Study and analysis of behaviour decision methods of non-player characters in first-person shooters

André Miguel Mana Pereira

Projeto para obtenção do Grau de Mestre em
Design e Desenvolvimento de Jogos Digitais
2º ciclo de estudos

Orientador: Prof. Bruno Miguel Correia da Silva

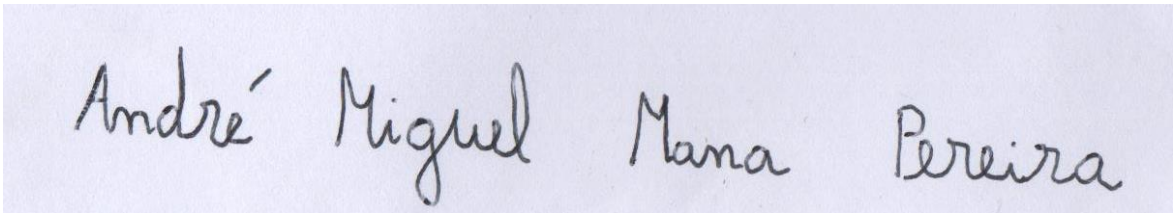
Outubro de 2022

Declaração de Integridade

Eu, André Miguel Mana Pereira, que abaixo assino, estudante com o número de inscrição M10305 do curso Design e Desenvolvimento de Jogos Digitais da Faculdade de Artes e Letras declaro ter desenvolvido o presente trabalho e elaborado o presente texto em total consonância com o **Código de Integridades da Universidade da Beira Interior**.

Mais concretamente afirmo não ter incorrido em qualquer das variedades de Fraude Académica, e que aqui declaro conhecer, que em particular atendi à exigida referenciação de frases, extratos, imagens e outras formas de trabalho intelectual, e assumindo assim na íntegra as responsabilidades da autoria.

Universidade da Beira Interior, Covilhã 28/10/2022



Acknowledgments

The conclusion of this project as well as my academic life would not be possible without the help of my friends and family. I would like to thank my advisor Professor Bruno Miguel Correia da Silva for his guidance. This project was realised in the investigation Lab SINS (Secure and Intelligent Networked Systems).

Resumo

Personagens não jogáveis (NPCs) são um dos tópicos mais importantes dos videojogos, pois é graças a eles que os jogos se tornam mais divertidos e menos repetitivos.
Com o aumento do realismo e complexidade dos videojogos, é necessário que o comportamento dos NPCs se torne também mais realista. Para resolver esse problema, vários métodos de decisão para NPC foram criados. Jogos de tiro em primeira pessoa (FPSs) são responsáveis por serem os pioneiros em técnicas tais como behaviour trees e goal oriented action planning que são agora utilizados em vários géneros de videojogos como métodos de decisão de NPC.
Alguns métodos de decisão são mais apropriados do que outros, dependendo do tipo de comportamento que pretendamos que o NPC exiba.
É proposto neste projeto, analisar, comparar e implementar diferentes métodos de decisão de NPCs.

Palavras-chave

Videojogos; inteligência artificial; métodos de decisão; jogos de tiro em primeira pessoa; máquina de estados finita; behaviour trees; goal oriented action planning; algoritmos genéticos; machine learning

Resumo Alargado

Em 1956, John McCarthy inventou o termo inteligência artificial para descrever que inteligência, ou o processo de aprendizagem, podem ser formalizados de tal forma que uma máquina os consiga interpretar e simular.

Há diferenças entre inteligência artificial dos videojogos e inteligência artificial académica, mas existe um consenso que quando uma personagem toma decisões próprias, essa personagem é considerada inteligente.

Na década de 1970, os vídeos jogos começaram a tornar-se mais populares e são atualmente parte da cultura moderna. Com o aumento do realismo e complexidade dos videojogos, é necessário que o comportamento dos NPCs se torne também mais realista de modo a não quebrar a experiência do jogador.

NPCs, são personagens que não são controladas por jogadores, mas pela inteligência artificial do jogo. Estas personagens têm vários propósitos tais como serem companheiros do jogador, substituição de oponentes humanos ou inimigos. A inteligência de um NPCs pode ser estruturada em perceção, decisão e ação.

- Perceção corresponde como o NPC recebe inputs do mundo a sua volta (exemplo: audição e visão).
- Ação corresponde ao comportamento que o NPC exibe (exemplo: disparar, patrulhar).
- Decisão corresponde à forma que o NPC escolhe qual ação tomar para atingir os seus objetivos, baseando-se no conhecimento atual sobre o mundo ao seu redor (exemplo: máquina finita de estados e behaviour tree).

FPSs têm feito grandes contribuições para a inteligência artificial dos videojogos. Alguns exemplos são Golden Eye 007 (1997) e Half-Life (1998) que ajudaram a solidificar o género FPS, através dos comportamentos inovadores e criativos dos NPCs que usavam máquinas finitas de estados, HALO 2 (2004) por introduzir a behaviour tree e F.E.A.R. (2005) por ser o primeiro videojogo a utilizar goal oriented action planning. É importante realçar que uma boa inteligência artificial não significa a mais complexa e difícil de jogar contra. A ilusão de inteligência é mais importante do que a inteligência real. Não vale a pena construir um sistema de inteligência artificial complexo e avançado se o jogador não o notar.

Os objetivos principais do projeto são demonstrar a importância dos FPSs e despertar interesse para a criação de novos métodos de decisão ou melhoramento de métodos existentes. Para atingir os objetivos do projeto vários capítulos principais foram formulados. O capítulo **Background** explica o conhecimento teórico sobre os tópicos

implementados neste projeto, tais como máquinas finitas de estados, behaviour trees, goal oriented action planning, algoritmos genéticos e machine learning. O capítulo **Decision methods applied** descreve os NPCs implementados no projeto e os seus respetivos métodos de decisão.

O capítulo **Results and discussion** compara os métodos de decisão entre si para compreender quais as vantagens e desvantagens das suas implementações. Os resultados apresentados são baseados em observações feitas na criação dos NPCs e respetivos métodos de decisão.

Abstract

Non-player characters (NPCs) have a big importance in video games because if they did not exist, games would feel monotone and without life. With the increase of complexity and realism in video games graphics, the behaviour of NPCs needs to keep up to not break the experience of the player. For that reason, new decision methods for NPCs are studied to handle complex behaviours. First person shooters (FPSs) have a big role on implementing novel ways to define behaviour decision methods of NPCs such as behaviour trees and goal-oriented action planning while other game genres end up using their standards. Some decision methods are better than others depending on what kind of behaviour we want the NPCs to possess, thus, we propose to analyse, discuss, and compare different behaviour decision methods of NPCs and implement some examples to showcase these algorithms.

Keywords

Videogame; artificial intelligence; decision methods; first person shooter; finite state machine; behaviour tree; goal oriented action planning; genetic algorithm; machine learning

Content

List of figures

List of tables

Acronyms

NPC	Non-player character
FPS	First-person shooter
FSM	Finite state machine
HFSM	Hierarchical finite state machine
GOAP	Goal oriented action planning

Chapter 1

Introduction

1.1 Project Focus and Scope

In 1956, John McCarthy coined the term artificial intelligence to describe that intelligence, or the process of learning, can be defined in a way that a machine can simulate it.
It is important noticing that video game artificial intelligence has some differences from academic artificial intelligence, but most agree when a character in a video game presents reasoning and appears to make decisions by itself, it is considered true artificial intelligence.
Video games started becoming mainstream in mid-1970s and are now part of the modern culture. With the increase of complexity and realism in video games graphics, the behaviour of video game characters needs to keep up to not break the experience of the player.
NPCs are video game characters that are not controlled by the player but by artificial intelligence. These characters have various purposes such as to be used as a partner to the player, replacement of human opponents or to be used as enemies. We can structure the intelligence of an NPC by sensing, thinking, and acting.

- Sensing is how the NPC perceives the surrounding environment (example: hearing, vision).
- Acting is the behaviour that the NPC presents after the thinking process (example: shoot, move to a location, play animations).
- Thinking is how the NPC decides what action to take to achieve its goals given the current situation and knowledge of the environment (example: behaviour tree, finite state machine).

FPSs have made big contributions towards video games artificial intelligence of NPCs. Some examples are Golden Eye 007 (1997) and Half-Life (1998) which helped to solidify the FPS genre with its innovative NPC behaviours using Finite State Machines, HALO 2 (2005) for introducing the behaviour trees and F.E.A.R. (2005) for being the first video games using goal oriented action planning. It is important to notice that a good NPC's artificial intelligence does not mean the most complex and difficult to play against. The illusion of intelligence is more important than actual intelligence. There is no point to

build a complex and advanced artificial intelligence system if the player will not notice that.

1.2 Motivation and objectives

The main goals of the project are to show the importance of FPSs to the video game industry as well as how they were the pioneers in most of the decision method techniques used in all genres, interest others to the FPS genre and to encourage people to improve or invent new decision method techniques. To achieve these main goals, the partial objectives were defined:

- **Review of the state of the art** to gather information on different decision methods, such as finite state machines, behavior trees, goal oriented action planning, genetic algorithms and machine learning.
- **Design and implementation** of NPCs and its decision methods based on the existing techniques in the video game industry.
- **Results and discussion** to compare the decision methods with each other to understand the advantages and disadvantages of their implementations. The results presented are based on observations made in the creation of NPCs and their decision methods.

1.3 Document organization

This document has the following chapters:

1. Chapter 1 - Introduction: Intends to present the project, its motivations, objectives and showcase the document structure.
2. Chapter 2 - Background: This chapter explains the theoretical knowledge on the topics covered in this project and the game industry state of art methods that were used.
3. Chapter 3 - Decision methods applied: This chapter describes the technologies used, the NPCs implemented and their respective decision-making methods.
4. Chapter 4 – Results and discussion: In this chapter we compare the decision methods implemented in this project with each other to see when they are recommended to be used.
5. Chapter 5 - Conclusion and Future work: In this chapter we reflect on the main takeaways from this work and what can be added to continue this project.

Chapter 2

Background

In this chapter we are going to present the decision methods that already exists in the video game industry.

