in their book "The Master Algorithm." The book describes how to design an AI system for playing games using the GOAP methodology.

GOAP is briefly examined but only based on a condition as how the decision-making handle of each is carried out; basically planning is revelatory whereas FSMs are procedural. The three usual benefits, a planning framework offers to AI software engineers are the decoupling of objectives and activities, dynamic issue resolution and layering of behaviours.(Dawe et al., 2019)

Dynamic re-planning is additional characteristic of the GOAP system as emphasized by (Vassos, 2013) . The developer regularly monitors the agent's present plan by inspecting if it is still binding. If annulled, it is probable for the GOAP system to either plan towards the same goal choosing differing actions otherwise it can choose a new goal and plan towards that. This diverse re-planning can bring about unused and sometimes startling conduct that's highlighted 7 of the most highlights of the GOAP framework. For instance, the NPC is striking an opponent and it finds its current weapon is out of ammo, rather than checking the pertinence of each objective once more, the same objective is arranged as it proceeds and another way of fulfilling it is found by changing weapon, going looking for ammo or performing a skirmish attack.

GOAP activities and objectives are decoupled and segregated, so it is as it were the new activities or objectives that must be compiled to consolidate any changes. At an advanced stage this game can have hundreds of classes and compilation time can last hours, decreasing compilation times and broad recompilation is largely alluring.

One of the crucial characteristics of GOAP is that actions can be constructed upon one another i.e. layering of behaviours (Orkin, 2005). For instance, a basic Attack action will be designed which will give basic attacking feature, but it is possible to build on-top of this by creating extra actions such as Attack Range 10 Meters or Attack Range 5 Meters. These extra actions make use of most of the same functions as the attack action but bring about different behaviour by their activation, validation, and completion criteria.

## Design of the game

The game implementing GOAP is developed in Unity Engine stable version 2021.3.7f1 using C# as the scripting language. As a better understanding in implementing C# makes it easier to implement it in unity rather than in unreal Engine. While developing the game, version control was implemented to provide safety by providing backups along with many other features.

To properly implement GOAP, a First Person Shooter 3-D game is developed. To save the environment development time a base model of FPS is implemented into the unity environment after which the GOAP algorithm was implemented into the enemy. Pre-defined Objects(Prefabs) were created with physics dynamics and collision boxes such as environment and the avatars of the player. The main objective of the player is to kill all the enemies inside the game. The enemies have different states which changes according to the environment. The player has a limited number of bullets after which the gun takes a few moments to load again. The enemies have different goals, in the initial state the hovering enemy searches for the player randomly along a designated area. When they encounters the player the goal state changes from searching to attack until the player dies or till their health is below a certain level after which the best goal is to evade and regenerate to repeat the process till their final goal of killing the player is completed..

## System Architecture

The GOAP architecture is implemented using C# scripts in unity. The basic states are implemented by defining the basic NPC working by FSMs and the planning as well as implementation is achieved by making different classes for GOAP planner and IGOAP. The Architecture of GOAP is divided into different parts:

1. Actions An Action is anything which an NPC does ranging from playing a sound or an animation or changing its state. From idle to running behind an enemy is also a different state. It is an individual encapsulated entity and doesn't have to worry about what other actions are. Each action is given a cost to make GOAP decide what the next actions are to be introduced. A lower cost is always chosen over a higher cost. We choose the path with the lowest cost by adding up the costs of all the action in the sequence.

   We assigned some costs into our game:

   - EnemyAttack : 100 - EnemyRun : 50 - HitPlayer : 75

   After carefully observing the above sequences and adding up the total cost of theses actions, we can deduce the cheapest sequence.

   - Idle State -> Enemy attack(100) -> Hit Player(75) -> Kill Enemy

- Idle State -> Hit Player(75) -> Kill Enemy

- Idle State -> Enemy attack(100) -> Hit Player(75) -> EnemyRun(50) -> Kill Enemy

Here, to kill the player is the final goal state and the starting position is idle, to kill the player the enemy has to run towards the player and then starts attacking which cost 175 while if the enemy is nearby it just stars killing with the cost of 75. But the Goap doesn't always choose the cheapest path due to preconditions attached to the actions.

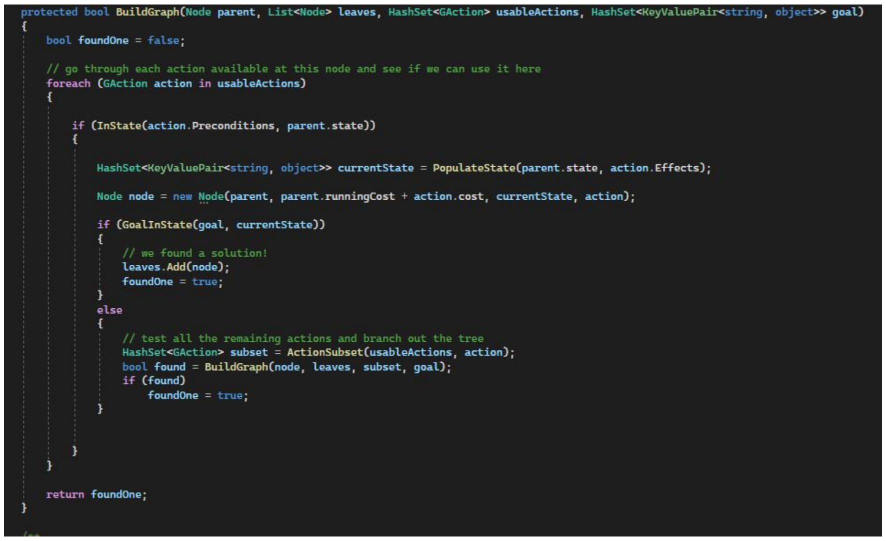
2. Preconditions and effects

   Actions have preconditions and effects. A precondition is the state that is required for the action to run, and the effects are the change to the state after the action has run (Orkin, 2004).

   As the EnemyAttack action is initiated it is also damaged by the player so EnemyRun action requires the enemy to have a health less than 25 to run. If the enemy has more than 25 health it will keep on the same state of EnemyAttack. Thus it needs to fulfill the precondition in order to run the EnemyRun action which then goes behind the bigger turret to hide from the player.

3. The GOAP Planner

   This code class represents the actions preconditions and effects and creates queues of action to complete a final goal. The goals are given by the enemy class, with world state and the actions performed by it (Orkin, 2004). With the code in the GOAP planner class it can order the series of action which can be implemented along with its preconditions and effects and decide the best path for the actions to be performed.



```
protected bool BuildGraph(Node parent, List<Node> leaves, HashSet<GAction> usableActions, HashSet<KeyValuePair<string, object>> goal)
{
    bool foundOne = false;

    // go through each action available at this node and see if we can use it here
    foreach (GAction action in usableActions)
    {

        if (InState(action.Preconditions, parent.state))
        {

            HashSet<KeyValuePair<string, object>> currentState = PopulateState(parent.state, action.Effects);

            Node node = new Node(parent, parent.runningCost + action.cost, currentState, action);

            if (GoalInState(goal, currentState))
            {
                // we found a solution!
                leaves.Add(node);
                foundOne = true;
            }
            else
            {
                // test all the remaining actions and branch out the tree
                HashSet<GAction> subset = ActionSubset(usableActions, action);
                bool found = BuildGraph(node, leaves, subset, goal);
                if (found)
                    foundOne = true;
            }


        }
    }

    return foundOne;
}
```

Figure 4.4: GOAP Planner

The code Figure 4.4 implemented above gives a better understanding of application of preconditions in GOAP planner. Until the precondition is activated the GOAP will try to find the least costly path to achieve the goal state which in this condition is killing the player.

