WEST UNIVERSITY OF TIMIŞOARA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
BACHELOR STUDY PROGRAM: Computer Science in English

BACHELOR THESIS

SUPERVISOR:
Lect. Dr. Isabela Drămnesc

GRADUATE:
Dragoș Ursan

TIMIŞOARA
2023

WEST UNIVERSITY OF TIMIŞOARA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
BACHELOR STUDY PROGRAM: Computer Science in English

Artificial Intelligence In Video Games

SUPERVISOR:
Lect. Dr. Isabela Drămnesc

GRADUATE:
Dragoș Ursan

TIMIŞOARA
2023

Abstract

When people think of AI, they think of complex system that can mimic a human, abstract machines capable of independent actions which sport a high degree of agency. In terms of video games, AI is akin to a magic trick. Through the use of certain algorithms and techniques, one can achieve very believable results, whilst keeping a low profile, which is an important aspect in video games. This paper explores these techniques and algorithms in short, providing examples at every step, and easily accessible resources. In addition, there is a small project that features a few implementations based on the topics discussed.

Contents

1 Introduction **5**

2 Related Work **7**

3 Artificial Intelligence in Video Games **8**

3.1 History in short 8

3.2 Techniques 8

3.2.1 Movement 9

3.2.2 Animation 12

3.2.3 Sound 12

3.2.4 Decisions 13

4 The demo **20**

4.1 Project Description in short 20

4.2 Game Description 20

4.3 Goals 21

4.4 Gameplay Instructions 22

4.5 General Gameplay and Ambiance 23

4.6 Technical Aspects 23

4.6.1 The engine 24

4.6.2 Run Instructions 24

4.6.3 The project 24

4.6.4 The techniques and algorithms 28

5 Scrapped Ideas **33**

6 Challenges **34**

7 Conclusion **36**

Bibliography **39**

Chapter 1

Introduction

Artificial intelligence (A.I. or just AI) is a vast and hard to clearly define domain ranging from advanced chatbots, to much more common place applications such as OCR (optical character recognition). According to Russel and Norvig [18], there have been multiple definitions for *A.I.*, all of which converge around the point of "machines that think". Due to the nature of A.I. as seen through the lens of a game designer, we do not really require a strict definition, as we are mostly concerned with behaviors and the perception of thinking. As such, whenever you encounter the word AI, it might be helpful to think of bots, which although distinct, usually have similar purposes.

One might come to question about why is it important to have AI in video games. A valid question indeed, and a suitable answer is: for fun. Most games are meant to provide entertainment, they are meant to provide a certain experience to their players. Games do not have a definite structure, as they are creative in nature, some are closer to simulations, such as Euro Truck Simulator 2[1], whilst others, on the polar spectrum, exist to provide memorable experiences, such as The Jackbox Party Pack [2]. Regardless of their purpose, in the end, we play games because we find something that we enjoy. This might be the reason why video games have been incorporated in industries.

Every aspect of a video game is important, and for every one of them, there is a certain kind of reasoning and purpose, each contributing their own value to make up the game. Looking at any game, at random, there is a high probability there would be some kind of agent. They can be players, but in most cases they are neutral, benevolent, or malevolent entities that are computer controlled. Throughout this paper, we will be referring to them by their more colloquial term, NPC (non-player character). These NPCs might serve simple purposes, such as animals, insects, a crowd of people (relevant mostly for racing games such as Dirt Rally 2.0[3]) or pedestrians (such as those from Grand Theft Auto 5 [4]). They have the purpose of populating the world, giving it some movement, some life. Other NPCs are more complex, and can react and remember your actions. These kinds of NPCs are closer to us, and we overlook the fact that, at the end of the day, they still only live as long as electricity flow. Such notable examples are Trip and Grace from Façade [5], as they can react to player written text, have moods, and they pressure the player into participating in social games. Whichever might be their purpose or complexity, all NPCs are instances of AI whose existence in the virtual world is meant to fill a certain kind of role, and for this reason, we can also refer to these NPCs as actors.

In broad terms, AI in video games represents, in almost all cases, a collection of algorithms or scripts that are meant to maintain the outward appearance of a thinking entity. As our perception guides us to anthropomorphize[8], we see thoughts behind agency. This gave rise to plenty of techniques meant to provide enough context, and actions at the appropriate time in order to mimic intelligence. As we venture further down this thesis, we will encounter various examples of AI implementations that achieved their purpose, how they achieved it, and various other aspects in regard to both implementation, and debugging. What should be noted is that, due to the nature of the video game industry, not all examples will have a publicly available codebase. I've done my best to provide games that are open source, but be warned nonetheless.

In order to provide the best reading experience, one is not expected to possess any particular set of skills, as such, this paper is geared towards those that want a short and compact entry point to the world of AI in video games. All examples are (or were) popular titles, and where codebases are available, the code will be summarized. This means that more advanced topics are not necessarily off the table, but they will not be discussed extensively (such as deep learning) as the end goal is to provide a good starting point for further research into AI in video games, or, at the very least, a good enough read for someone new to this domain.

In addition, although not a necessity by any means, any kind of knowledge about games (either from a developer or from a gamer) is more than welcome. There will be mentions regarding immersion (i.e. when the player has mentally entered the game world), emergent gameplay (i.e. actions or activities not directly designed, but happen to occur) and gameplay experience. In this regard, I would recommend Jesse Schell's book, The Art of Game Design [19]. It is not specifically aimed at video games, and it provides an intuitive system to gauge various aspects of a video game.

Another information to keep in mind throughout this paper is the various names we use when referring to AI.

- An **agent** is an individual entity carrying its own tasks based on a given ruleset
- An **NPC** is an entity which can either be an agent, or it can be scripted, acting on specific cues or working just as an interface, like a merchant
- An **AI system** manages the game's various aspects and parameters working with a certain ruleset

Finally, there is a demo attached to this paper, that features a few ways of implementing AI in a video game. This demo has been made explicitly for this paper, by the author, using Godot [5]. Further details regarding design and implementation are present in 4.

Chapter 2

Related Work

Although there is plenty of research in this domain, I wasn't able to find many resources that were modern, complete and aimed at a more general audience. By far, the closest resource is Artificial Intelligence For Games by Ian Millington and John Funge [12]. This book represents the backbone of this thesis. It has a lot of content, and it touches as many concepts as possible, whilst keeping all explanations concise.

Although not directly reference throughout the thesis, I have read parts of Steve Rabin's AI Game Programming Wisdom [16], especially the fourth book in the series, to help me better grasp a few concepts like scripting.

Another resource, although not necessarily academic, is a small YouTube channel, ran by Dr Tommy Thompson, called "AI and Games" [22]. It was my starting point for a lot of topics discussed in this thesis. Most of his videos are compact, and full of explanations. I have managed to find a lot of useful information in his work, which I would recommend as an introduction to AI in video games.

The Game Development Conference (or GDC) [4] is a great online resource full of talks from various game developers. As many of the popular games are usually closed source, these talks, coming from the people that worked with the actual game, or with people developing the game, shed light on the inner mechanics and challenges that came when implementing various aspects that make up a video game, including artificial intelligence. Although quite lengthy, they provide enough content to get a deeper understanding of some of the more complex sides of making games.