2.1 Finite state machine

Although the finite state machine (FSM) was not introduced by a first-person shooter and already was being applied to video games such as Pac-Man (1980), it was with video games like Half-Life (1998) and GoldenEye 007 (1997) that the true potential of this technique was shown. Some first-person shooters that use this technique are Half-Life, Doom (1993 and 2016), GoldenEye 007, Quake and Duke Nukem 3D.
A FSM is a model with a finite number of states in which a state represents a specific action that a character can execute, for example, move, attack, idle, follow another character or die. The FSM hold the current state until it receives an input (condition) that will make the system transit to other state. It is the designer of the FSM that decides what conditions are needed to be satisfied to transit from one state to another. Finite state machines are commonly represented by graphs (see Figure 1) in which the nodes of the graph (circles) are the states of a FSM and the edges (arrows) indicate the condition that need to be satisfied to transit from one state to another. In Figure 1 we can see that to transit from the STATE A to the STATE B it needs to satisfy the condition 1. For example, if the NPC wants to change its action from patrolling an area (STATE A) to chasing the player (STATE B) it needs to detect the player (condition 1), so, the conditions are just a simple if statement that when satisfied change the behaviour of the NPC.

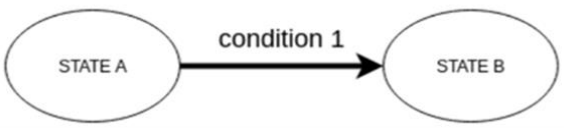


Figure 1: Finite state machine of two states

There are other ways to represent finite state machines such as transition table (see Table 1). The table contain three columns in which the first represents the current state the NPC is in, the second represent the condition that needs to be satisfied to transit to the next state and the third represents the resulting state of taking a condition in the current state.

Table 1: Transition table

Current state	Condition	Resulting state
STATE A	condition 1	STATE B
STATE B	-	-

We will look at two examples of FSMs. First, we are going to look at the guard enemy of the game Wolfenstein 3D, since Wolfenstein 3D is considered the first FPS that introduced the standard mechanics of modern FPS such as, shooting at enemies, switch weapon, and gain health and ammunition through boxes. Then, we are going to look at the behaviours of the enemy called Imp of the game Doom. After that we are going to discuss about hierarchical finite state machine, a variant of finite state machine.

2.1.1 Wolfenstein 3D – guard's finite state machine

By looking at the source code of Wolfenstein [1] we can represent the guard's finite state machine as the following graph (see Figure 2) and table (see Table 2).
The guard starts his behaviour on IDLE and only transit to the ALERT state when he detects the player. He can detect the player by sight, hearing or if it is too close. When he detects the player, he will transit to the ALERT state and will start chasing the player until it has a line of sight to shoot. If he has a line of sight to shoot, he will transit to the SHOOT state, otherwise he continues chasing the player indefinitely.
PAIN and DIE are states that are triggered by an external input. PAIN is triggered after the guard is attacked by the player. After the pain's animation ends, the agent transits to the ALERT state. DIE state is triggered after the guard loses all of its life.

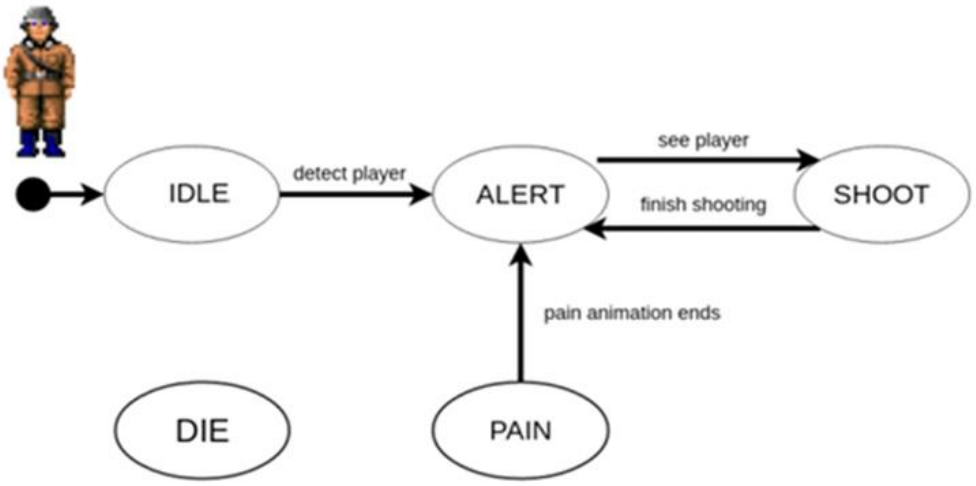


Figure 2: Guard's finite state machine

Table 2: Guard's transition table

Current state	Condition	Resulting state
IDLE	detect player	ALERT
ALERT	see player	SHOOT
SHOOT	finish shooting	ALERT
PAIN	pain animation ends	ALERT
DIE	-	-

2.1.2 Doom 1993 – Imp's finite state machine

By looking at the source code of Doom [2] we can represent the Imp's finite state machine as the following graph (see Figure 3).

The monster starts on the IDLE state, and only changes to the ALERT state after detecting a target. A target can be the player or other monster, due to monsters' possibility to fight with each other, known as monster infighting. The monster can detect its target by sight, hearing or if it is too close. In case a detection is triggered, the Imp will change to the ALERT state and will start chasing its target until it has a line of sight to attack. When he has a line of sight, he will transit to the DISTANT ATTACK state or MELEE ATTACK state, depending on the distance to the target, otherwise he will continue chasing. (ALERT state). After killing its target, if it has no more targets to kill, it will return to IDLE state.

DIE and PAIN are triggered by external inputs. PAIN is triggered after the Imp is attacked by an external trigger, such as the player's projectile. DIE state is triggered after the monster loses all its life.

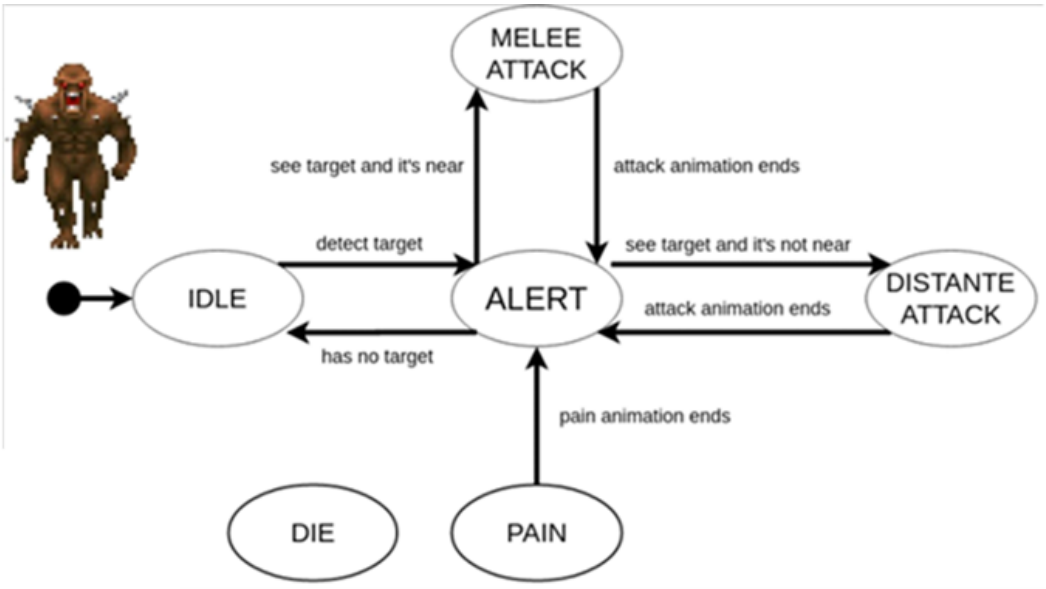


Figure 3: Imp's finite state machine

2.1.3 Hierarchical finite state machine

In the above examples, the NPCs had a small number of states. When NPCs starts to have more actions, the number of states will increase and consequently the number of conditions will increase too, so, the complexity of the system will get bigger, and because of that it will be more difficult to manage it. Hierarchical finite state machine (HFSM) is a variant of FSM that can help to alleviate the complexity by reducing the number of transitions.

HFSM introduces the concept of super states. A super state is a state that contain inside it a group of states. States are grouped inside a super state when they all have the same outgoing transitions. This can help to reduce redundant transitions because if there are states with the same outgoing transition, we can reduce the number of transitions by grouping all together in a super state [3]. In the Figure 4 we can see a representation of a HFSM. In HFSM, if we want to modify a state, we do not need to worry about breaking the whole system because we only need to focus on the super state.

In the Figure 5, the states inside the super states, were hidden for a better visual understanding. This means that is now possible to focus on the higher level of the FSM.

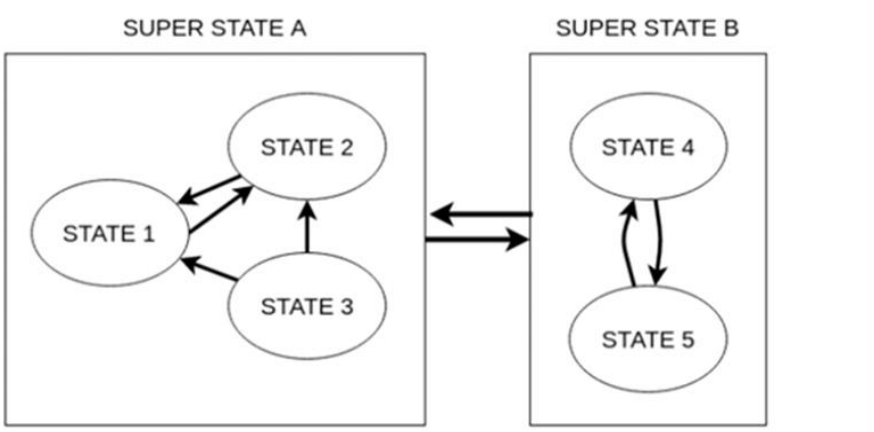


Figure 4: Hierarchical finite state machine representation

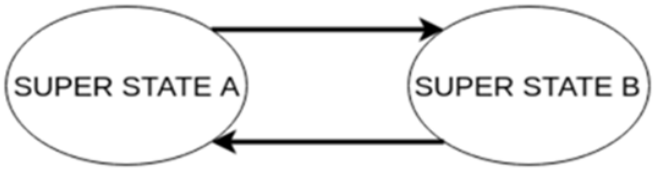


Figure 5: High level hierarchical finite state machine representation

2.2 Behaviour tree

Behaviour Tree was first introduced in the video game HALO 2. Since then, a vast genre of games started using this idea. Examples of video games that use behaviour trees are Halo 2, Halo 3, Far Cry Primal and Alien: Isolation.