4. Procedural Preconditions

   These are the conditions which are needed to be checked before the pre conditions and effects are activated. For instance, the EnemyRun action has the precondition of "A turret Enemy is available" which it will search in the world. This type of precondition is procedural and requires a certain code to be implemented. These procedural preconditions takes a while to run so its preferred to be performed on Coroutines in unity. The same conditions can be applied to effects too and by changing the cost of the actions the results can be much more dynamic.

5. GOAP and State

   The GOAP system implemented is organized in small Finite State Machines and Goal states. The three basic Finite states are:

   -Idle -MoveTo -PerformAction

   The idle state is the initial state of our enemy or NPC. This code is handled outside of the GOAP architecture. The Goap will just initiate the actions which are to be performed. When it chooses the action from the FSM the action is passed to the GOAP planner with world and the NPCs initial state, and the GOAP planner will return a set of actions.

   The planner will try to execute the first action from the list of actions provided by it to the enemy. If the actions are in the range then the FSM will try to push it from idle state to the next state MoveTo. This will inform the enemy to move to the player location. The enemy will then move to the player range and then the PerformAction state can be pushesd in.

   The GOAP planner then will run the next PerformState Action in the list. This action can be instantaneous or perform for an extended period until its completed and then the next action can be performed after checking if it needs to be performed.

6. A-Star Algorithm

   In GOAP architecture, every AI has a fixed number of goals which contains "insistence", and are represented by a number. The higher the number, the more our NPC ai will work towards achieving the goal. Each NPC AI has a number of actions which it takes to reach the goal in minimum cost or number. Here, for killing the player attacking has a higher number until its health is below 25 then the goal changes to hide to regenerate until its health is full. This logic is introduced by the use of A-Star algorithm. Generally A-star is used as a searching or path-finding algorithm in spatial graphs. According to Orkin (2005) in his GOAP paper it can be easily implemented for searching through all the actions an NPC can take. In this model, the actions taken by the NPC results in connections and then creates a path which leads towards the end goal called a Plan. But the application of A* in this game is obtained by implementing the cost values in the world state which acts as a repository of variables that the NPC requires to understand the world. Using these world states, the NPC can have a more complex goal handling (Doris and Silvia, 2007). GOAP uses a duplicate world state in order to track the progress towards the goal. A* uses a distance calculation as a heuristic approach to find the next action to the goal state path. These actions between world states acts as a connection and are assigned different cost which the A* algorithm takes into consideration to find the least expensive path.

## Implementation

Monolithic approach was implemented while creating the classes. Each class is self-contained and independent. It resulted in a much better environment for debugging and modifying the functionality of each class. Once the game starts all the assets are loaded and the IGOAP script starts initiating the states. It pushes the initial state FSM into it which contains IdleState, moveToState and performActionState. Each State contains predefined conditions which pushes the change of states once its fulfilled. The GAgent script updates the states and provide different actions present in those states by calling the GAction scripts which holds all the preconditions and effects applicable in those NPC circumstances. The cheapest course of actions to reach the goal state is calculated using GPlanner which contains the A* heuristic search. In this game the initial state is to be idle and find the location of the target player and the final state of the NPC is to kill the player before it is being killed.

## Strengths of GOAP

- Benefits to Runtime Behaviour – An NPC that governs his own plan at runtime can tailor fit his actions to its existing environment, and vigorously finds an alternate set of solutions to deal with problems.(Heinimaki and Vanhatupa, 2013)
- Benefits to Variety - The structure levied by GOAP is superlative for producing a diversity of character categories which display dissimilar behaviors, and also share behaviours across numerous projects. The character is provided with a pool of actions from which to search for a plan.
- Goal oriented action planning helps game characters make decisions based on their goals. This means that if you want your character to go left instead of right, it will be able to do so. This can be particularly useful for characters who are trying to complete a mission or accomplish some other task at hand.
- It allows for more complex decision making than other methods like heuristics or Bayesian decision theory would allow. Because goal oriented action planning allows for both long-term planning as well as short-term decisions, it can handle situations where your character needs to plan for something in advance but also needs to take immediate actions once they arrive at their destination
- It can be used to create more realistic behaviors by allowing an AI system to mimic human behavior. In addition, goal oriented action planning is useful for reducing uncertainty in an AI system's environment by giving it a set of predetermined actions that it must perform.

## Weakness of GOAP

- Large Resource Usage - GOAP burns out a lot of processing power while creating a plan- in Artificial Intelligence for Games, Millington and Funge (2018) suggests that: Amount of searchable possibilities = number of actions m̂aximum actions in a plan.
- Less Direct Control - While AI controls what to do at runtime is less taxing, it however grants lesser direct control over what actions may or may not shadow each other. For instance: If the opponent's characteristics parallelly try to establish a pace for the

player, more energy may be required to safeguard that actions are specifically executed at required events in time.

- The entire system is based on goals, which can be difficult to maintain over time. If there is a change in the environment, or an update to the game code, then it may be necessary to change the goals of the AI. This can lead to problems with the AI not being able to adapt to new situations.
- It relies heavily on what has already been programmed into the game, meaning that if something unexpected happens (such as a bug), then it could cause problems for players who have already played through most of their game.
- It does not take into account how all parts of the system work together. If one part of the system fails for some reason (such as a bug), then it could cause problems throughout all different parts of the system. This means that even if another part works perfectly fine, there could still be issues related to other parts working incorrectly due to bugs within those other components

### 4.1.3 Behavior Trees

The behavior tree is a pattern of game AI that uses an iterative, hierarchical process to solve problems. The behavior tree can be seen as a generalization of the traditional AI approach to game AI, where instead of using fixed rules and finite states, it uses more naturalistic behavioral patterns and allows the agent to learn from its experience in the world. The algorithm was first proposed by Demis Hassabis, who also founded DeepMind, a company that makes artificial intelligence (AI) software. The behavioral trees in games for AI are a new way of creating complex behavior for NPCs and creatures. They are different from traditional AI methods in that they focus on the movement of the character, rather than other aspects of the game like combat or story Silver et al. (2017). A Behavior tree defines a structure of functions that are executed in both time sequence as well as parallel utilizing a set of instructions known as operators or nodes.