Finally, we have The Artificial Intelligence for Interactive Digital Entertainment Conference (AIIDE) [2]. Running for about two decades, This conference features a lot of papers on many topics pertaining to AI in video games. A lot of these papers target specific subjects related to pathfinding, decision-making and procedural content generation. It is by far the most consistent source of inspiration, but also the hardest to navigate due to the sheer amount of work hosted there. I would recommend this as the next stepping stone for anyone pursuing more advanced topics in this field.

Chapter 3

Artificial Intelligence in Video Games

3.1 History in short

Early video games, such as Spacewar! [6], did not feature NPCs, instead, as their board or tabletop counterpart, featured a second player, usually as a rival. Things have slowly progressed since then, with Pong [7] and Pac-Man [8] being two of the more memorable games of their era. At that time, the logic behind the actors was very simple [12], and as the industry picked pace, so did the innovations in AI. The ghosts of Pac-Man had the goal of chasing the player, but each of them, based on an internal state, would exhibit a different behavior, which gave them some intelligence, aiding the illusion that there was actually some thought behind the action.

Around the mid 90s, we saw games like Thief: The Dark Project [9] whose gameplay was all about being stealthy or Half-Life [10] which presented the player with more believable characters, as well as more intriguing enemies. Games like Halo: Combat Evolved [11] and First Encounter Assault Recon (F.E.A.R. or simply FEAR) [12] had come up with their own or promoted innovative techniques (Decision Trees, and later on Behavior Trees for the Halo franchise and Goal Oriented Action Planning in F.E.A.R.).

Since then, AI in video games has come a long way and has evolved to include more complex behaviors and advanced algorithms. The research and accumulated knowledge allow nowadays games to achieve complex behaviors, with enemies that seem to learn (the xenomorph from Alien: Isolation [13]), advanced ecosystems (the critters in Rain World [14]), smarter enemies and much more. Further down this paper, we will explore more about these techniques, what are their strengths, what are their weaknesses, and how to apply them.

3.2 Techniques

In order to make an NPC more believable, there are multiple factors involved in its inception. In almost all cases, they need to make decisions on their own, find their way through the environment and telegraph (visual or auditory cues that can tell the player what the NPC is or will be doing). These individual components are not

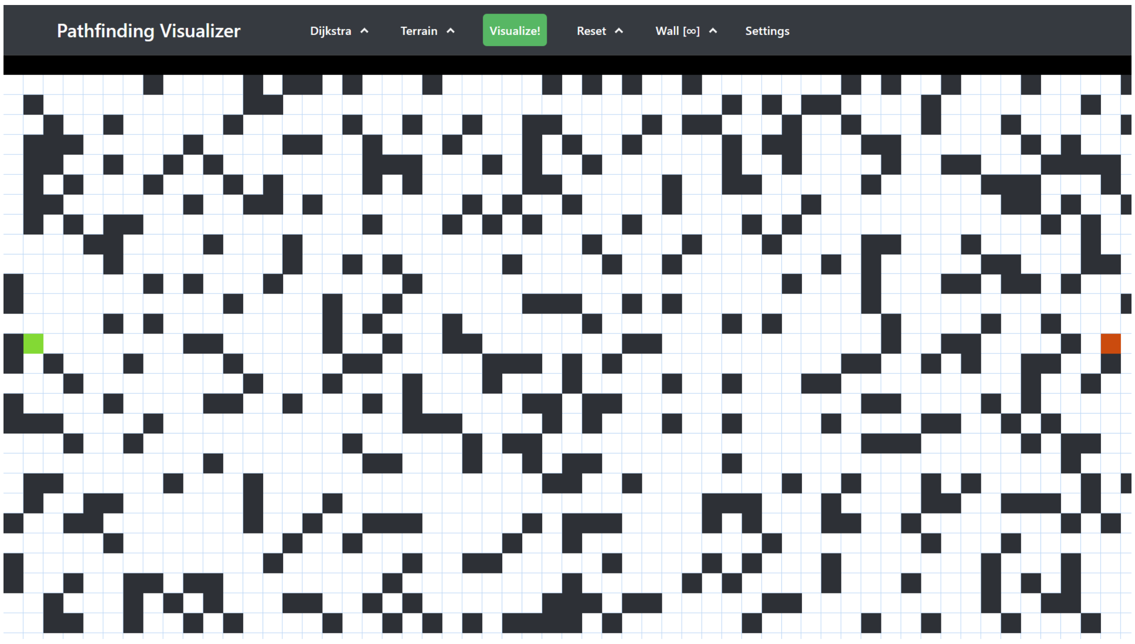


Figure 3.1: A Clear Picture Of The Grid

necessarily always present, but at least one of them has to be present, or else we are not dealing with NPCs or agents, we might as well be dealing with a button!

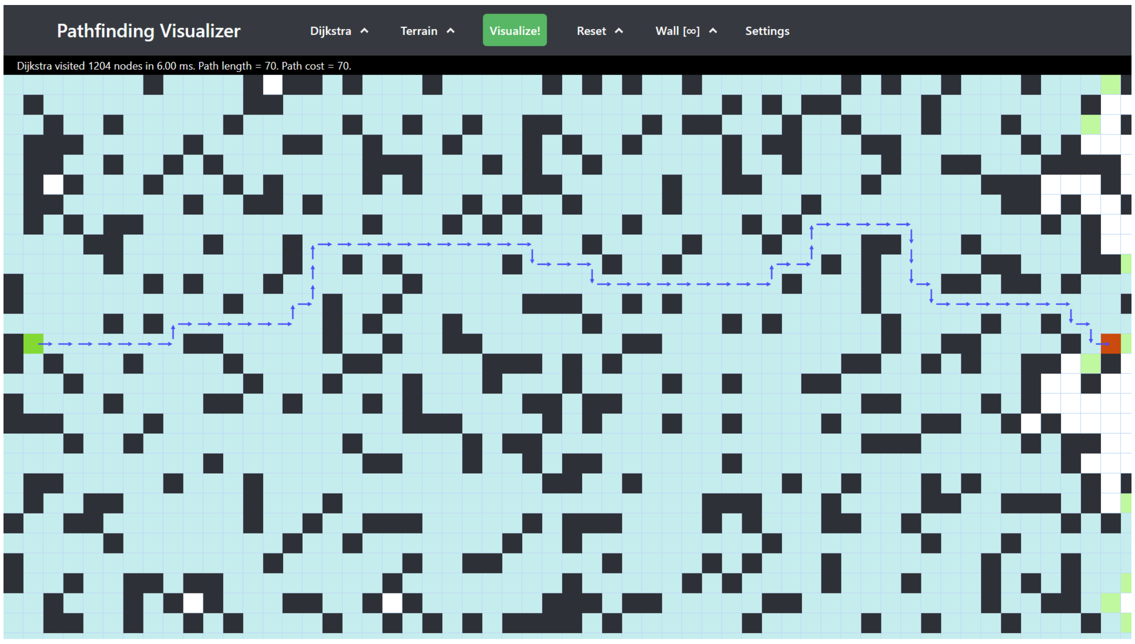
In the following subsections we will be exploring various topics regarding movement (or pathfinding), animations (visual cues), barks (auditory cues) and decision-making (or thinking) techniques. Each of these might very well be expanded into their own papers, as the possible choices, and their advantages and disadvantages, are quite vast. As such, the information presented shouldn't be seen as nothing more than a summary.