Behaviour trees follow the same rules as trees in computer science. The execution of the tree begins in the root node and flows through the other nodes, from up to down and left to right. Each node is reported to its parent, until it reaches the root node of the tree. The report tells if the node has succeeded, failed or if it stills running a particular rule. The behaviour tree has two main types of nodes: leaves and compositors. In the Figure 6 we can see a representation of a behaviour tree.

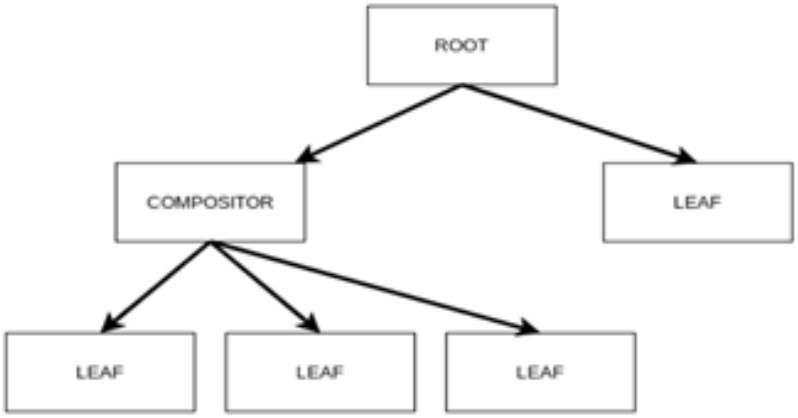


Figure 6: Example of a behaviour tree

2.2.1 Leaf

Leaf is a node that is in the extremity of the tree, so, leaf nodes cannot have children, only parents. A leaf represents the actions that the NPC can take, for example, walk to a position, flee, seek or attack (see Figure 7). A leaf can also represent conditions, that means that conditions will check if something needs to be satisfied before the NPC take an action, such as the leaf 1 of the Figure 7. The parent of a leaf can be of the type compositors, either sequences or selectors.

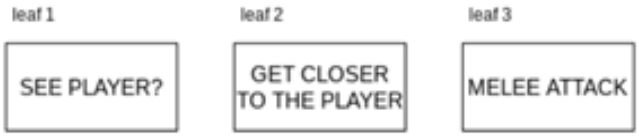


Figure 7: Example of leaf nodes

2.2.2 Compositor

Compositor is a node that groups other nodes together, that means, compositors can have one or more children. The children can be of any type of node, including other compositors. They are useful to gather leave nodes together since isolated actions would not make any sense and would make a NPC useless. There are two types of compositors: sequences and selectors.

2.2.2.1 Sequence

Sequence executes its children's nodes in sequence, one after another. All the children of a sequence need to be successful for a sequence to succeed. If one of the children fails to execute its rule, the sequence fails immediately too. In Figure 8 we can see an example of a sequence.

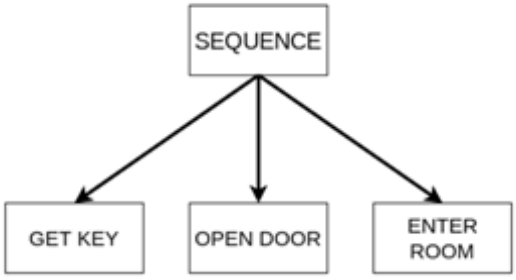


Figure 8: Example of a sequence node

The NPC can only enter the room if it has a key and has opened the door. If the leaf GET KEY fails, it will report failure to the SEQUENCE node. SEQUENCE only succeeds if the actions GET KEY, OPEN DOOR, and ENTER ROOM all succeed.

2.2.2.2 Selector

Selector execute its children in sequence, one after another, and succeed immediately when one of its children succeed too. If a child reports that has failed, the selector will try the next node and only fails when all the children have reported that have failed. In Figure 9 we can see an example of a selector.

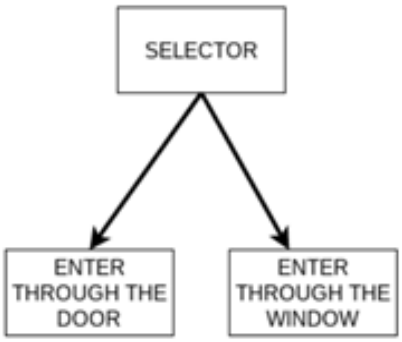


Figure 9: Example of a selector

In the example above the NPC wants to enter a room. The NPC can enter the room through the door or through the window. First it will attempt to enter through the door, and if it has succeeded, will report success to the selector. SELECTOR will succeed too and will not try the next child.
If the NPC cannot enter through the door, the selector will not report failure but will try the next node and attempt to enter through the window. The SELECTOR only fails when the NPC cannot enter through the window, that is, when all the children have reported failure.

2.2.3 Blackboard

Blackboards (see Figure 10) is a data set that contains information about the game world such as player position, distance to other objects and ammunition remaining. Using a blackboard has the benefit to share information between nodes, avoiding repeating the same calculations in different nodes.

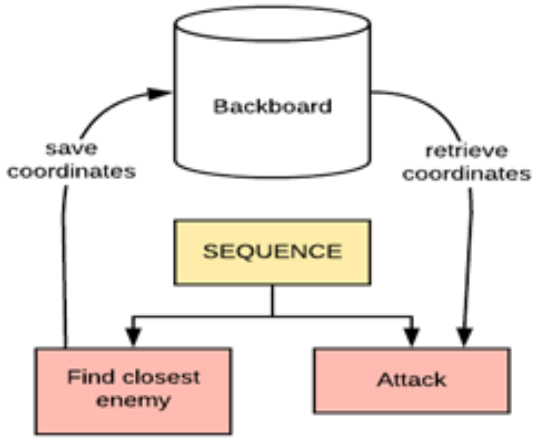


Figure 10: Blackboard [4]

2.2.4 Example of behaviour tree

The behaviour tree (see Figure 11) shows an example of a NPC that wants to enter a room.

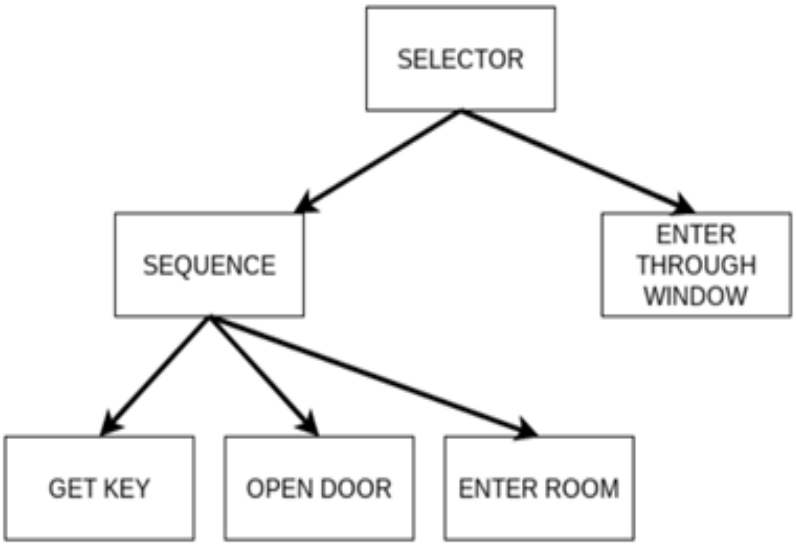


Figure 11: Example of a behaviour tree

The flow of the tree goes from up to down and left to right. It starts from the root (SELECTOR) and goes to the first child (SEQUENCE). The SEQUENCE tells the NPC to get a key, open the door and enter the room, in that specific order. If one of these actions fails, the SEQUENCE will report failure to the root node. Since the root node is a SELECTOR, will then execute the other child and enter the room through the window.

2.3 Goal oriented action planning

Goal oriented action planning (GOAP) was first introduced in the video game F.E.A.R: First Encounter Assault Recon.

Since then, a vast genre of games started using this idea. Examples of first-person shooters that used the GOAP system are F.E.A.R: First Encounter Assault Recon, F.E.A.R. 2: Project Origin and S.T.A.L.K.E.R.: Shadow of Chernobyl [5].

GOAP is a system that allows the NPCs to plan a sequence of actions to satisfy a particular goal [6]. As opposite of FSM, where the actions are connected to each through conditions that are pre-written by the designer of the video game, in GOAP, these actions are decoupled from each other before the run time, and connect during runtime (see Figure 12), depending on the goal that the NPC wants to achieve. The GOAP have three main components: goals, actions, and planner.

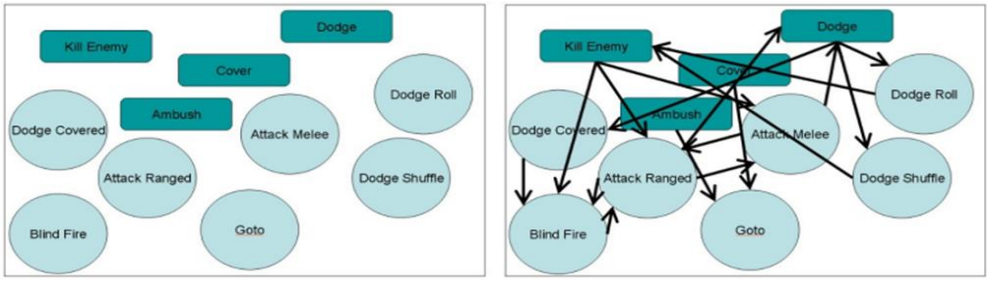


Figure 12: GOAP vs FSM [7]

2.3.1 Goals

Each NPC has a purpose. This means that the designer of the game gave a set of goals that the NPC should achieve, for example, survive and kill. The goals are in a list priority and are constantly being checked in order. When a goal cannot be pursued, the system tries the next in the list. For example, if the goal is killing an enemy, and the NPC does not know where the enemy is, the goal is no longer pursued, and it will try to execute the next in the list. The goals are achieved through actions. In the Figure 13 we can see the goals of the solider NPCs of the game F.E.A.R.