Since its Conception in the Halo 2 (Järvinen, 2002), behavioural trees have been a popular method for modelling artificial intelligence (AI) in games. The concept for BTs arose from earlier work on designing story character behaviours (Bates et al., 1994). BTs are frequently used to illustrate the decision-making of a non-player character (NPC). They're also gaining traction as a good way to operate robots (Colledanchise and Ögren, 2014).

These trees can be used to create natural interactions between characters, such as when a character is trying to get past another character who is blocking their path.The behavioral trees also allow for more complex behaviors that are not possible with traditional AI methods. For instance, you could use them to create an NPC who finds a way out of their current situation by using objects around them or using their environment

#### Design of the game

The Behavior tree in the game is implemented using unreal engine 5 using blueprints and Blackboard. Some classes from the FSM architecture are implemented into it such as walking and running of the enemy. The enemy is given human mimicking feel as a guard. The enemy patrols in a fixed path until the condition of CanSeeEnemy is fulfill. Once it changes from cannot see to seeing the enemy it starts chasing the player and attacking it. Another condition is placed inside the tree when the player gets out of sight perception system of AI the enemy will remember the last known position of the player and then scout the area for some time after which the enemy will again start to patrol from its original position much like a real time guard.

## System Architecture

Behavior trees are a very common design pattern in the field of Artificial Intelligence and Game programming. Actors can be created, updated, and deleted in Unreal Engine, so they can be fully controlled by a Behavior Tree. Behavior trees are useful because they are easy to use and well documented. The Unreal engine has many built-in functions that allow developers to easily make behaviors, it also gives programmers access to designers while giving them some freedom of expression as well. The behavior tree has a left to right priority ie. it gives higher preference to the left nodes than the right nodes.

The Architecture of Behavior tree is divided into different nodes

1. Root node : This node represents the starting point of the Behavior trees. Its a unique node and have some predefined rules. It doesn't support decorator or service nodes and can have only one connection to it. Eventhough it has no special properties of its own, selecting the node by clicking on it will show the properties of behavior tree by which we can set the Blackboard assests of the Behavior tree.

2. Composite node: The composite node defines the root of a branch and the base rules for how that branch is executed inside the tree. It is the first starting point from the root node and contains more than one thing to execute within them.

   There are three types of composite nodes:

   - Selectors : They go from from left to right searching for a successful node until it finds a successful node it will keep on finding the next nodes until one of them succeeds and then it goes back up into the tree (Iovino et al., 2022). Here, we have applied two selector nodes. First one is to select between CanNotSeePlayer and CanSeePlayer nodes while the second one is in CanNotSeePlayer which decides when to patrol the path and if the player got out of sight.
   - Sequence : They also starts from left to right until a node fails. Once the node is successful, it will get to the next node. Until one of the nodes fail it will keep on repeating the tasks. When the node fails it will go back up to the tree. Here we have used different sequence nodes for different function of the NPC such as HavebeenChasing, Loop, PatrolPath, PatrolPathType, ChasePlayer and AttackPlayer.
   - Simple Parallel: In this all the sub nodes are executed parallelly the single task which is in purple in parallel with subtree which is gray in color.

3. Decorators : These are the conditional type nodes attached to another node and decides whether the branch in a tree or even a single node can be executed. The composite decorator node enables us to introduce more complex logic than the pre-defined nodes. Once you add it to a node, by double clicking on it the composite graph can be opened

where further standalone nodes can be added using operators for more complex functionality (Iovino et al., 2022). Various different types of nodes such as blackboard decorator node and loop decorator node .

4. Services : These nodes are being executed at a defined frequency as long as their branch is executed. It updates the nodes when it is in running nodes. ChangeNpcSpeed and IsPlayerInRange are few services used inside the behavior tree.

5. Tasks : These nodes are the actual actions which operates the NPC and are the last one in each branch of the behavior tree (Iovino et al., 2022). Conditionals and sequence node can be added to these nodes to give more functionality to them. MoveTo Playing animations, Running, Attacking, searching and FindRandomLocation are some of the actions used in our BehaviorTree.

### Implementation

The behavior trees works on Depth First Search algorithm (DFS) which completes the first branch from left to right until the conditions are fulfilled. Once we have made our basic functionality of the NPC such as walking, running and attacking animations the behavior tree is implemented. we start by creating a Blackboard having Blackboard keys which contains all the values of the variables used inside the blueprint namely target_Location, CanSeePlayer, PatrolPathVector, PatrolPathIndex, Direction, WaitTime and ChaseStatus. It acts as a reference table for the NPC to retrieve information regarding the environment around the NPC. Initially from the root node a selector node is initiated which delves into two different nodes which can be shown in fig 4.5 and Fig 4.6 as CanNotSeePlayer and CanSeePlayer which are further divided into sub nodes based on different input and environment conditions.

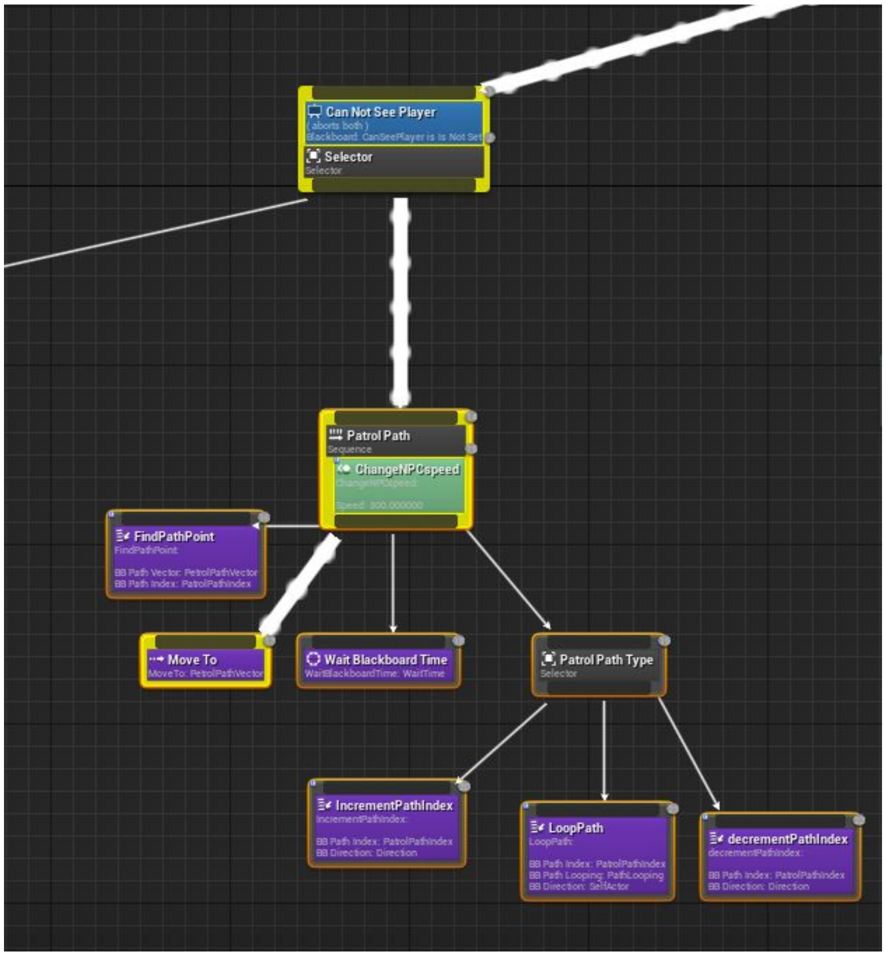


Figure 4.5: Cannot See Player Behavior Tree

When the NPC cannot see a player it contains two nodes to choose either it can roam in a particular specified points or after seeing if the player is out of sight as shown in fig 4.5 and fig 4.7. The PatrolPath sequence node is initiated and the NPC starts roaming in the specified path points initiated by PatrolPathType selector node which find the points placed inside our map and then FindPathPoint and MoveTo action nodes are instigated which are shown in fig. 4.5