3.2.1 Movement

Movement is one of the most basic actions in a video game, as well as the primary way we as players understand how the environment works. We expect things to move, both as a sign that something is happening in the world, as well as a sign that the game is still working. Most decisions are followed by an action, "I want to eat, so I'll go to the fridge", "I want to fight a monster, so I need to prepare for my attack". There might be some decisions or reactions that are not necessarily followed by movement, as is the case in Fallout 3 [15] where the player is alerted by an in-game message that a character will remember, and subsequently adapt, to a choice the player made, but in general, we can safely assume that decisions mean movement.

In terms of physics, to be in motion means to experience a change in position over a period of time. This can be easily achieved through code, animation, or by the use of a physics engine, but there is a problem: what do we do if we can't reach our target directly? We need to be able to make decisions based on the environment topology in order to reach our target, and this is where pathfinding algorithms come into play.

Pathfinding algorithms have been applied and researched way before video games, or even computers, have surfaced. All of this research comes from graph



Dijkstra visited 1204 nodes in 6.00 ms. Path length = 70. Path cost = 70.

Figure 3.2: Dijkstra's Algorithm for Pathfinding

theory, which is a part of mathematics that studies graphs, their compositions and their connections.It is up to the developers to interpret and adapt a graph into their game, in order to leverage all the knowledge available in the field, and based on the needs of the game, there are two common approaches to pathfinding:

- Dijkstra's algorithm
- A*

In order to provide a better understanding of these algorithms, will be using Joseph Prichard's pathfinder visualizer [15] online demo. I have tested a few other online pathfinder visualizers, and I found this to be by far the easiest to use. In 3.1 we have the initial state of the grid. The dark-colored squares represent walls or any other form of impassable terrain.

Dijkstra's algorithm

Edsger Dijkstra proposed an algorithm that computes the shortest path (the path with the lowest cost) between two nodes given a weighted graph. To put it simply, it constructs a complete map of the environment, taking into account all possible paths from a starting point. This algorithm is useful when we want to assess the shortest distance to all possible locations, however, most of the time, we want to know the shortest path towards a single point. This means that the algorithm will have explored a good chunk of the graph before arriving at the given goal. Given the time it takes to compute the cost towards every single node, it starts to become apparent that time is being wasted, because we are looking everywhere, as we can see in 3.2.This is where our second technique comes in handy.

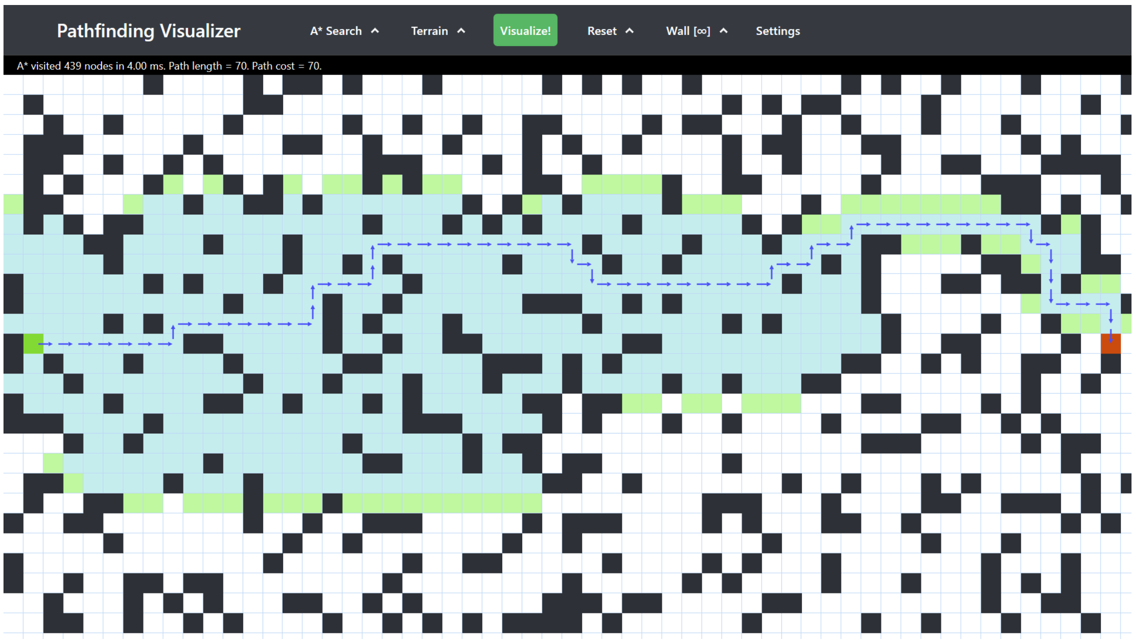
A*

A* (pronounced A-Star) is a similar algorithm to Dijkstra's, but with one major difference: we also compute and keep track of an additional value, a heuristic. This means that, in theory, we can expect to achieve results that are not optimal, which would be devastating if we actually cared about the best route available. In practice, A* can be faster, as this speed is solely dependent on the function that computes the heuristic. The simplest way to compute it is by checking the Euclidean distance towards our target. This allows us to focus on getting closer to the intended target, rather than check all available options, and as such, we will be able to find **a** towards it. Some of the common heuristic function used by A* are :

- Euclidean
- Manhattan
- Octile
- Chebyshev

Each have their own strength, depending on whether the grid allows diagonal movement or not, and also strongly dependent on the grid geometry. The Godot documentation provides internal implementations for these 4 algorithms [3]. We can also find the specific implementation of these algorithms used for the pathfinding visualization in Joseph's GitHub repository [1].

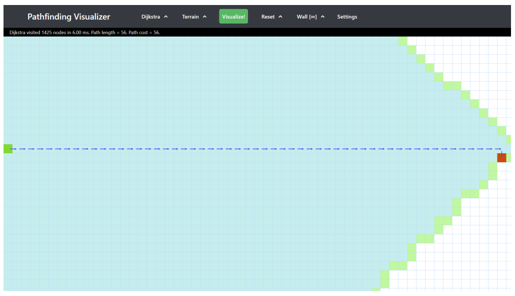
In the end, if we don't care about accuracy, A* should be the preferred choice, as in most games no one cares that the path an NPC takes is suboptimal. We can see in 3.3 that a lot less nodes have been visited (both visually, and as indicated by the console output), and that a different path was chosen.



```
A* visited 439 nodes in 4.00 ms. Path length = 70. Path cost = 70.
```

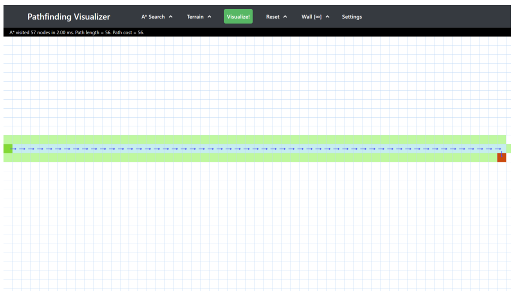
Figure 3.3: A* using the Manhattan Heuristic

It should be noted that when pathfinding in an open environment, A* is the preferred choice, as long as the heuristic keeps track of the distance to the target, it will always outperform Dijsktra's algorithm as seen in 3.4.



Dijkstra visited 1425 nodes in 6.00 ms. Path length = 56. Path cost = 56.

(a) Dijkstra's algorithm



A* visited 57 nodes in 2.00 ms. Path length = 56. Path cost = 56.