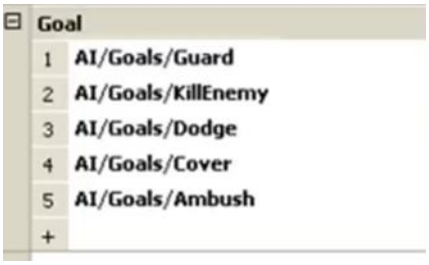


Figure 13: Goals of the soldier NPC [7]

2.3.2 Actions

Actions are isolated behaviours that a NPC execute such as shooting a gun, chase a player or patrol. In the Figure 14 we can see an example of actions that the soldier of the game F.E.A.R. can take.

In the GOAP system each action contains preconditions and effects. Preconditions are facts that need to be met for an action to be executed, for example, for an NPC to take the action FIRE it needs to meet the precondition hasAmmunition== true (see Figure 15).

Effect is how the world state has changed by taking an action. The world state is a collection of variables about the game environment such as position of a target, health of enemies or ammunition available.
Every action has a cost associated to help the GOAP system decide what action to choose when multiple actions have the same preconditions/effects. Cheaper actions are chosen over expensive ones. To reach the desired goal from the present world state the GOAP system must build a sequence of actions.

Figure 14: Actions of the soldier NPC [7]

Figure 15: Example of an action

2.3.3 Planner

The planner is a system responsible for constantly checking the list of goals and find a valid one to create a sequence of actions to achieve that goal.
Since the variables of the world are always changing, the planner can create a different sequence each time a goal is presented. This can add a dynamic behaviour to the NPC, allowing it to react differently to the same situation.
To build the sequence of actions, the planner uses a pathfinding algorithm, such as A*, to create a path from the desired world state to the current world state, connecting the actions in a sequence through preconditions and effects forming a chain of actions (see Figure 16).

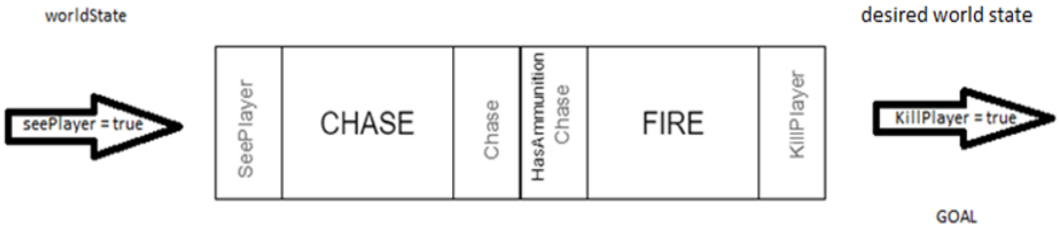


Figure 16: Example of a sequence of actions to achieve a goal

The planner creates the sequence of actions by adding the costs of the actions of the sequence. If it founds a sequence with a lower total cost, will choose that sequence instead of one with a higher cost.

The planning works backwards from the goal to the actual world state to see if there is a sequence of actions to achieve the goal. The planner begins by finding an action that satisfy the goal, then find an action that satisfy the previous action and repeat this process until the current world-state is matched.

The planner is constantly re-evaluating the priority of actions and goals, allowing the NPC to have long-term thinking and not react to triggers impulses like FSMs.

In the Figure 17 we can see a scheme that resumes the GOAP system.

The goals, world state and actions are supplied by the NPC, and with this information the planner creates a sequence of actions to decide which are the best to execute.

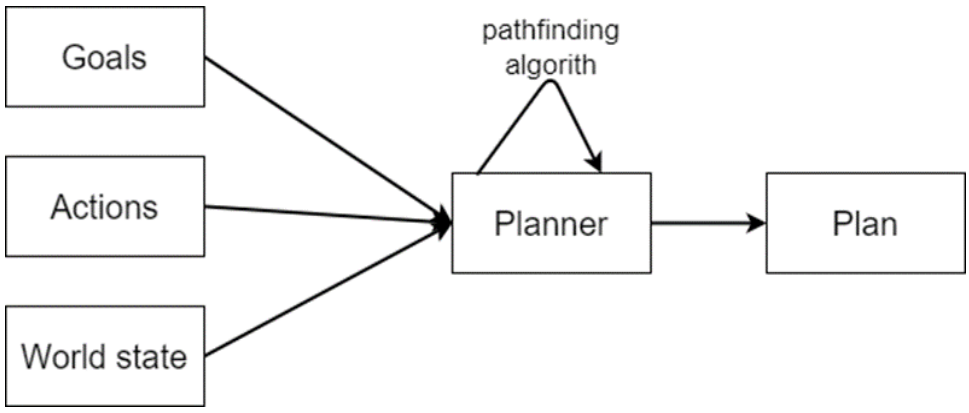


Figure 17: GOAP system shortened

2.4 Genetic algorithm

Genetic algorithm is an algorithm based on natural selection and evolution [8].
It was designed to search for solutions in search spaces that are poorly understood, where traditional techniques fail due to lack of knowledge [8].
Genetic algorithms are used to solve nonlinear problems such as optimization, automatic programming, machine learning and economics [9].
There is not much information about the use of genetic algorithms in FPSs except for some academic research [8][9][10][11] that were done in the past. Still for this being an unknown algorithm in FPS, there is in fact a positive view because it opens room for research and experimentations. Genetic algorithm can be used as a decision method by itself but there are virtually no examples of that. They are used to complement decisions method such as FSM , by automatically tuning some values without the need of a person [10]. In the figure 18 we can see a chart flow of a genetic algorithm. Now, we are going to discuss each component of the chart flow and learn how a genetic algorithm works.

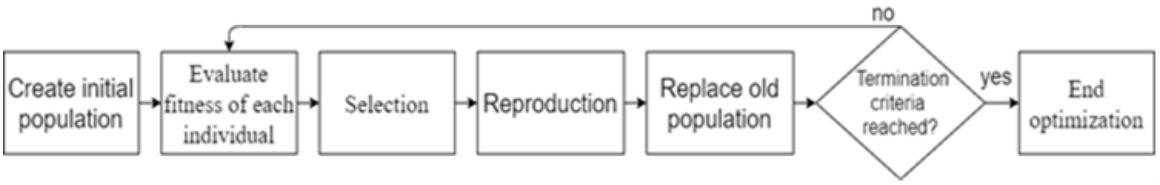


Figure 18: Diagram of a genetic algorithm

Create initial population

To get an optimal solution to a given problem we need to have a population with enough individuals in which everyone has the potential to solve the underlying problem. Everyone within the population has a virtual “DNA” (can be represented as an array) that contain a set of properties (genes) (can be the elements of the array) that determine the traits of the NPC [12]. Genes have an arbitrarily nature since they can represent anything, e.g. from move speed to weapon preference. To have enough variety to evolve towards the optimal solution, the population is typically created randomly to allow a variety of individuals with different gene values.

Evaluate fitness of each individual

To determine which individuals of the population, have a closer solution to the problem, all the individuals are evaluated with a fitness function. This function is problem specific and typically the function will produce a numeric score that will represent the fitness of a

given member of the population [12]. The success of the genetic algorithm is dependent on the selection of a proper fitness function [11].

Selection

After the fitness function is calculated, the most fit individuals will be selected to breed a new generation. This can be done through crossover or mutation.

Reproduction

To create the next generation of individuals we can crossover or mutate individuals that were selected to do so. Crossover means that parents, are breeding a new individual with the genetic code of the parents. The resulting offspring has a combination of traits from it parents. One way to crossover is by picking up the first half genes from the first parent and the last half genes from the second parent. Mutation means that a single individual will breed a new offspring with the same values of its progenitor DNA but with some gene altered randomly. This can add variety to the evolutionary process itself and prevents stagnation and premature convergence [11].

Population replacement

After the reproduction process, a new generation is created, typically fitter than the previous one. The old generation is now removed from the population since a fitter population exists. The process of evaluation, selection, reproduction, and population replacement is repeated until a termination criterion is reached.

Termination

The most common criteria to stop the process are when a certain number of generations are reached, the time we set ends or the solution to the problem was found. The best genetic code from the final generation can then be saved to a file and then “injected” to the NPC.

2.4.1 Genetic algorithm in FPS

Numerous variables of the NPCs are parametrized with values that are set by the designer of the game. Choosing the correct values might not be an easy task, especially if the number of parameters is big. Changing one parameter will eventually negatively impact others because they might be non-linearly dependent [8]. Some of these values can influence decision methods such as FSMs For example, if the value that needs to be tuned

is the distance from the NPC to the player and this value is responsible for triggering the next state of the FSM, changing this value will change how the FSM work. In the paper [8] the authors use a genetic algorithm to tune some parameters of NPCs in the video game Counter-Strike such as weapon preference, initial aggressivity, path preference, and style of gameplay. The author used a fitness method that punishes friendly fire and rewards NPCs that earned more money during the game, only allowed the parents with the higher values on the fitness function (best genes) to breed and pass the genes to the next generation and create a new generation each 3 minutes (time that a round last). The authors conclude that by using a genetic algorithm, the NPCs that were tuned by the genetic algorithm can play as well as NPCs tuned by a human with expert knowledge about the game [8].

2.5 Machine learning

In the past decades, the term artificial intelligence has been used in video games to describe NPCs that look intelligent but in fact are just following a set of prewritten instructions. True artificial intelligence algorithms are recently being applied in video games and are not only helping evolve the game industry, but video games are being used as testing grounds of research in academic artificial intelligence.

Machine learning is a subfield of artificial intelligence that uses data and algorithms to mimic intelligent human behaviour, without explicitly being programmed. These systems try to mimic the way humans learn and solve problems, gradually improving its accuracy [13].