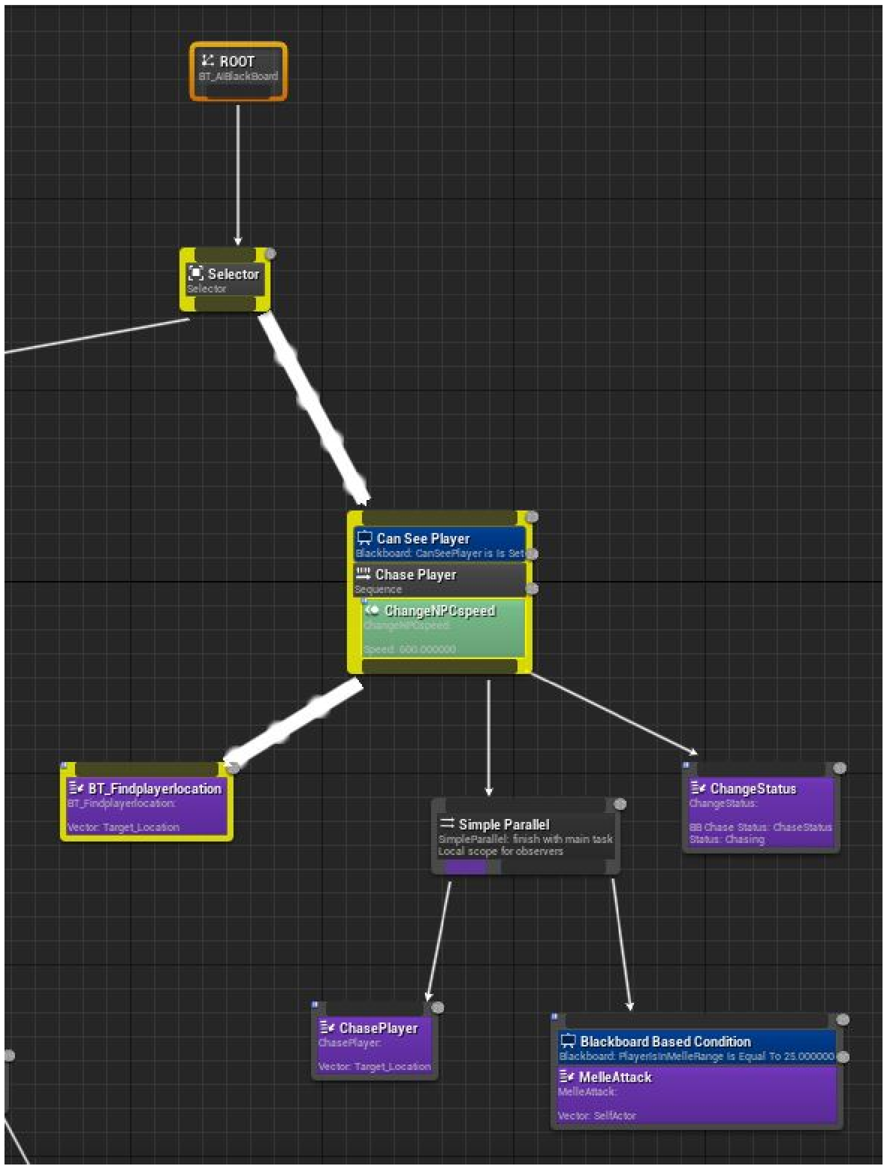


Figure 4.6: Can See Player Behavior Tree

While roaming the ChangeNPCspeed service node is initiated which updates the speed of our NPC to 300. Once the player is in the visual range the NPC starts chasing the player as the CanSeePlayer decorator sequence node is instigated along with ChangeNPC-speed service node which changes the speed of our NPC from 300 to 600. At the start, the BT_Findplayerlocation Action node finds the vector location of the player then start chasing to that location.

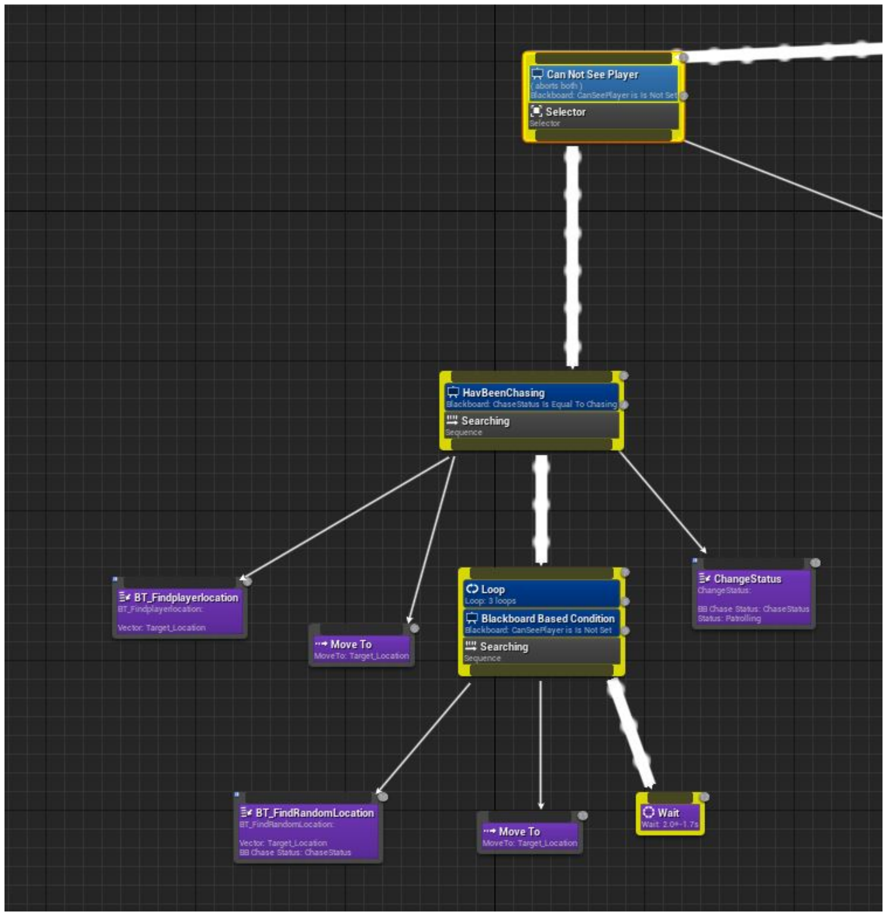


Figure 4.7: Player Out of Sight Behavior tree

Here, a simple parallel composite node shown in fig 4.6 is placed which parallelly updates the MelleAttack action node with chasing node and once the blackboard condition of PlayerInRange is fulfilled it starts attacking the player along with chasing it. If the player is out of range it again goes to CanNotSeePlayer selector node where it instigates HavBeenChasing sequence node which finds the last known location of the player and starts moving towards it. Once NPC reaches the location point it starts searching for the player in nearby random location using Searching sequence node for 3 loops which is defined by placing a loop decorator node. As it completes the loop a change status node changes the current path from HavBeenChasing to Patrol path sequence node.