(b) A* (any mentioned heuristics)

Figure 3.4: Pathfinding in an open environment

3.2.2 Animation

Animation servers as a way to bring life to game entities, as well as remove the need to write detailed movement inside the codebase. Usually, animation is pre-processed in an external software, and then imported and applied to game entities. Nowadays, there are ways to dynamically generate animations on the fly using techniques such as inverse kinematics (a method of computing how joints should be rotated in order for an end-effector(e.g a foot, a hand) to be in a specific place.) or procedural animation [17] which provide a sort of organic movement, that can adapt itself to rough environments, without the need for large amounts of animations and animation blending. A few good examples of these are Rain World [14] and Overgrowth [16].

In addition to improving the overall quality of the experience, animation also plays a critical role in the illusion of artificial intelligence. By providing nonverbal cues through gestures and movements, players can infer the current state of non-playable characters. For example, we can deduce that a guard is talking to another guard, or that a guard has spotted our player just by looking at their gestures. This implicit communication provides a powerful way of communication, since players can now to gather information without relying on text or dialogue (Bonus points when the game isn't in displayed in a language known by the player).

3.2.3 Sound

Sound is an important aspect when it comes to video games. When it comes to AI, there are two main ways in which we can use sound: barks and dialogue. Albeit as important as the visuals of the game, sound is very well researched, and it's even worse when it comes to its use for AI. To provide better context for this, we will be relying on Thomas J. Holmes' master thesis "Defining Voice Design in Video Games" [7], which I found to be quite an interesting read, as well as a good starting point for this topic.

Dialog

Dialogues are the main course of storytelling when it comes to video games, as they convey essential information about the game, the world, a mission, or even about the NPC itself . Indeed, some games can very well survive without a single line of spoken(or written) dialogue, but when it comes to conveying intelligence, we can soon find that a mute character seems rather chaotic and random, since we cannot infer what it is thinking.

For example, in Hades [17], every character has different unique dialogue lines when interacting with the player (or even between themselves) and even the characters that don't really speak (like Charon) still make use of grunts, and other unintelligible sounds to convey their intentions [9].

Barks

Barks represent audio clips that convey some information about the current state of the actor or the world. Though Holmes breaks this category down in multiple parts, for our intents and purposes, we will use bark in the sense of short lines of dialogue that convey some form of information about the actor itself, or about the world. The main purpose of using these kinds of sounds is to add more life to the characters. It's easier for us to understand a character if they speak out loud what they are currently thinking, doing or plan to do, since examining the body language isn't always reliable (animations can glitch, play at incorrect times, or not at all), which means that we lose a got chunk of information, that needs to be supplemented in some way.

In Dishonored [18], the player can learn about the world by listening to NPC dialogue, or learn if they are currently wary of the player (if the player caused a commotion and an alarm was triggered, the guards will be much more cautious), they will shout when throwing a grenade, or when one is thrown their way, and will alert everyone nearby when they have to chance to engage in combat. Through clever use of audio cues, the player is made aware of the presence of enemies nearby, as well as their intent, which gives the player the upper hand, and some sense of control over what's happening.

3.2.4 Decisions

Another important aspect of an AI system in video games is its ability to take decisions for itself, or for other entities. There are a quite a lot of ways to encode such a system, and in the following sections we will be looking at the most common ways decision-making has been implemented. It should be noted that although we make a distinction between these various techniques, they usually don't appear in such homogenous structures, instead, various parts of the same system, or of the game may make use of one or more techniques and algorithms. And as we will see, some techniques are better suited for a given scenario, whilst others provide more flexibility.

Scripting

Scripting in our context refers to the use of custom written rules, systems and behaviors that are usually atomic and static. Scripts allow us to explicitly define the behavior for a game entity, sometimes including a few branching options. We use scripts when we need to have predictable behaviors. They are quite powerful and versatile, but they are limited by design. This could work very well for a static NPC that doesn't have any decision to take, except for a few dialogue lines, and through the use of animations and sound, we can sell the illusion of intelligence with very little work. Usually, we can observe this kind of behavior in shopkeepers, which represent an interface between the player and a shop.

An example of a simple script to control behavior can be found in the demo, specifically, the chicken AI. Although not very complex, through the use of animation transitions, and a bit of delay, all chickens that are spawned, will, after a random interval, settle down, then walk around, then settle down, at random intervals, walking to a random point in their vicinity.

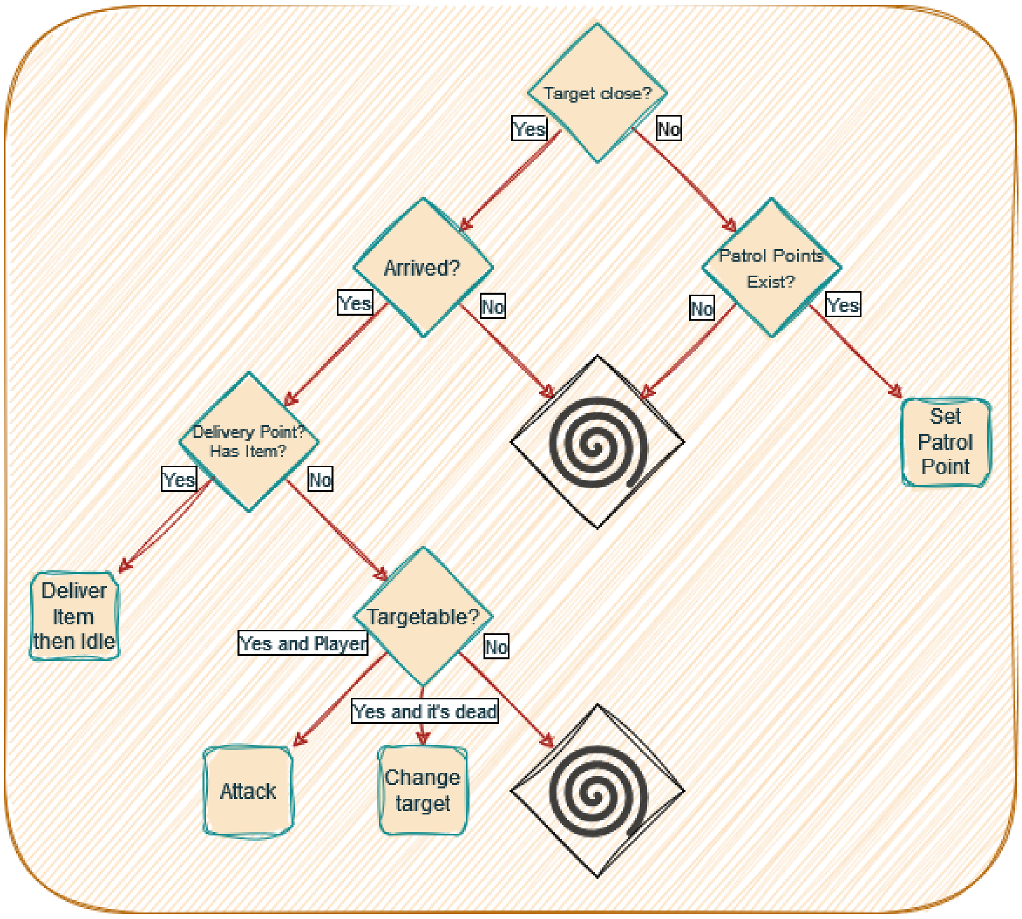


Figure 3.5: Main Decision Tree logic structure

Decision Trees

A decision tree (or DT) is a simple to implement, and quite effective, approach that allows use more flexibility when it comes to designing AI. As their name implies, graphically, we can represent a given behavior in a tree-like structure. By using branching logic, we can effectively allow the entity to adapt to changes into its environment.