Some real-world uses of machine learning are object detection, self-driving cars, recommendation algorithms and medical imaging [13]. While humans can easily distinguish pictures of different faces, computers can't to it that easily. Writing a software in a regular programming way to distinguish different images might be very time-consuming or almost impossible, so machine learning allows the computers to learn in a similar way as humans learn, through experience [14].

In some cases, machine learning can even outperform humans such has in web searches, since a human couldn't search as fast as a computer.

Machine learning can be divided in three main subcategories:

Reinforcement learning: Uses trial and error to take the best action in a particular environment. Uses a reward and punishment system to give feedback and tell if an action taken by the agent was good or bad. In this way it helps the agent to learn what actions to

take and it learns from experience. Reinforcement learning is used in self-driving cars and video games.

Supervised learning: Uses labelled datasets to train the algorithm to classify data or predict results accurately. Supervised learning is used to describe objects in pictures or videos.

Unsupervised learning: Try to find hidden patterns in unlabelled datasets. It is used in customer segmentation.

Machine learning is also related to the field of neural networks. Artificial neural networks are based on the human brain as the nodes of an artificial neural network are connected to each other to simulate the connection of human neurons (see figure 19).

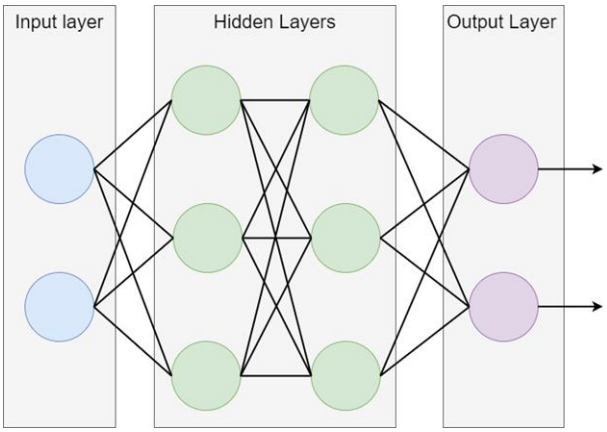


Figure 19: Artificial neural network

Artificial neural networks consist of an input layer, hidden layers, and output layer. Each artificial neuron, also called node, executes a mathematical function on the input signals to generate an output signal. The connections between nodes have an associated numeric value called weight and if the output of a node is above a specified threshold value, it is activated and forwards data to the next layer of the neural network to generate output signals [13].

The goal of the neural network is to learn the weights of the connections, over numerous iterations.

One example of a neural network is one that detect street signs. The neural network receives an image signal as the input and different layers will detect different parts of the image such as colour, geometry, and shapes. The developers of the artificial neural network don't control how the neurons organize their values. The organization comes from the automated learning process which will lead to the accurate detection of the street sign.

Typically, supervised, and unsupervised learning use neural networks, while regular reinforcement learning does not. But it is possible to mix reinforcement learning and neural networks such as in deep reinforcement learning.

2.5.1 Machine learning in first person shooters

Machine learning has been applied in first person shooters in numerous ways. Some examples are decision methods of NPCs, software that play video games by itself such as DOOM (1993) [15], and even software that helps players to cheat.
Tradition cheating methods such as aim-bot uses files from the game to get the positions of other players to point the gun automatically to them. These old methods can be detected because they manipulate the game files. New cheating methods take advantage of machine learning by capturing the screen (frames of the game), without manipulating any game files. Then, the algorithm can detect human-like silhouettes and automatically point and fire at them. To make this even less undetectable the program can run on a different computer with a capture card inputting data coming from the main PC [16].
Countermeasures can be taken by using machine learning to study player behaviour to spot cheaters unexpected changes of behaviour in real-time data [17].

2.5.1.1 Battelfield 1

As opposite to typical board games, where a finite number of possible outcomes exists, first person shooters have a huge number of possible actions such as run, crouch, fire and reload, that combined increases the possible outcomes, making the task of mapping all the actions and learning harder [18]. In an interview the technical director of Battelfield 1, Magnus Nordin, explained that machine learning was used to create behaviours of NPCs.
To train the NPCs, a combination of two methods were used, imitation learning and reinforcement learning. In the imitation learning, the software observed human playing and attempted to mimic them. After that, the NPCs started to train on its own, figuring out the rest of the game themselves, using reinforcement learning [19]. During the reinforcement learning it were given rewards to the NPCs for completing tasks such as killing enemies. [18] These rewards gave feedback to the NPCs allowing them to experiment through trial and error. This process was done hundreds of times and the NPCs improved over time. In the end, NPCs learned how to dodge bullets, teamwork, adjust their aim to gun recoil, and behave differently when have low ammo or low health [19][18].

Chapter 3

Decision methods applied

In this chapter we are going to present the technologies used, the NPCs implemented and their respective decision-making methods.

3.1 Used technologies

To implement the decision methods, a game world was created where the NPCs live, thus, a set of tools were needed. Above are described the tools used to develop this project.

3.1.1 Unity 2019.4.36f1

Unity is an engine used to create videogames for computer, consoles, mobiles, and virtual reality platforms. It contains crucial built-in features such as physics and 3D rendering. Unity comes with a default set of tools, but more can be added. Some of these tools are animation time-line editor, navigation mesh, audio editor, ragdolls, and terrain editor. Unity uses C# as the scripting language. It was used to create the game world as well implement the decision methods.

3.1.2 Notepad++

Notepad++ is a free source code editor. It was used to write the code of the project.

3.1.3 Audacity

Audacity is a free and open-source software used to record and edit audio. It was used to cut undesirable parts of sound effects and to create new sounds by joining multiple sound effects.

3.1.4 Gimp

Gimp is a free and open-source image editor. It was used to edit sprites such as user interface icons.

3.2 Implementation details

To implement the decision methods, an FPS environment was created with the help of free assets such as 3D models, animations, particle effects, music and sound effects found in the Unity's asset store.

Although the focus of the project is the decision methods of NPCs an extra content was implemented in the project such as a main menu, a real time cutscene and a pause menu that includes mouse sensibility. It was implemented a main character that the player can controls its movement, jumping and firing. The main character has health and three weapons: pistol, rifle, and knife. The guns mechanics includes reload, recoil, weapon sway and bullet shell drop effect. Each gun has its own properties such as damage, bullet capacity and sound effects. It is also possible to gain health through health boxes and get extra ammunition through ammunition clips and boxes. In the Figure 20 we can see a screenshot of the environment of the project.

Figure 20: FPS Environment

Six NPCs are going to be presented, each with its own decision method. As mentioned in the chapter 1, for a NPCs to decide what action to take, it is necessary first to gather information of its surroundings trough sensors and based on that information choose an

action to take. In the project, the sensors used were vision and hearing. There are multiple ways for a NPC to see a target. In the project it was used the angle between the forward vector of the NPC and the vector that points from the NPC to the target (see Figure 21). If that angle is lower than a threshold and the distance to the target is lower than a threshold, a raycast is sent from the NPC to the target and if there are no obstructions between them, that means that the NPC can see the target. This check is done constantly, in the update function of the game, between the NPC and all its enemies, to see if there are targets that the NPC can see.

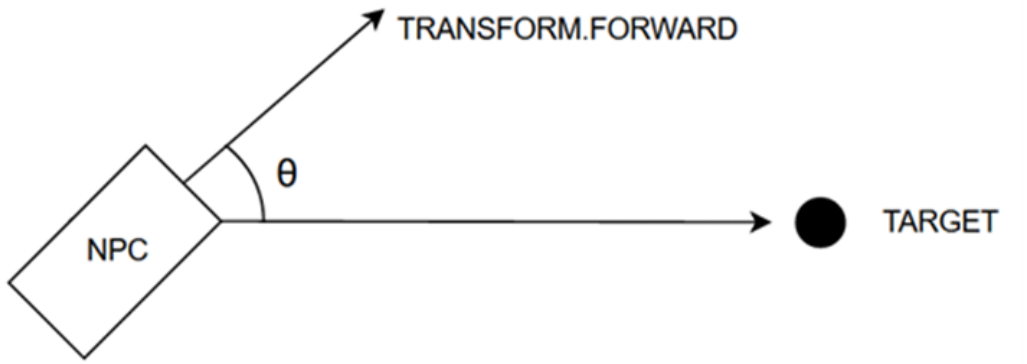


Figure 21: NPC's sight detection

For the NPC to listen a target it can just use the distance between itself and the source of the noise. This check is done constantly, in the update function of the game, between the NPC and all of its enemies, to see if someone that is near is making noise. For example, footsteps or gun shots sound effects that are being played by other NPC or player.
In the Figure 22 we can see a representation of the targets that can be heard by the NPC. The enemies that area inside the blue circles can be heard because the distance is lower than the threshold. The enemies that are outside the bigger circle (small white circles) cannot be heard because they are too far away from the NPC.

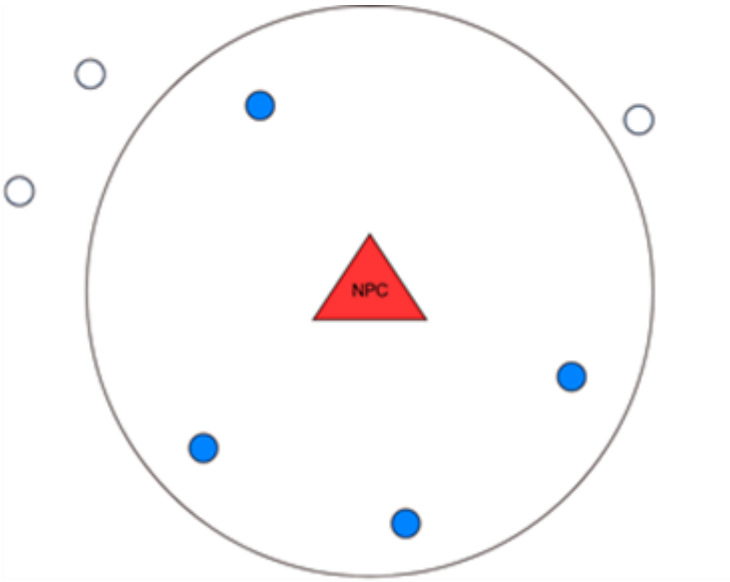


Figure 22: NPC earing detection representation

3.2.1 Spider robot – Finite state machine

The spider robot (see Figure 23) is an NPC that has the purpose to eliminate the player and Zombies NPCs. It uses a FSM as decision making process and can take five actions: patrol, fire, inspect, chase, and die.