### Strengths of Behavior Trees

- Code Reusability: In any large, complicated, long-term project, having reusable code is critical. The capacity to reuse designs is dependent on the ability to construct larger objects from smaller components, as well as the independence of those parts' input and output from their application. Each module must interface the control architecture in a clear and well-defined manner to allow code reuse. Because each subtree may be reused in several places of a BT with correct implementation, BTs enable reusable code.

- Furthermore, while developing code for a action node, the developer just has to worry about giving the right return status, which is always either Running, Success, or Failure.

In contrast to FSMs, whose outbound transitions need knowledge of the following state, leaf nodes in BTs are built without concern to which node will be performed next. As a result, BT logic is separate from leaf node operations and vice versa. Reactivity refers to the ability to respond to changes quickly and effectively.

### Weakness of Behavior Trees

- BTs are unable to express all game entity behaviours in a straightforward manner. They are not well adapted to modelling state-based behaviours, for example, especially when an entity must respond to external events, such as halting a cowardly agent's current patrol path to go into hiding when the player is discovered. Modeling cooperation is especially challenging since it requires coordinating activities amongst behaviour trees.
- Another drawback is that behaviour trees demand a lot of memory, especially if each agent has its own behaviour tree.

# Chapter 5

# Experiments

## 5.1 AI Memory Resources

The AI algorithms implemented inside the game engines have a transition rate per frame from one state to another which describes the usage of CPU resources within that frame. It gives a better understanding of the resources used by different algorithms. In this analysis we are playing the games created and record the statistics used by each AI and compare them according to the retrieved values per frame. This is the most probable way to compare between different algorithms statistically. As Finite State Machine and Behavior trees are structured in unreal engine we have used the inbuilt tool unreal insights which shows our AI's transition time in real time and for the Goal Oriented Action Planning AI we have used unity's inbuilt tool Unity profiler. As from the statistics Figure 6.1 below there are many parameters to compare but inclusive max is the most appropriate method in our case for unreal engine as it resembles to the feature behavior update in unity to perform a comparison analysis between the AI methods in different game engines. Inclusive max is the maximum amount of time spent in a function and its child function ie. the time from start to the return of the function while the update behavior in scripts inside unity is the total amount of time for a function script to complete from start to end.

## 5.2 Developer Survey

To get a clearer understanding a survey was created using google forms for the game developers based on the type of AI they use for developing the games along with the codes used here in these games. The survey was posted in one of the biggest game developer community on discord server called GameDevRaw. There were 78 participants for the survey from different parts of the world. The survey represented a good view of the current industry is using the AI algorithms in a game NPC. Out of these 78 participants 25 of them opted FSM algorithm to implement in their game while 18 of them resorted to GOAP as their AI algorithm and the remaining 35 chose Behavior tree as their main method to incorporate in AI development.

## 5.3 Player Survey

Another survey was conducted to determine the experience of the gamers about the NPC they found better. So a batch of 12 participants from my friends was made to play the games created and neither of them were reported the AI they were playing against. Three games were played by each of them and a questionnaire was given to them after playing the games including a question "Which game they liked the most and why?". Even though not all the functionalities of our games are common but the basic feel of the algorithm is visible in them and this survey gives a better understanding of a players mindset and the strengths and weaknesses of our AI in a qualitative aspect. Only 2 participants chose FSM AI over others showing their basic functionality against other AI. While 4 others chose GOAP AI and the other 5 chose Behavior trees as the AI they liked.

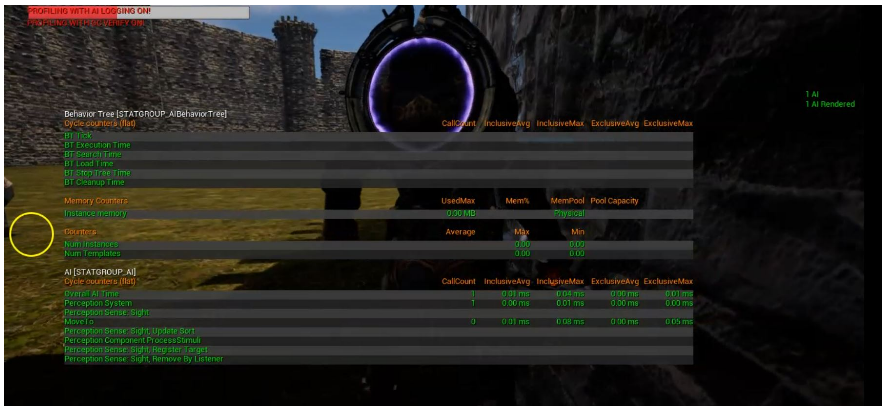
# Chapter 6

# Results

This chapter contains the results of the experiments conducted previously in the project. It includes the memory resources usage values of each AI algorithm, and an overview of the surveys collected during the qualitative analysis. Overall, 88 people participated in the study. The participants were used for qualitative analysis by using survey forms.