Decision trees usually contain two abstract components: action nodes and decision nodes. Action nodes (or leaves) represent concrete and atomic activities that can be executed. For example, walking towards an enemy, firing a gun, opening a door, etc. In order to be able to choose which action node to execute, we require decision nodes. They allow an entity to decide between, usually, one or more nodes. There are multiple flavors of decision trees out there, but for our purposes we'll stick with this simple one. By definition, whenever we run our AI logic, we start from a root node, which is just the first decision node in the tree, we check our current condition and parse decision nodes until we reach an action node. One thing that should be noted is that, when designing such a tree, we need to make sure that it is balance and simple, i.e. that we don't spend too much time traversing it, as these can negatively impact performance.

For example, an implementation of decision trees has been used in the demo for the NPCs, specifically the "simple" version. Made out of a series of if/elif/else statements, we can see how most of the time, only one action is performed at the end, though it's not mandatory that this should be the case, this represents a simple implementation of a decision tree.

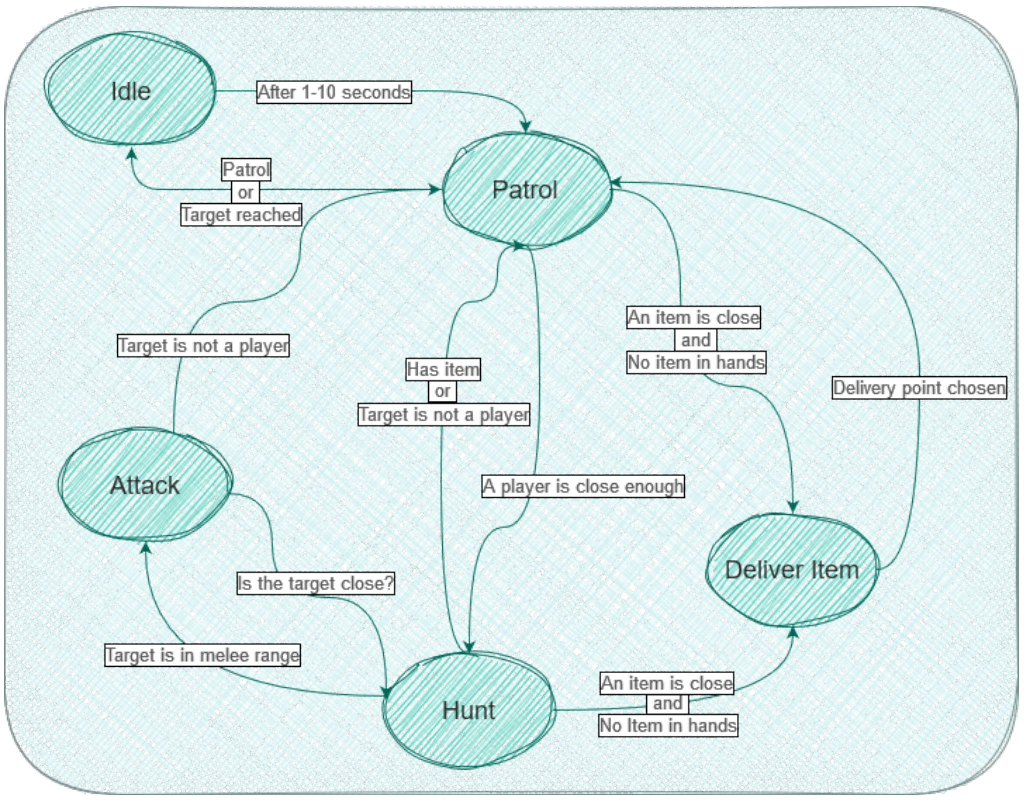


Figure 3.6: Main Finite State Machine logic structure

Finite State Machines

Finite State Machines (FSM) are an important aspect in computer science, due to their power to provide clear and easy to use abstractions for a range of functionalities. At their core are states. An FSM can only be in and process only one state at any given time [23]. In video games, FSMs are used when we want to encapsulate and connect complex behaviors in order to achieve believable results. As the name implies, in order to achieve this, we make use of states, which contain information about both the internal state and the state of the world. This means that it's easy to create and link these states. However, poor planning can prove catastrophic, as larger FSMs are usually more complex, and harder to debug [21].

FSMs show their true colors when the behavior doesn't need to be complex but still diverse. As an example, when designing a stealth game, it might prove more advantageous to divide a guard's behavior into a patrolling state and an aggressive state. While patrolling, the guard is looking around, moving at a slower pace and is not yet ready to engage in combat. If the guard spots something suspicious, it will go and check. If it can clearly identify a threat, it shall switch into a more aggressive state, and chase, possibly even attack, the target. A good game for this example would be Thief: The Dark Project [9] (additional information about how the AI has been implemented can be read in Tom Leonard's [11] article).

Behavior Trees

A behavior tree (or BT) represents another approach to design the logic of an AI through the use of a tree structure, which also makes it easier to represent them graphically. The difference between this and a Decision Tree is that BTs make use of a few specialized nodes for flow control: sequence, selector, action, and decision nodes. Additionally, these nodes always return some sort of status when they are being executed, usually either success, failure, or running.There are multiple other implementations that offer additional nodes and statuses, but these are always present. Action and decision nodes work similarly to their decision tree counterparts, but here, instead of using decision nodes to branch out the tree, we make use of the selector and sequence nodes.The selector node runs through each of its children, in order, until one of them succeeds, whilst, the sequence node will stop when one of its children fails.

This kind of design makes it easier to break down behaviors into concrete actions, as well as provide an intuitive way to visually explore and expand the tree, without any exponential increase in complexity, and most importantly, does not require specialized knowledge to design. This is a very strong point, given that FSMs and HTNs require significantly more work to design and implement, and especially unlike FSMs, make better use of modular code. Though it should be noted that it's not always easy to design a good behavior tree.

A very good example where behavior trees are used is the xenomorph from Alien:Isolation, which makes use of two AI systems, a director AI, that oversees from a higher point the pacing of the game, and nudges the alien towards the player, and a behavior tree that controls the alien itself [20]. The interesting part about this implementation is that parts of the tree are not accessible initially. As the player progresses through the game, some of its action allow the xenomorph to permanently unlock additional branches of the tree, effectively simulating learning.

Hierarchical Task Network

Hierarchical Task Networks (or HTNs) represent another approach to modelling AI that focuses on a static, pre-defined structure (hierarchy) to encode a specific behavior. All possibilities are encoded into this structure, which means that they inhibit emergent behavior (behavior that was not initially planned by the designers). The way in which HTNs work is by recursively decomposing tasks into subtasks, until an atomic (primitive) task is reached. These primitive tasks are the action that the AI will be performing, so selecting the appropriate decomposition becomes crucial. This is where another integral part of the HTN comes into play: the planner. The planner uses the pre-defined decomposition methods to break down high-level tasks into subtasks, evaluating conditions at each level to determine which branch of the hierarchy to descend. It continues applying methods until primitive tasks are reached, then executes the sequence of primitives to achieve the goal. This approach is not unique to HTNs, as we will see in the following sections. A keen eye might have spotted a similarity between HTNs and Behavior Trees. Indeed, both of them make use of a tree data structure to encode the logic of the AI. However, BTs allow much more flexibility, as they can react to changes into their environment. A good example of where HTNs where used is in Transformers: Fall of Cybertron [19]. . This isn't necessarily always good, and for certain applications, it might be more desirable to use HTNs, which are more predictable, and possibly more efficient. For those that wish to delve deeper into this topic, there is a great article by Kelly, J.-P., Botea, A., & Koenig, S. [10] that provides more details, as well as possible implementations.