Figure 23: Spider robot model

Each action that the spider can take is a state of the FSM. In the Figure 24 we can see a representation of the FSM. To change from one state to another it needs to perceive what is around itself through vision and hearing sensors.

As shown in the figure 24, the NPC starts in the PATROL state. In this state the spider moves through waypoints indefinitely until it detects a target. When it detects a target by sight, it will transit to the CHASE state and will start to follow its target. If it detects a target by hearing, it will transit to the INSPECT state and will start to inspect the source of the noise and if during the inspection finds a target, it will start chasing it, otherwise it will return to PATROL state. While in CHASE state, if the spider is near enough to the target to shoot, the spider will transit to the FIRE state and starts shooting. While shooting, if the NPC loses sight to the target, it will transit to the INSPECT state and start inspecting the zone to try to find more targets, and in case of finding none, it will return to the PATROL state. As opposite of other states, the DIE state is activated externally and not by the conditions of other states. The FSM only activates the DIE state when the NPC has no health.

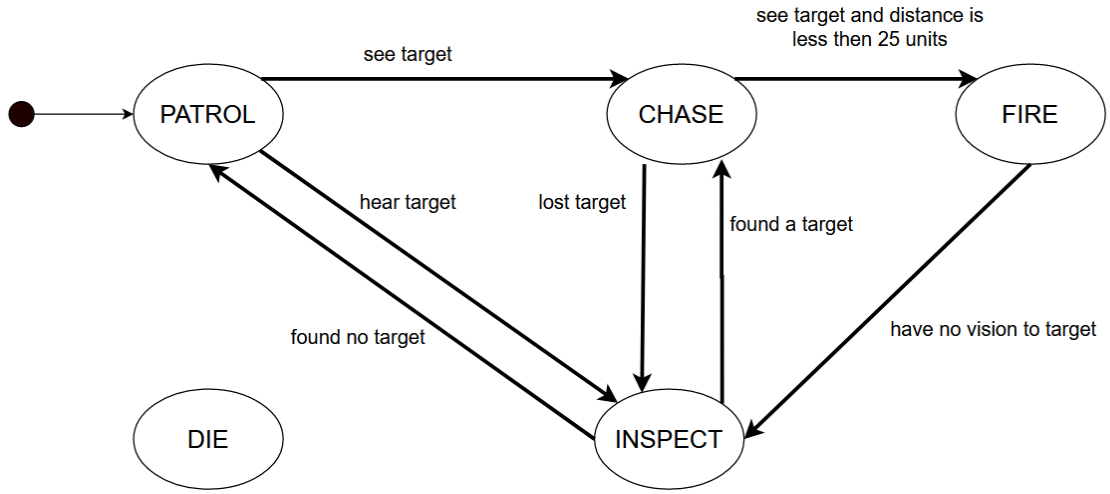


Figure 24: Spider's finite state machine

3.2.2 Soldier – Finite state machine

The soldier (see Figure 25) is an NPC that has the purpose to aid the player in his journey.

Figure 25: Soldier model

Soldier uses a FSM as decision making process and can take four actions: follow, attack, idle and die. Each action that the soldier can take is a state of the FSM. In the figure 26 we can see a representation of the FSM. To change from one state to another it needs to perceive what is around itself through vision sensors. The player can give an order to the soldier to stay in place (IDLE state) or to be followed by the soldier (FOLLOW state). While idling or following the player, if it sees an enemy, it will start shooting at it. After finishing killing the enemies in sight, it will return to the previous condition, either following or idling. The soldier also possesses a friendly fire mechanic. When the soldier is attacking an enemy and the player shoots at the soldier, the soldier will think that it was done accidentally and will forgive the player. If the players start shooting the soldier for no reason, the player becomes the new enemy of the soldier, and the soldier will start shooting at the player until he is dead. The DIE state is activated externally and not by the conditions inside states. The FSM only activates the DIE state when the NPC has no health. The soldier also has the capacity to reload its gun when it runs out of ammunition.

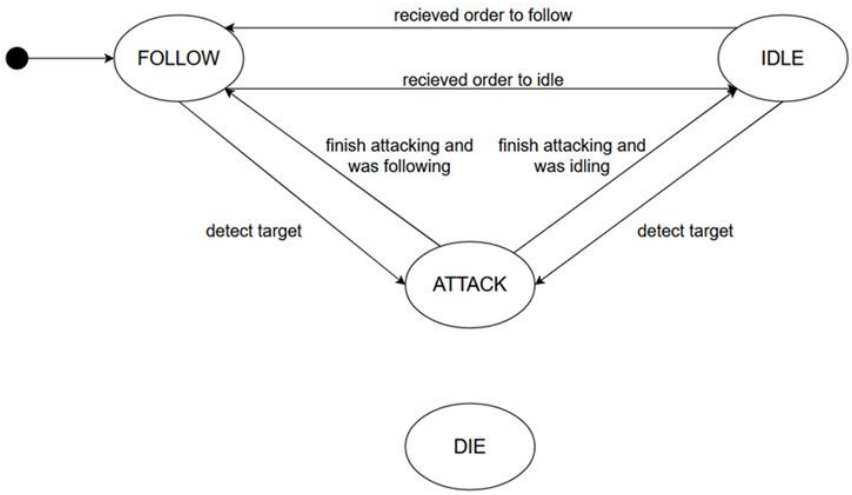


Figure 26: Soldier's finite state machine

3.2.3 Zombie – Behaviour tree

The zombie (see Figure 27) is an NPC that has the purpose to eliminate the player. The zombie uses a behaviour tree as decision making process and can take eight actions: run into cover, recover, chase, melee attack, inspect origin of the noise, get health, patrol, and die. The zombie perceives the environment through vision and hearing sensors.

Figure 27: Zombie model

In the figure 28 we can see a representation of the behaviour tree. Each action that the zombie can take is represented by the rectangular leaves of the behaviour tree. The DIE action is not represented in the behaviour tree because it's called externally. The tree is continuously being called in the update function of the game and when the zombie dies, it is deactivated.

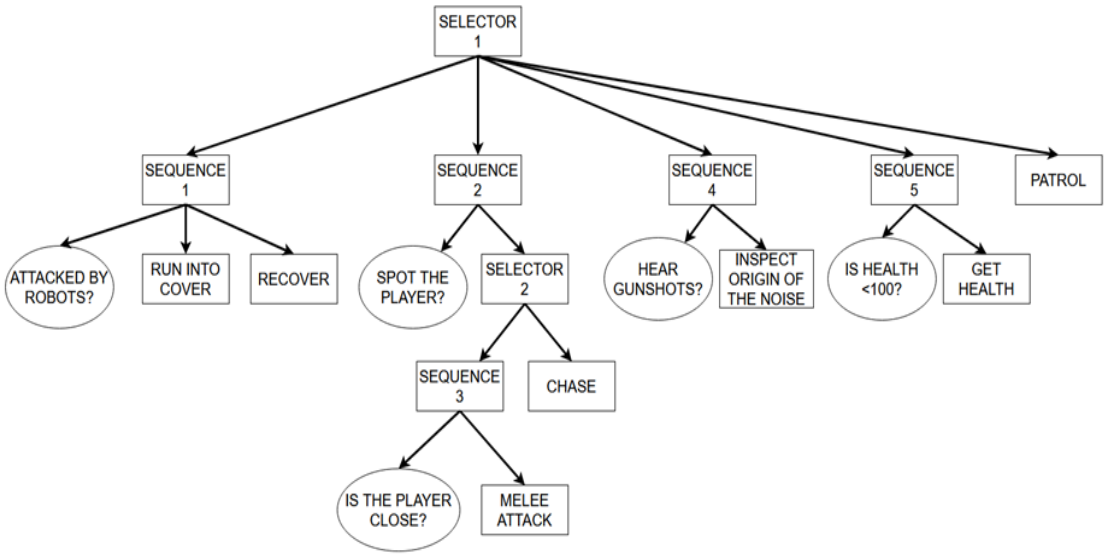


Figure 28: Zombie's behaviour tree

The flow of the tree goes from up to down and left to right. It starts from the root selector (SELECTOR1) and goes to the SEQUENCE1. The SEQUENCE1 tells the zombie that when attacked by robots, to run into cover and rest a little bit to avoid being killed since the zombies are afraid of robots and can't fight them. The first child of the SEQUENCE1 tests if the zombie is attacked by robots. If it's true, will go to the next leaf and run into cover. When finish doing the action RUN INTO COVER, will start the action RECOVER. After recovering, the SEQUENCE1 is completed and will return SUCESS to the root's tree. The SEQUENCE1 will return FAILURE to the root's tree in case the zombie is not attacked by robots. Only goes to the SEQUENCE2 when the SEQUENCE1 has failed, when there is no danger of being killed by a robot. The SEQUENCE2 tells the zombie that whenever it sees the player, if it is close to use the melee attack, and if too far, to chase the player. The first leaf of the SEQUENCE2 tests if the zombie can see the player. If so, the zombie will execute the SELECTOR2. SELECTOR2 will then run the SEQUENCE3 because it is its first child. When the player is too close, will do a melee attack and return SUCESS to the root. In case it is false, the SEQUENCE3 will return FAILURE to the SELECTOR2, and the SELECTOR2 will execute the second child, CHASE, and the zombie will start chasing the player. If the zombie has no sight to the player, the SEQUENCE2 will return FAILURE to the root and will try the SEQUENCE4. The SEQUENCE4 tells the zombie that whenever it hears gunshots, to go to the place where the noise come from and see if anyone is there. The first leaf of the SEQUENCE4 will check if the zombie has heard gunshots. In case true, will inspect the place where the gunshots came from. If the zombie doesn't hear any gunshots, will return FAILURE to the SEQUENCE4 and subsequently to the root, and

then try the SEQUENCE5. The SEQUENCE5 tells the zombie that whenever is health is lower than 100% to go and get some food to full the health. The first leaf of the SEQUENCE5 checks if the zombie doesn't have full health. In case it's true will move towards the place where food is. If the zombie already has full health the SEQUENCE4 will return FAILURE and try the next leaf, PATROL. The order of importance of the actions goes from left to right. This means that only will patrol whenever it's not attacked by robots, can't see, or hear its enemies and has full health.