## 6.1 Resources Usage Data

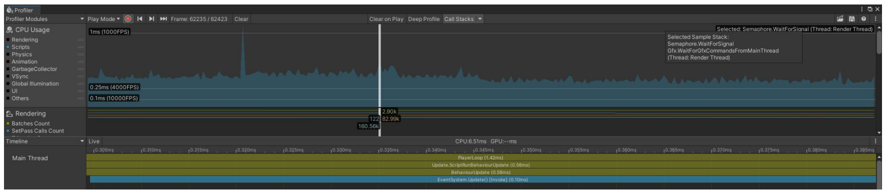
Each AI algorithm has a different way to be implemented in an engine. For FSM's a StateMachine(inherited) Blueprint Class is generated which contains predefined events such as OnBegin, OnUpdate and OnExit which makes it easier to develop functions inside the the class for our AI. In it Animation Blueprints for a skeletal mesh are built to transit from one state to another as shown in Fig: 4.1 which makes it easier to debug and add more states to it. For GOAP not a lot of resources are available inside the engines so a part of it needs to be coded in an engine which makes it complicated for a beginner game developer and needs a lot of research before implementation as A-Star searching algorithm as well as the states are needed to be coded inside it for cost implementation. This algorithm is generally implemented by experienced developers depending upon the AI requirements where goal states are needed to be implemented such as strategy games. For Behavior trees unreal engine has a predefined Blueprint class called BehaviourTree which contains all the basic nodes defined in section 4.1.3 which helps to implement functions of the AI and change its behavior. While its complicated to determine the best one as it's based many factors such as the experience level of the coder and type of game. As we can observe in pictures 1, when a single FSM AI is running the inclusive max in overall AI time for a frame is 0.04 ms. It shows that the total time to run the AI scripts inside the engine.



```
PROFILING WITH AI LOGGING ON!
PROFILING WITH GC VERIFY ON!

Behavior Tree [STATGROUP_AIBehaviorTree]
Cycle counters (flat)                        CallCount  InclusiveAvg  InclusiveMax  ExclusiveAvg  ExclusiveMax
BT Tick
BT Execution Time
BT Search Time
BT Load Time
BT Stop Tree Time
BT Cleanup Time

Memory Counters                              UsedMax        Mem%        MemPool  Pool Capacity
Instance memory                              0.00 MB                    Physical

Counters                                     Average         Max           Min
Num Instances                                               0.00          0.00
Num Templates                                               0.00          0.00

AI [STATGROUP_AI]
Cycle counters (flat)                        CallCount  InclusiveAvg  InclusiveMax  ExclusiveAvg  ExclusiveMax
Overall AI Time                                      1       0.01 ms       0.04 ms       0.00 ms       0.01 ms
Perception System                                    1       0.00 ms       0.01 ms       0.00 ms       0.00 ms
Perception Sense: Sight
MoveTo                                               0       0.01 ms       0.08 ms       0.00 ms       0.05 ms
Perception Sense: Sight, Update Sort
Perception Component ProcessStimuli
Perception Sense: Sight, Register Target
Perception Sense: Sight, Remove By Listener
```

Figure 6.1: FSM AI Rate

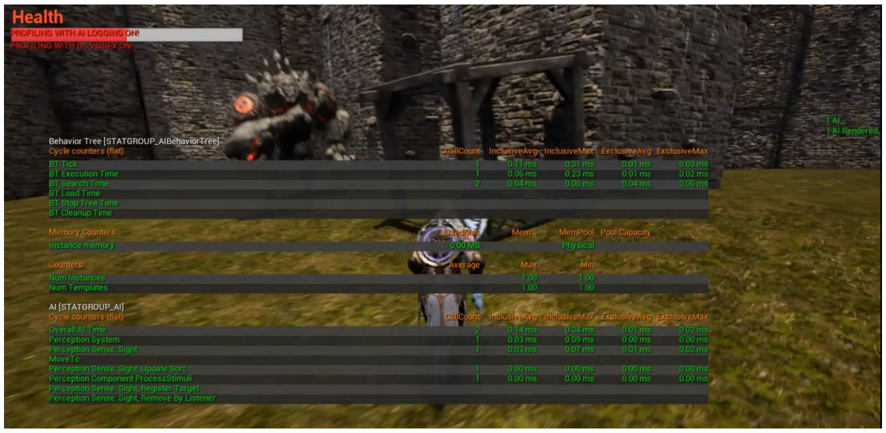
According to picture 2, When a single GOAP AI is running the update behavior in overall AI time for a frame is 0.56 ms and represents the total time to run the GOAP scripts using A* algorithm.



```
Selected: Semaphore.WaitForSignal (Thread: Render Thread)
Selected Sample Stack:
Semaphore.WaitForSignal
Gfx.WaitForGfxCommandsFromMainThread
(Thread: Render Thread)

PlayerLoop (1.42ms)
Update.ScriptRunBehaviourUpdate (0.56ms)
BehaviourUpdate (0.56ms)
EventSystem.Update() [Invoke] (0.10ms)
```

Figure 6.2: GOAP Script Behavior Update

As in picture 3, When a single Behavior Tree AI is running the inclusive max for a frame is 0.34 ms and represents the total time to run the AI scripts inside the engine



```
PROFILING WITH AI LOGGING ON!
PROFILING WITH GC VERIFY ON!

Behavior Tree [STATGROUP_AIBehaviorTree]
Cycle counters (flat)                        CallCount  InclusiveAvg  InclusiveMax  ExclusiveAvg  ExclusiveMax
BT Tick                                              1       0.11 ms       0.31 ms       0.01 ms       0.03 ms
BT Execution Time                                    1       0.06 ms       0.23 ms       0.01 ms       0.02 ms
BT Search Time                                       2       0.04 ms       0.06 ms       0.04 ms       0.06 ms
BT Load Time
BT Stop Tree Time
BT Cleanup Time

Memory Counters                              UsedMax        Mem%        MemPool  Pool Capacity
Instance memory                              0.00 MB                    Physical

Counters                                     Average         Max           Min
Num Instances                                               1.00          1.00
Num Templates                                               1.00          1.00

AI [STATGROUP_AI]
Cycle counters (flat)                        CallCount  InclusiveAvg  InclusiveMax  ExclusiveAvg  ExclusiveMax
Overall AI Time                                      2       0.14 ms       0.34 ms       0.01 ms       0.02 ms
Perception System                                    1       0.03 ms       0.09 ms       0.00 ms       0.00 ms
Perception Sense: Sight                              1       0.03 ms       0.07 ms       0.01 ms       0.02 ms
MoveTo
Perception Sense: Sight, Update Sort                 1       0.00 ms       0.00 ms       0.00 ms       0.00 ms
Perception Component ProcessStimuli                  1       0.00 ms       0.00 ms       0.00 ms       0.00 ms
Perception Sense: Sight, Register Target
Perception Sense: Sight, Remove By Listener
```

Figure 6.3: Behavior trees AI Rate

Even though the type of functions each algorithm contains differ by each other in some manner there is a clear difference between the memory resources used by each one of them. While the FSM algorithm uses the least resources, it also has to compromise in the quality of AI behavior as it exists only in one state at a time resembling an inexpressive AI. As the complexity increases the resources used are greater. The GOAP is the costliest AI algorithm with a value of 0.56 ms to implement as it requires constant update of the paths to take for the cheapest cost using A* algorithms. It needs to continuously update the scripts to have a more functioning AI. The Behavior trees are the most optimal ones to use with a value of 0.34 ms memory usage by not compromising the functioning of AI. It gives a good balance between functionality and resources consumed as it works on Depth First search(DFS) algorithm in a tree. It doesn't have to check each and every node and keep processing just a single branch until the conditions are met. A bar graph has been constructed showing the use of memory resources by each AI algorithm.