Goal-Oriented Techniques

To have a Goal-Oriented AI means that we create a system in which we give an entity at least a goal, a set of actions and a way to choose these actions, and then let it find its own way to accomplish it. There is no limit to how many actions or goals we implement, and there isn't a strict definition on how a goal is achieved (for example, fully or partially) or how an action affects other goals (side effects), but it should be noted that the more possibilities we create, the slower the decision process becomes.

Up until this point, we've mostly discussed what are called "reactive" techniques, which mean that, for the most part, the AI was reacting to some input, and otherwise, it would default to a given behavior. When implementing goal-oriented techniques, we give the system some agency in choosing its own goals and actions, rather than only let it act according to an algorithm. This means that we benefit from a more life-like result, but we also must pay the cost of increased complexity. As such, we shouldn't rush to implement the most realistic option, when we can achieve a similar result using simple rules.

There are usually two big approaches when it comes to goal-oriented techniques, and we will be discussing them in the following parts of this section. Those are:

- Goal-Oriented Behavior
- Goal-Oriented Action Planning

Both of these techniques achieve what they set out to do, but they have different strengths and applicability ranges, which should be taken into account should one choose to use either.

Goal-Oriented Behavior

Goal-Oriented Behavior (or GOB) refers to a range of techniques that shift the focus of an AI system away from reactive decisions and towards accomplishing some internal goals. In contrast to the previously explore techniques, there are a lot of ways of implementing this. In general, a goal has a certain value attached to it. Each action affects this value in a certain way. An action can for example completely satisfy or partially satisfy a goal. It is not excluded for certain actions to have side effects. Since there are multiple types of implementations, and according to Ian Milington & John Funge, they are not many games that implement this kind of technique, as such there isn't a clearer consensus on how to achieve a certain kind of behavior. Additionally, there are multiple issues with this kind of technique. The most critical of them being that we can only choose from a predefined list of actions. This can take us only so far.

A good example where this technique has been used (and may be still probably in use) is in The Sims [20]. Due to the very nature of its gameplay, all sims (the humanoid characters) have a few basic goals pertaining to natural needs like hunger, thirst, hygiene and socialize. These needs are mapped into various meters visible in-game, and an action can increase or decrease any of these meters. When a sim is hungry, they have the option to go to the fridge and grab a snack, or cook a meal. When they are sad, they can play games, or find other sims to talk to. Due to the way in which GOB is implemented, there is the possibility that an action might have undesirable side effects (e.g. a sim that is sleep-deprived might choose to cook a meal, which might take too much time to complete and thus, there is the risk that they will collapse on the floor), which aren't always accounted for. This is where our next technique comes in.

Goal-Oriented Action Planning

Goal-Oriented Action Planning (GOAP) was first described by Jeff Orkin in his entry in the AI Game Programming Wisdom 2 [14]. This architecture meant to overcome the limitations of other GOB implementations by providing adding a planner to the system. The planner searches the space of possible actions and selects actions that, executed in a sequence, will satisfy a given goal. This adds a layer of complexity, given that a naive approach can basically run a GOB algorithm for each step. A solution would be to have an encoded representation of the game state, which can be easily checked and modified, and have another part of our code do the heavy lifting.

There is an unexpected turn of events about the planner: we have basically created a space through which we need to find a path; and what better pathfinding algorithm to use than A*? If we have many options, and the world state can be easily checked, we can effectively repurpose A* as a way to navigate, where each world state represents a node, and nodes are linked by actions, in terms of the goals they can satisfy.

There are multiple examples of GOAP being implemented in video games, but one of the most well-known examples is F.E.A.R. In his article [13], Jeff Orkin provides a great overview of how GOAP was implemented in the game. In F.E.A.R., each type of non-player character has an available list of actions from which it can choose, with each action fulfilling a particular goal. The game uses a finite state machine with three states (Goto, Animate, UseSmartObject) to direct the character on what action to execute at any given time. GOAP's role in this system is determining when to switch between these different states.

The available actions for any NPC depend on various factors, including: what goals are relevant to that NPC based on their type; what props or interactive elements exist in the surrounding environment; and what animations/behaviors have been created for that NPC to utilize. The planner must also account for relationships between actions, as well as ordering actions correctly to achieve certain goals.

For example, if an NPC's goal is to "get a soda", the list of actions may include: walk to the vending machine; insert coins into the machine; press the button for a soda; reach down to grab the can; and walk away while drinking. Each action moves the NPC one step closer to accomplishing their goal. The planner is responsible for stringing these actions together efficiently and overcoming any obstacles or pathfinding issues to eventually fulfill the overall objective.

Chapter 4

The demo

Now that we've learned more about the different AI techniques, and also after seeing some in practice, it's time to see a more simple, and modern approach to AI. Although I wouldn't claim it to be the best, I do believe that what we are about to see would serve as a good entry point, maybe even the basis, for a newcomer. Half-Life has great examples of AI implementations, but I felt it to be hard to track at times, and especially since the project was quite large, it seemed natural to want to make a simpler version, that excludes a lot of the nuance and additional mechanics that the game implements. Still, it's a great example, and I highly recommend the entire tutorial series from The Whole Of Half-Life website. As for our small application, the goal is to showcase a possible implementation of an AI system in a video game.

4.1 Project Description in short

In order to showcase the differences one would take in their approach, I have made a small game, using the Godot Engine and its custom programming language, GDScript. The source code of the game will be available through a GitHub repository, alongside the complete current version of this thesis. The repository contains all necessary components to edit and run the project using Godot 4.03 (or a later version, if it doesn't break the project), as well as all assets used. Additionally, all script files contain comments for almost everything, so it should be easier to follow the logic.

4.2 Game Description

Based around a medieval theme, the player takes control of a random character in the game world. The world is populated with characters similar to the player, with the only exception that they are bound to only be able to walk on roads. The player has this freedom given to them in order to be able to explore the given environment without constraints. Throughout the world are a few chickens, meant to be mostly decorations, and items, placed at various crossroads. Both the player and the NPC have the ability to pick items when close to them. The NPCs also have a hidden goal to collect a certain type of item, and when they come into its possession, they will deliver it at a watch tower. There are only a handful of towers around the map, and the AI will choose the closest as the delivery point. Additionally, once an item has been picked up, it can no longer be picked by anyone.