3.2.4 Cyborg - Goal oriented action planning

Cyborg (see Figure 29) is an NPC that has the purpose to eliminate the player. It uses goal orient action planning as decision making process and to help creating the behaviours it was used a GOAP library [20].

Figure 29: Cyborg model

The goal of the cyborg is to kill the player and to achieve it, can take five actions, GET AMMUNITION, PATROL, CHASE, SWORD ATTACK and FIRE. In the Figure 30 are shown all the actions with preconditions and effects. The actions are not connected with each other, which make it easy to manage the complexity of the system. It is the GOAP's planner that build a sequence of actions to achieve the goal, so, the NPC does not react immediately to trigger conditions like in FSM but take in consideration what the goal is.

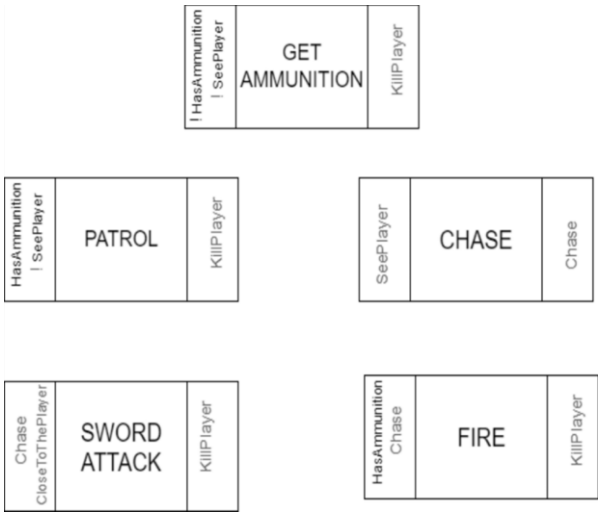


Figure 30: Cyborg's actions

When the goal is presented a sequence of actions is chosen based on the state of the agent and the world. The planning works backwards from the goal to the actual world state to see if there is a sequence of actions to achieve the goal. This is done by matching the effects with preconditions all the way back to the actual world state (see Figure 31).
The goal of the cyborg is to kill the player, so, the planner will first find actions that have an effect "KillPlayer". If there are multiple actions with the effect "KillPlayer", will find the one with the lower cost. It will then check the preconditions and see if there is some action that can be matched. This process is repeated all the way back to the current world state.

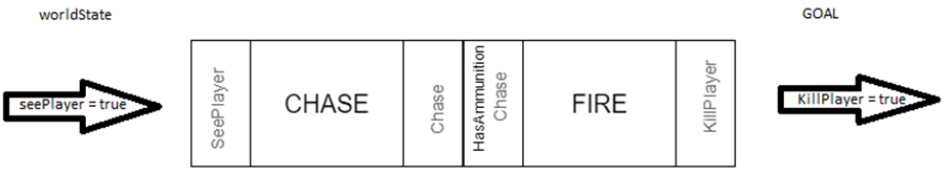


Figure 31: Example of a sequence of actions to achieve a goal

3.2.5 Turret - Genetic algorithm

Turret (see Figure 32) is an NPC that has the purpose to eliminate the player and Zombies. It uses a genetic algorithm that indicates to the turret each time it sees the player or a zombie, to start shooting at them, but when it detects robots such as cyborgs or spider robots, to ignore them and don't shoot. The turret perceives what it is around itself through a vision sensor.

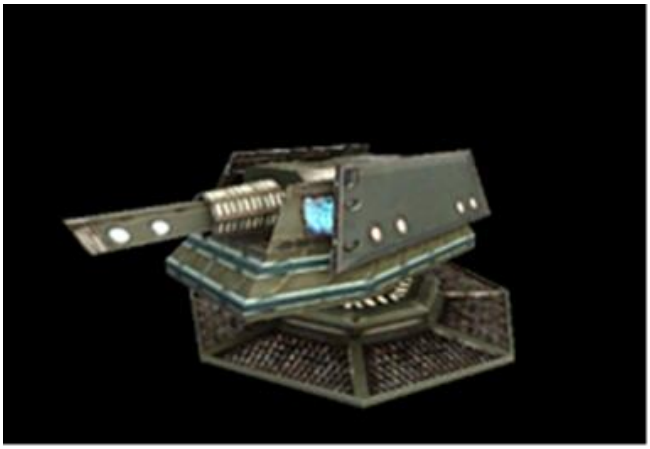


Figure 32: Turret model

To came up with the optimal genotype, that is, only kill zombies and player, a population of 25 turrets were created where everyone contains random gene values.
The possible genes are kill player, kill zombie, kill robots, or kill nothing.
Here are some examples of possible “DNAs” for turrets:

[kill player, kill zombies, _]
[kill player, _, kill robots]
[kill player, kill zombies, kill robots]
[_,_,_]
[_, kill zombies, _]

To get a turret with the optimal genes, 25 turrets were placed in a training environment, where they were separated by walls and in front of them, they had one robot, one zombie and the player’s character to shoot at (see Figure 33).

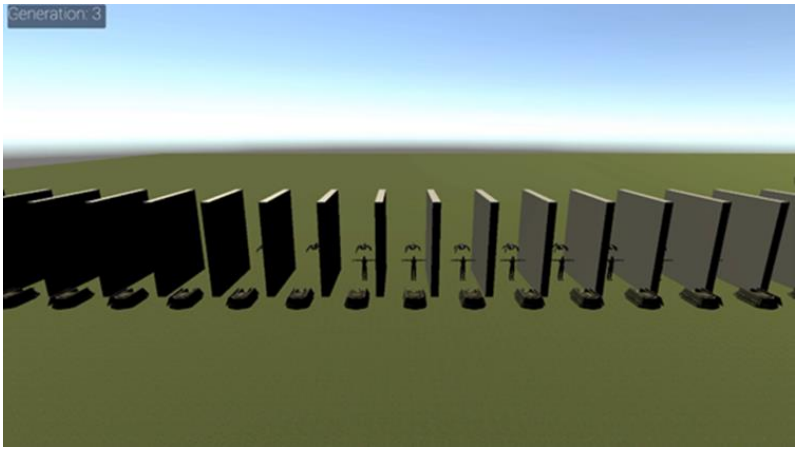


Figure 33: Genetic algorithm’s training environment

Each time a turret kills a zombie or the player, the fitness score increases by 1 unit and when kill robots the fitness score decreases 1 unit. Each generation of turrets had a limited time to exist, so, when a countdown ended, the half of turrets that had the higher score reproduce and outbreed more fit individuals. This is done by picking the half most fit individuals of the current population and choose two of them to breed a new child. The crossover is done by picking up the first half genes from the first parent and the last half genes from the second parent. This process is done until the number of children reached the number of the parents' generation, to maintain the population with the same number of elements. This process will continue until the turrets are fit enough to only shoot zombies and player.
Finally, the genes are picked from a turret of the final generation and inserted in the turret of the main game. Although the results of the NPC could be achieved without a genetic algorithm, just by using simple if-else statements to detects a player or zombie, it is interesting noticing the turrets evolving each generation and at the end only shooting at zombies and the player. A genetic algorithm was not used to make the decision method per se but helped deciding what were its targets.

3.2.6 Driver - Machine learning

The driver (see Figure 34) is an NPC that as the purpose to drive the player along a path, while the player is shooting at enemies (see figure 35).

Figure 34: Car model

Figure 35: Driver and player

To help creating the decision-making process, ML-agents was used, a Unity machine learning framework that uses Python and TensorFlow that supports reinforcement learning. Reinforcement learning consist of two processes, training, and inference.
During the training process, the NPC tries to get the best rewards possible. The rewards are feedback that tell the NPC if an action that is being taken is good or bad. To get the best rewards possible the artificial neural network tries to adjust the weights of the nodes to acquire the best outputs for any given inputs. During the inference mode, the NPC's artificial neural network has already been trained and the weights of the nodes are no longer adjusted, so, the actions the NPC takes are from the already trained ANN.
In both processes, training and inference mode, the perceptions of the NPC need to be converted from unity data into input data that the ANN could understand. Then, the ANN processes that data that is afterwards converted into output data that are the actions the NPC will take in the unity's environment.
For the NPC to drive along the path properly, first it needs to be trained. A track was created with walls and checkpoints to give feedback when it is making progress (see Figure 36). The objective is to increase the distance from the walls and decrease from checkpoints. The driver learns through rewards, so, when collides with a wall receives a reward of -1 and when collides with a checkpoint receives a reward of +1. A new episode of the training starts when the driver collides with a wall or with the last checkpoint of the track. This means that each time the car collides with a wall, it is respawned in the beginning of the track, but this time with preview knowledge (better ANN connections values) already from the preview episodes. The driver uses a default machine learning agent component called ray perception sensor 3D, that fires raycasts to discover at what distance it is from the walls and checkpoints (see Figure 36). The data calculated by the

raycast sensors is constantly being added to the observations. It is also added to the observations the dot product between the forward vector of the car and the forward vector of the checkpoint. In that way we can get the correct rotation that the car should have and learn how to face the same direction has the checkpoint.
Observations are the input data that are fed into the ANN. The Unity's ANN accepts different data types as input such as scalar numbers and vectors.
The car has a constant velocity and the only thing that changes is the angular velocity of the y axis (see green arrow in the Figure 36). Based on the values from the raycast sensors, rewards and dot product referred previously, the artificial neural network will then output a value that will be used to rotate the y axis of the car.