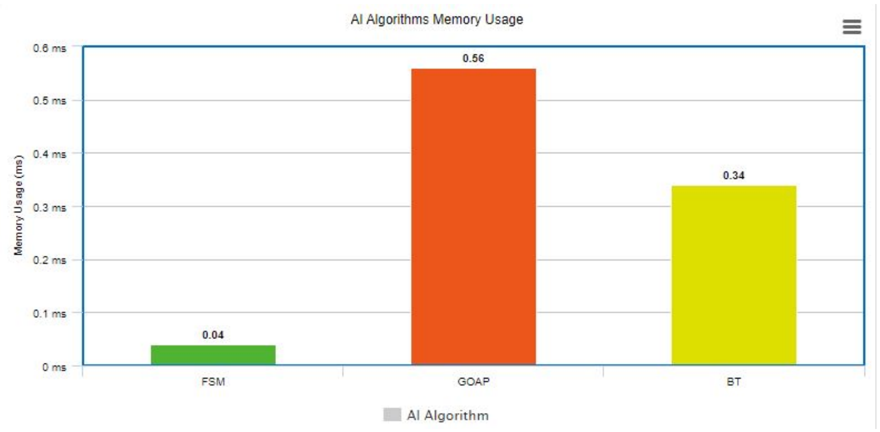


Figure 6.4: Comparisson Memory Usage AI

This gives an extensive idea about the memory usage of each AI but it also depends upon the type of game and the type of functionality a developer needs from an AI while FSM's are great for small functioning AI and uses less resources it cannot mimic the human type complexity for our NPC AI.

## 6.2 Qualititative Data

As discussed in chapter 5.2 and chapter 5.3, the surveys of developers and players were taken. The data from these surveys are taken and presented in the following sections.

### 6.2.1 Developer Statistics

The survey created collected the data from developers around the world and gave an insight of the way the developers use different algorithms in their games. It was created to analyze the general perception of the developer community about these algorithms and compare with our findings to validate them.

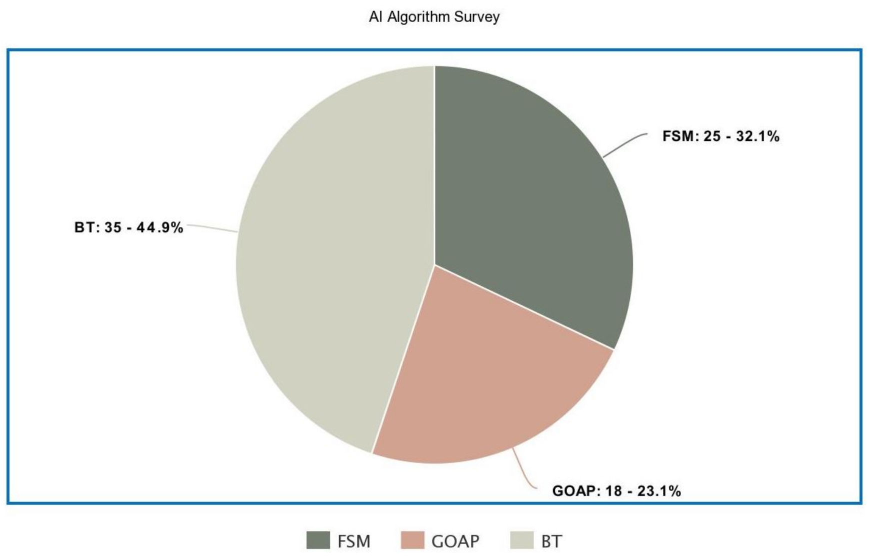


Figure 6.5: AI algorithm survey

The pie chart above represents the algorithm distribution among game developers. As it can be seen from the distribution around 32% of the developers use Finite State Machines which represents quite a large number of people still uses FSM due to its simplicity and ease, around 23% uses Goal Oriented Action Planning showing it is still used by developers and the rest 45% uses behavior trees which shows a popularity of using them in game AI and their ease of development. Most of them chose Behavior trees as it gives an easy way to add new functionality to AI with least amount of bugs as the distribution of tasks is clearly defined in them each branch acts independently. While for goal driven tasks such as in strategy games where a goal completion is the main task GOAP is used as a better alternative than other AI's.

### 6.2.2 Player Statistics

In this survey, the data is collected by the player who played the game and chose the best AI which they liked the most while interacting with it. The survey was taken with 10 people and 5 out of them chose Behavior tree which further shows the implication of using it, the more intelligent behavior is possible in it with ease. The feedback from the players were also recorded while most of them were satisfied with the AI algorithm some improvements were also suggested regarding the movement of the NPC as well as the improvisation of the AI perception system into it such as hearing should be incorporated into it to give it a more intelligent behavior. These systems are easily possible in Behavior Tree rathe than the other AI approaches which whill require an extensive process to be infused in the predesigned algorithms. This gives a better understanding of the strengths and weaknesses of these AI approaches

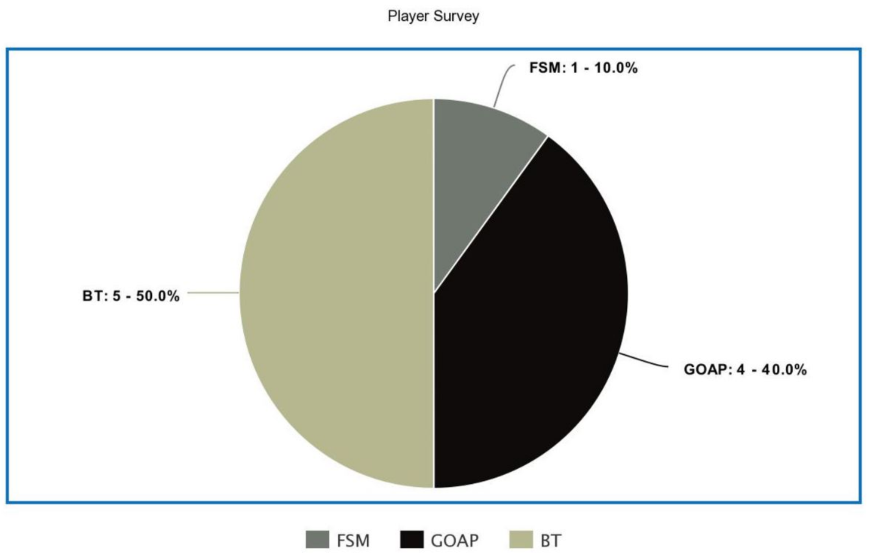


Figure 6.6: Player Survey Chart

In the Fig. 6.6, It can be clearly observed that the FSM occupies half of the votes regarding the choice of AI approach while 40% votes are given to the GOAP approach and 10% is given to FSM. This distribution shows that the observations made in the thesis about different algorithms is true. The developers use Behavior tree as it gives an intelligent AI which in response make the players engaged into the game.

# Chapter 7

# Conclusion

This study was successful in discovering the strengths and weaknesses of the three AI approaches, namely, FSM, GOAP and Behavior trees into incorporating them into our games. In particular, the resource usage method gave a better understanding of the strengths and weaknesses in terms of resources used while processing the tasks using unreal insights and unity profiler. It showed a greater view of the implementation of these tasks and helped in debugging the game's performance issues.