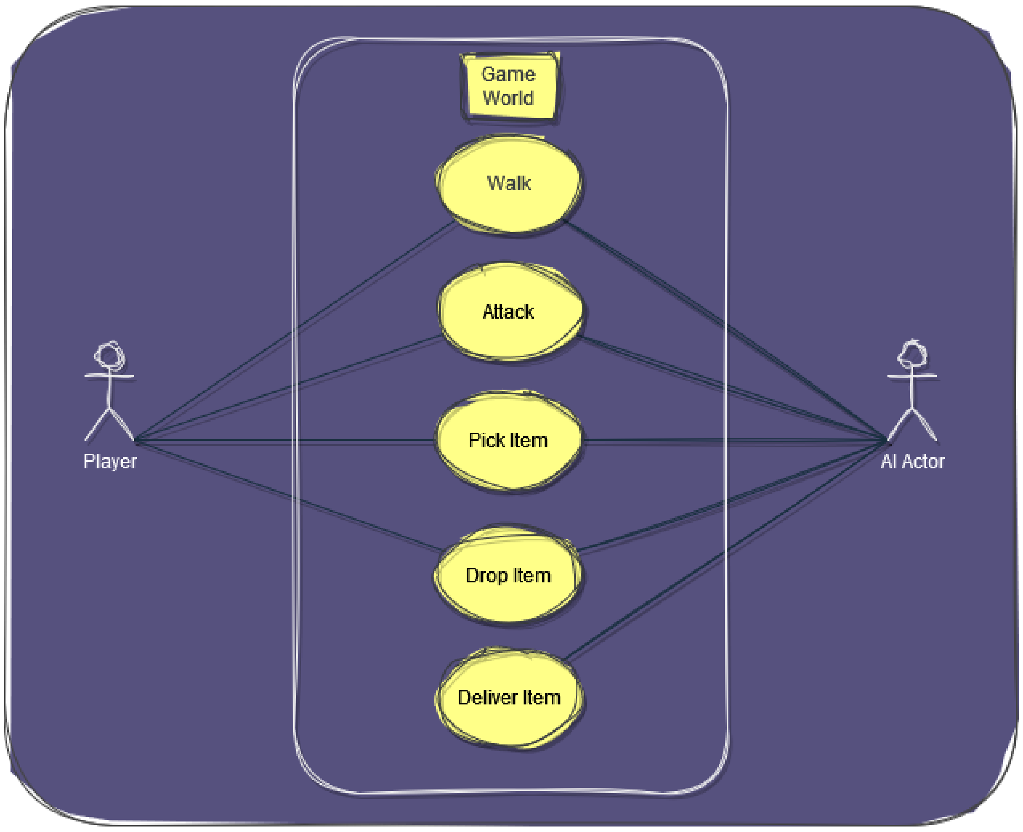


Figure 4.1: Use case diagram for the game

Outside these deliveries, every NPC has a random patrol route through the map. Whenever two NPCs collide, given that the roads only allow one of them to pass at a time, they will play an invisible game of chicken, whose winner gets to continue to their initial destination, whilst the loser has to choose another path. If the player gets too close to one of the NPCs, they will start attacking the player, the latter also sharing this ability. When an NPC, or the player, dies they will play a death animation, and can no longer move or think.

4.3 Goals

The tricky part when implementing AI system is to resist the temptation to make it very smart. It's tempting to keep adding new features, as one would think that the more complex it is, the better it will look; it will be more believable. That's not always the case, as players don't usually spend too much time around an NPC, enough to learn about its complex thought process. Naturally, this could result in wasted effort, which is what we are trying to avoid!

The first goal of this project was to implement some sort of decision-making. I wanted the NPCs to be able to make decisions for themselves, and I wanted to be able to have multiple such functions, and even the ability to switch between them, and more importantly, allow the user to switch them on the fly!

There were quite a few candidates for decision-making: GOAP, BT, FSM, DT, and at some point even some machine learning. After looking at some papers on GOAP, and even some Godot implementations, I decided to not pursue it. All implementations I've seen were quite fragmented. One file contains actions, a script contains the planner, and another one contains the agent. Too many files, too many functions, and too much complexity. It didn't fit with the goal of the paper, which is to make techniques easy to understand. The same thing applies to machine learning as well.

Behavior trees are another story. They were quite simple to explain and visualize, and the code wasn't too complex either, but there was not enough time. I've played with some implementations, but it took me too much time to understand, and I believed it would take much more time to code. So we're left with decision trees and finite state machines.

Initially, when the AI was drafted, it only had two goals:

- Patrol when there is nothing else to do
- Hunt the player when it has been spotted

This proved to be quite shallow, as the AI didn't have a goal of its own, some sort of desire. And so a new goal has been added.

> ”If I see an item that I want, I'll pick it up and deliver it to a delivery point as soon as possible”

Now we've added some more nuance to its behavior. Now it can think for itself, react to changes into the environment, and pursue its own goals. It was now making decisions, it has grown up. What a simple life! The implementation allows extensions, or even new techniques to be somewhat easily added. I am saying somewhat because the code is made in such a way, but the UI is completely detached so any changes in script, can only be modified when the project is being edited in Godot. This applies to both decision-making and pathfinding. Speaking of pathfinding ...

The other goal of this demo was to implement pathfinding in at least two distinct ways. Initially, I wanted to implement Dijkstra's algorithm and A* by hand. As I continued to research, I've found out that Godot had an easy to apply solution for navigation, and it also provided all the tools necessary to create your own A* algorithm. I haven't been too successful in finding tools for Dijkstra's algorithm, I decided to drop it, as I also feared that it would slow down the project too much, both at runtime and at development time.

Though I feel that I achieved my goal, I still believe that one or two decision-making techniques wouldn't hurt. But a more complex code would also make it harder to follow, so I am content with the current configuration.

4.4 Gameplay Instructions

The game features a very simple control schema:

- Movement : W, A, S, D;
- Attack: Left Click;

- Item Pick Up: Left Click (when close to any type of item);
- Item Drop: Right Click (when holding an item)
- Zoom in/out: Scroll wheel;
- Exit: Escape;

There isn't a real end goal for the player. One could go around and commit murder at every occasion, be eliminated on the spot, steal all the items, never allowing the NPCs to collect them, or watch the chickens. Whichever the player chooses to do, the NPCs will continue their routines as if the player does not exist.

An additional, *mechanic* is the ability to choose, during gameplay, what pathfinding and decision-making algorithms the NPCs should use. Although this was made to expose the available options in-game so that the project doesn't need to be edited in Godot to access these additional features, it should be noted that it might look more like cosmetic rather than a tangible change. This is due to the fact that both decision-making algorithms are almost identical in scope, and my implementation of A* is somewhat modeled after Godot's implementation, with the code being inspired by one of Godot's 4.0 demos regarding AStar2D.

4.5 General Gameplay and Ambiance

The game is slow paced, top down and, technically, an adventure game. Though there isn't necessarily any underlying goal, the player is free to explore the map. As the main point of the game is to present a few AI implementations, most mechanics aren't polished. Attacking, picking up and dropping items are all valid actions for the player and NPC, but there are a few instances where an NPC might try to attack, or follow the player, even when he is out of reach, or the player might pick up items by mistake. Some of these difficulties will be explored in the Challenges section 6.

4.6 Technical Aspects

It is now prime time to dive into the nitty-gritty. As it has been stated initially, the entire game has been written in Godot game engine, specifically, Godot 4.0, using GDScript. The project has been entirely written by the author, with most of the help coming from the documentation of the engine itself, and a few parts, actually coming from online tutorials. There are links sprinkled throughout the code where I deemed necessary to point to the relevant source of the code snippet. The assets have been downloaded from itch.io, with all the assets using permissive licenses, or even public domain. In order to be able to edit the project in Godot, one needs a 4.0 version of Godot. Although the project might still work with a later version, there is a risk that an update might have bricked some components of the project, as something similar almost happened when transitioning the project in the early stages from Godot 3.5 to 4.0.