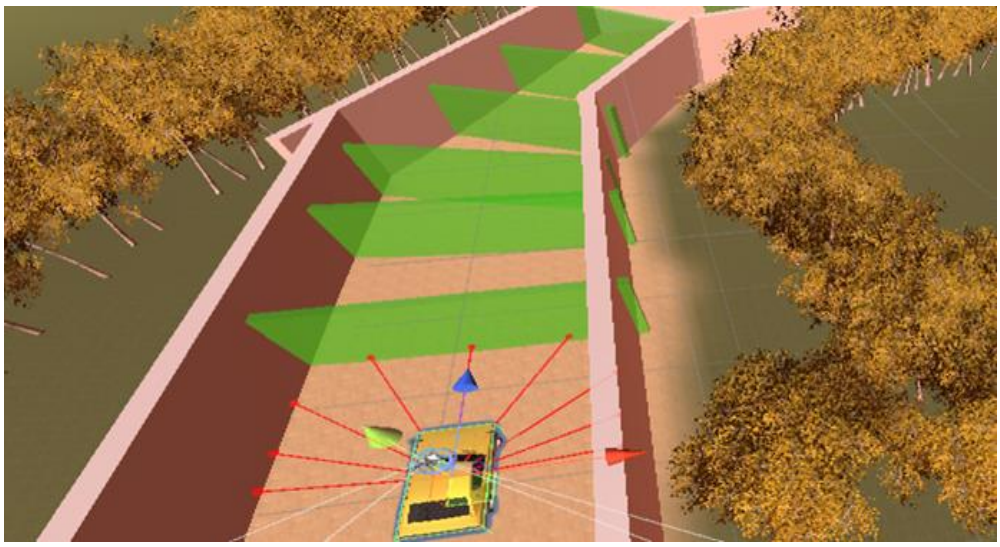


Figure 36: Track walls, checkpoints (green), raycast sensors (red lines)

When the car no longer collides with walls, we can terminate the training process, save the neural network's connections values that were improved during the training, and hide the walls and checkpoints from the track (see Figure 35).
Finally, during the inference mode, the car uses the trained artificial neural network to make decisions to increase the distance from the walls and decrease the distance from checkpoints by rotating the y axis properly. This decision method is similar how humans act, where the perception of the world lead to decisions and decisions lead to actions.

Chapter 4

Results and discussion

We are going to compare the decision methods with each other (see Table 3) and see when they are recommended to be used, since all of them have its own advantages and disadvantages. Although genetic algorithms can be used as decision making processes, there are no real examples of use in the video game industry. Genetic algorithms are recommended to be used as support of decision-making process methods.

Table 3: Table of results comparison

	Finite state machine	Behaviour Trees	Goal oriented action planning	Machine Learning
Reusability	Bad	Good	Excellent	Bad
Dynamic behaviour	No	No	Yes	Yes
When to use	Simple behaviours	Complex behaviours	Complex behaviours	Complex behaviours
Performance	Good	Good	Medium	Bad
Maintainability	Hard	Easy	Easy	Hard
Scalability	Bad	Good	Excellent	Relative
Implementation difficulty	Easy	Medium	Hard	Hard
Debug	Easy	Easy	Hard	Hard

4.1 Reusability

In Finite state machines the transitions are inside the states, so if we want to reuse a state, we need to rewrite the conditions to new states. Behaviour trees allow behaviours to be reused because the transitions are not inside the leaves itself but on the tree structure. In goal-oriented action planning each action is decoupled from each other, so behaviours can easily be reused. In machine learning the behaviours are learned during the training process and are not pre-written, making it harder if not impossible to reuse.

4.2 Dynamic behaviour

Finite state machines and behaviour trees don't allow a dynamic behaviour because all the transitions from one action to another are prewritten. This means that the behaviour of the NPC will always be the same when a specific situation is presented. On the other end, NPCs that use goal oriented action planning or machine learning can have a dynamic behaviours because in the face of the same situation, the NPCs can present different behaviours since the transitions between actions are not prewritten.

4.3 When to use

Finite state machines are recommended to be used when the NPCs don't have much diversity of behaviours and consequently not a lot of states and transitions need to be implemented. The complexity of the finite state machine can grow easily and be hard to represent. Behaviour trees can be used when the behaviour of an NPCs is complex. Due to the tree properties, branches and leaves can be easily reused (inserted and removed). Goal oriented action planning and machine learning can represent complex behaviours and add some extra realism due to the capacity to present different behaviours to the same situation.

4.4 Performance

Finite state machines and behaviour trees don't require much computational power and can be implemented simply with if else conditions. Goal oriented action planning has a worse performance relatively to behaviour trees and finite state machines because it is always planning what actions should be executed to achieve the goal in real time, and to do that, it is always calculating the path using a pathfinding algorithm such as A*. Machine learning has the worst performance in comparison with the other decision methods because the NPC calculate the behaviours during its runtime. We also must consider the amount of work put into training a neural network and possibly the memory requirements on using different type of data to make its nodes.

4.5 Maintainability

Finite state machines are hard to maintain because removing or adding states to the system will require changes to the conditions inside states. Behaviour trees are easy to maintain because adding or removing leaves don't affect the other leaves or branches since the relation between nodes is in the tree itself and not on the leaves. Goal oriented programming is easy to maintain because the actions are not connected to each other and have no relation between them. Machine learning is hard to maintain because every time we want to modify the parameters of the system, we need to retrain it again.

4.6 Scalability

Finite state machine complexity can grow easily just by adding few states. The system becomes confusing and hard to understand. Once again, have the problem due to the connection between the states is inside the states themselves. Behaviour trees can handle this problem easily by reusing the branches, leaves, or sub-trees. Goal oriented action planning complexity doesn't grow by adding actions because the actions are independent from one another. Machine learning scalability is a little different from traditional programming because it depends on how the system is designed. It is the responsibility of the designers to create a scalable system.

4.7 Implementation difficulty

FSM is relatively easy to implement in comparation with other decision methods since it can be implemented with some basic knowledge of object-oriented programming.
To implement behaviour trees, it is necessary to have knowledge on how computer's science trees work. GOAP requires a more advanced knowledge to implement. To implement the planner, it is necessary to have knowledge in pathfinding algorithms such as A*. Machine Learning is the hardest to implement because requires knowledge in various areas such as probabilities, statistics, and computer science.

4.8 Debug

Finite state machines and behaviour trees can easily be debugged because the conditions are prewritten before the runtime allowing understanding the flow of the system. Goal oriented action planning is hard to debug because the system is constantly creating new chains of actions in real time. This means that the transitions between actions are not prewritten and consequently it can lead to unpredictable behaviours.
Machine learning can be hard to debug due to millions or thousands of parameters changing making it hard to understand where the problem is.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

With the conclusion of this project, it was demonstrated that there are multiple ways to build decision methods of NPCs and that a magic formula does not exist. Each decision method has its own advantages and disadvantages, so it is the developer's responsibility to choose which decision methods fits best with the type of behaviours of the NPCs they want to create. It's not worth the effort of implementing a complex decision method such as GOAP or machine learning if the behaviours can't be noticed. One good example is Doom (2016), where the developers opted to use a finite state machine instead of a more complex system, since the game is fast passed, and the player has no time to appreciate the enemies' behaviours. Due to video games realism and complex environments, they can now be used as a testing ground for academic research in artificial intelligence. In the years to come, it might be possible to see a replacement of standard techniques by ones that are considered "true artificial intelligence". It might be possible that the NPCs will be powered by machine learning, not only in the decision methods, but in animation and sound effects [19].

5.2 Future work

It would be interesting to create a full video game or a demo, since most of the mechanics of an FPS are already implemented in this project. Machine learning can be explored more in depth, since in each different dataset we use to train our model can impact our inference greatly. The takeaway here is that pre-processing the data recorded from the environment is a study/project by itself and understanding the intrinsic relations between different parameters is key to properly train a model.

Bibliography

[1] https://github.com/id-Software/wolf3d. Last access: October 2022.

[2] https://github.com/id-Software/DOOM. Last access: October 2022.

[3] Devang Jagdale, (2021). Finite State Machine in Game Development.

[4] https://blog.zhaytam.com/2020/01/07/behavior-trees-introduction/. Last access: October 2022.

[5] http://alumni.media.mit.edu/~jorkin/goap.html. Last access: October 2022.

[6] https://shivagupta14.blogspot.com/2016/02/goal-oriented-action-planning.html Last access: October 2022.

[7] Jeff Orkin, (2006). Three States and a Plan: The A.I. of F.E.A.R.

[8] Nicholas Cole, Sushil J. Louis, and Chris Miles, (2004). Using a Genetic Algorithm to Tune First-Person Shooter Bots.

[9] Chishyan Liaw, Wei-Hua Wang, Ching-Tsorng Tsai, Chao-Hui Ko & Gorden Hao, (2013). Evolving a team in a first-person shooter game by using a genetic algorithm.

[10] Pablo García-Sánchez,Juan Julián Merelo Guervós,Antonio Mora, Anais Martinez-Garcia, Anna Esparcia-Alcázar, (2010). Controlling bots in a First Person Shooter game using genetic algorithms.

[11] T. Bullen and M. Katchabaw, (2008). Using genetic algorithms to evolve chareacter behaviours in moder video games

[12] Daniel Shiffman, (2012). The nature of code, the evolution of code

[13] https://www.ibm.com/cloud/learn/machine-learning. Last access: October 2022.

[14] https://mitsloan.mit.edu/ideas-made-to-matter/machine-learning-explained. Last access: October 2022.

[15] Shehroze Bhatti, Alban Desmaison, Ondrej Miksik, Nantas Nardelli, N. Siddharth, Philip H. S. Torr, (2016). Playing Doom with SLAM-Augmented Deep Reinforcement Learning.

[16] https://www.tomshardware.com/news/impossible-to-detect-cheating-tool. Last access: October 2022

[17] Anssi Kanervisto, Tomi Kinnunen, and Ville Hautamäki, (2022). GAN-Aimbots: Using Machine Learning for Cheating in First Person Shooters

[18] https://www.theverge.com/2018/3/22/17150918/ea-dice-seed-battlefield-1-ai-shooter. Last access: October 2022

[19] https://www.ea.com/news/teaching-ai-agents-battlefield-1. Last access: October 2022

[20] https://github.com/sploreg/goap. Last access: October 2022