A few big problems that were encountered in this study were during the implementation of the GOAP algorithm, even though some theory is available about it by orkins() a practical approach was lacking for it on unreal engine. While the FSM's and Behavior trees were implemented easily, the GOAP algorithm was a staggering problem. It was overcome by using c# language inside unity, whose structure made it easy to implement the algorithm. Another issue was faced while searching for the tests to conclude these aims. As there is little documentation available regarding the analysis of scripts or process, it was a daunting task to find the tools required for extracting the script data. A lot of research went into creating a testing task to verify the findings of the algorithms.

The aim of the project is achieved by implementing these methods inside the games and understand the working of each AI and comparing them with each other to better streamline the relationships between them. As we incorporated more sophisticated actions or tasks, the more it was comprehensible to deduce a difference between their strengths and weaknesses.

Since the games implemented are small, a clear distinction is hard to make, but still a lot of visible differences are observed, which are explained in chapter 4 of the project.

The conclusion from this project is that by implementing different algorithms in an NPC, we can make it much smarter and humnoid by keeping in mind the strengths and weaknesses of the algorithm and incorporating them according to the requirements of the game. But much more research is needed to find the perfect solution.

## 7.1 Suggestions for Future work

The future of AI in games is an exciting time for game developers. AI technology allows developers to reduce programming and design time, allowing them to create more advanced and challenging games.

Since a long time, the field of AI in games was dormant because of the limited resources available to the individuals as well as more emphasis was placed on graphical part of the game. But recently, as the advancements in hardware technologies is exponentially increasing the focus on developing much better AI's are in motion and are generating media interest and investment. There are many other algorithms which are being used in developing much smarter AI other than the one's discussed. Utility AI, Neural Networks, deep learning methods, Natural Language Processing(NLP) are some of the algorithms which are started to being used in the main stream AAA studio games. With the rise of AI in video games, it plays a crucial role in our everyday lives. However, they still aren't able to replicate human intelligence rather than run automated systems that are capable of playing games.

The future of AI in gaming will be different to the past but still contain many exciting developments. The technology will continue to advance, with more complex games becoming possible and new techniques being developed to help players in their interactions with both the game world and the AI.

# Bibliography

Andrade, A. (2015). Game engines: A survey. *EAI Endorsed Trans. Serious Games*, 2(6):e8.

Andrea, R., Akbar, R. I., and Fitroni, M. (2014). Developing battle of etam earth game agent with finite state machine (fsm) and sugeno fuzzy. *ICCS Proceeding*, 1(1):184–187.

Bates, J. et al. (1994). The role of emotion in believable agents. *Communications of the ACM*, 37(7):122–125.

Bevilacqua, F. (2013). Finite-state machines: Theory and implementation.

Bigdata (2021). History of ai use in video game design.

Bourg, D. M. and Seemann, G. (2004). *AI for game developers.* " O'Reilly Media, Inc.".

Burfoot, D., Pineau, J., and Dudek, G. (2006). Rrt-plan: A randomized algorithm for strips planning. In *ICAPS*, pages 362–365.

Colledanchise, M. and Ögren, P. (2014). How behavior trees modularize robustness and safety in hybrid systems. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1482–1488. IEEE.

Dawe, M., Gargolinski, S., Dicken, L., Humphreys, T., and Mark, D. (2019). Behavior selection algorithms: an overview. *Game AI Pro 360*, pages 1–14.

de Macedo, D. V. and Formico Rodrigues, M. A. (2011). Experiences with rapid mobile game development using unity engine. *Computers in Entertainment (CIE)*, 9(3):1–12.

Diller, D. E., Ferguson, W., Leung, A. M., Benyo, B., and Foley, D. (2004). Behavior modeling in commercial games. In *Proceedings of the 2004 Conference on Behavior Representation in Modeling and Simulation (BRIMS)*, pages 17–20.

Doris, K. and Silvia, D. (2007). Using gaming technology to model and predict target movement. In *Proceedings of the 2007 spring simulation multiconference-Volume 3*, pages 283–288.

Gibney, E. et al. (2015). Deepmind algorithm beats people at classic video games. *Nature*, 518(7540):465–466.

Heinimaki, T. J. and Vanhatupa, J.-M. (2013). Implementing artificial intelligence: a generic approach with software support. *Proceedings of the Estonian Academy of Sciences*, 62(1).

Iovino, M., Scukins, E., Styrud, J., Ögren, P., and Smith, C. (2022). A survey of behavior trees in robotics and ai. *Robotics and Autonomous Systems*, 154:104096.

Jagdale, D. (2021). Finite state machine in game development. *algorithms*, 10(1).

Järvinen, A. (2002). Halo and the anatomy of the fps. *Game Studies*, 2(1):641–661.

Laird, J. E. (2002). Research in human-level ai using computer games. *Communications of the ACM*, 45(1):32–35.

Lee, M. (2014). An artificial intelligence evaluation on fsm-based game npc. *Journal of Korea Game Society*, 14(5):127–136.

Lee, M.-J. (2016). A proposal on game engine behavior tree. *Journal of Digital Convergence*, 14(8):415–421.

Livingstone, D. (2006). Turing's test and believable ai in games. *Computers in Entertainment (CIE)*, 4(1):6–es.

Long, E. (2007). Enhanced npc behaviour using goal oriented action planning. *Master's Thesis, School of Computing and Advanced Technologies, University of Abertay Dundee, Dundee, UK.*

Loyall, A. B. (1997). Believable agents: Building interactive personalities. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE.

Lucas, S. M. (2009). Computational intelligence and ai in games: a new ieee transactions. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(1):1–3.

Millington, I. and Funge, J. (2018). *Artificial intelligence for games.* CRC Press.

News, B. D. A. (2021). History of ai use in video game design. Website. last checked: 26.11.2021.

Orkin, J. (2004). Symbolic representation of game world state: Toward real-time planning in games. In *Proceedings of the AAAI Workshop on Challenges in Game Artificial Intelligence*, volume 5, pages 26–30.

Orkin, J. (2005). Agent architecture considerations for real-time planning in games. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 1, pages 105–110.

Salles, A., Bjaalie, J. G., Evers, K., Farisco, M., Fothergill, B. T., Guerrero, M., Maslen, H., Muller, J., Prescott, T., Stahl, B. C., et al. (2019). The human brain project: responsible brain research for the benefit of society. *Neuron*, 101(3):380–384.

Sanders, A. (2016). *An introduction to Unreal engine 4.* AK Peters/CRC Press.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al. (2017). Mastering the game of go without human knowledge. *nature*, 550(7676):354–359.

Stenros, J. (2017). The game definition game: A review. *Games and culture*, 12(6):499–520.

Vassos, S. (2013). Introduction to strips planning and applications in video-games.

Warpefelt, H. (2016). *The Non-Player Character: Exploring the believability of NPC presentation and behavior*. PhD thesis, Department of Computer and Systems Sciences, Stockholm University.

Warpefelt, H., Johansson, M., and Verhagen, H. (2013). Analyzing the believability of game character behavior using the game agent matrix. In *DiGRA Conference*. Citeseer.

Yildirim, S. and Stene, S. B. (2010). *A survey on the need and use of AI in game agents*. InTech.