4.6.1 The engine

Godot is an open-source project, and albeit not an old player in the game engine market, it still boasts enough features to be a reliable choice. I won't discuss what are the differences between it and other game engines, as I myself don't have too much experience with anything other than some Unity[6], but what I will say is that the documentation is very well written, and other than a few edge cases, everything I needed was most of the time easily accessible, and easy to follow. Setting aside the advertisement, there are a few technical reasons that I kept in mind when choosing what to work with.

Firstly, maybe the most important aspect, was choosing a game engine. None of the techniques I talked about are tied to any architecture, framework or programming language in particular. They offer shortcuts, yes, but in the end, what we've seen are conceptual ideas. The point of working with a game engine, is that it offloads a lot of the work one would have to put into making gameplay more complex than using just text. I had a lot of ideas on how to approach the project, and so, I ended up choosing an engine that would suit all my needs.

Secondly, when it comes to writing Ai, a statically typed programming language would produce harder to read code. Godot's main programming language, GDScript, is dynamically typed, and works similarly to Python, with its own twist on many aspects. This has the advantage of making it easier for the programmer to write the kind of modular and dynamic code that AI requires. However, we trade speed, which is usually undesirable, for this cleanliness. Nonetheless, the project is meant to be understood by a wider audience, and GDScript is not too hard to read, bonus points for those that know how python works.

Finally, although not initially taken into account, the fact that the software is open source came in handy multiple times, as there were quite a few instances where I had some difficulties, and resorted to recompiling the source code in order to fix them. We will talk more about these difficulties and other challenges later on, but I considered this an important feature for which I was very glad to have.

4.6.2 Run Instructions

In order to run the project, download Godot 4.0 from their official website and download the project from GitHub. Open the Godot application, and click on import, then browse to the folder where the project has been downloaded and select the *project.godot* file. The project "Thesis Game" should now appear in the list (if there are too many projects, scroll until you find it), then press on "Edit" if you want to take a deeper look at the code and play around, or run the game directly by pressing "Run".

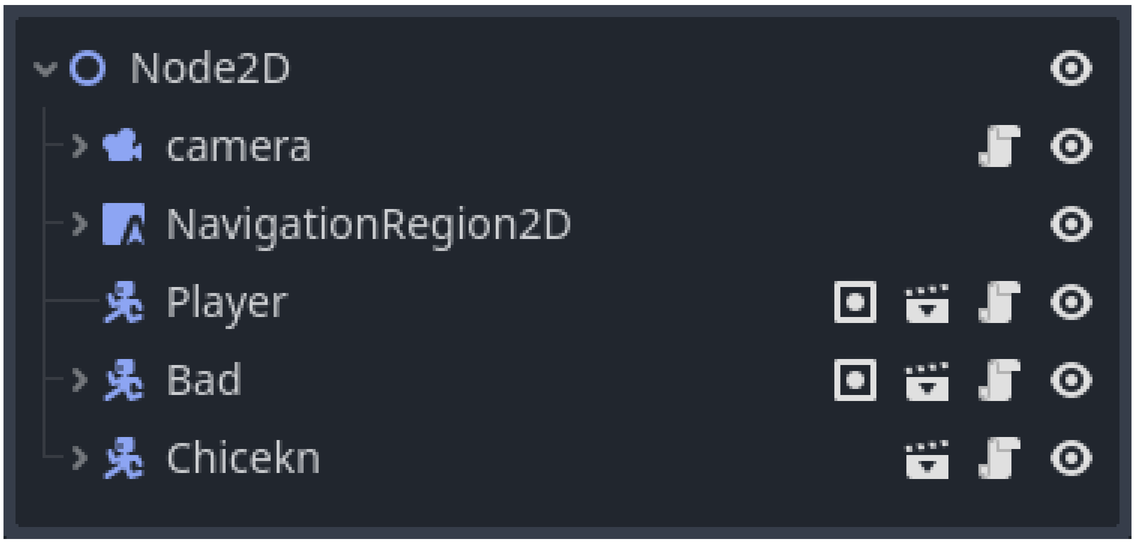
4.6.3 The project

Godot makes use of nodes to structure its projects. Nodes are used everywhere, and by architecting a hierarchy, we can achieve anything we want. There are dozens of available options, some are meant to replace commonly used code (such as the timer node, or remote transform), others are meant to provide better control over the game (such as container nodes for structuring UI elements, or Area nodes).

A collection of nodes, that can be independently deployed or used as a basis, is usually referred to as a scene. Scenes can contain for example: an entire game level, a character, an asset, particle effects and many more. This provides the advantage of making the hierarchy less cluttered, and isolating complex objects.

The project hierarchy contains five scenes.

1. The main scene, where we have the map, actors and props.
2. The player scene, which serves as the basis for NPCs, but also works on it's own
3. The NPC scene, which contains all the necessary features that enable the NPC to sense the world around itself
4. The item scene, which contains a few nodes for collision and spawn logic
5. The chicken scene, which contains the logic that controls a chicken



```
Node2D
  camera
  NavigationRegion2D
  Player
  Bad
  Chicekn
```

Figure 4.2: Snippet of the main scene, shortened, as seen in Godot

Main Scene

The main scene is made up of 5 collections as seen in 4.2:

- The camera, which contains nodes needed for the UI elements as well as a pair of "ears"
- The navigation region, which contains all necessary nodes to create the A* grid, patrol points, delivery points, item spawn points and the tile map itself
- The player scene
- Multiple NPC scenes
- Multiple chicken scenes

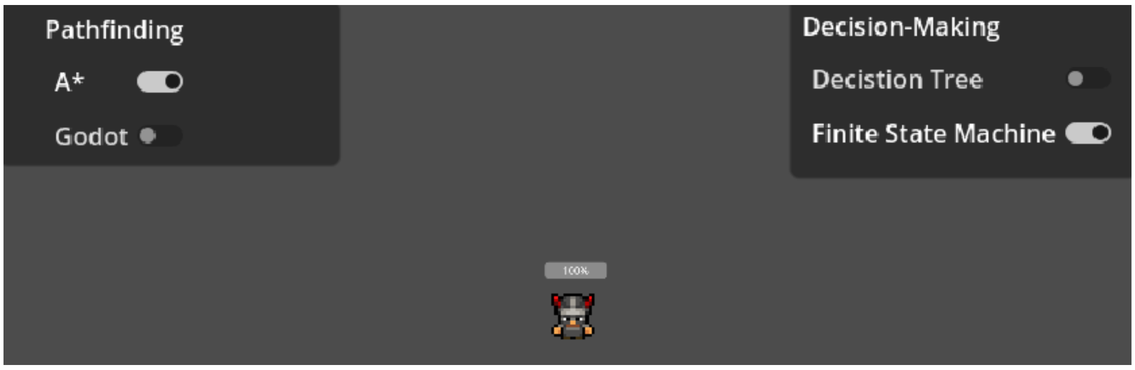


Figure 4.3: UI elements

The camera is always following the player. This is achieved through the use of a specialized node, call a remoteTransform2D, which simply pushes its transform (position, rotation and scale) to another object, that is attached to the player. There is a script attached to the camera (denoted in 4.2 through the paper symbol) that controls zoom and provides the player with the ability to switch between the available decision-making and pathfinding algorithm of all NPCs. We will discuss more about this in the techniques and algorithms section 4.6.4. The UI elements facilitate this behavior through the use of exclusive toggle buttons (i.e. only one of the pathfinding or decision-making algorithms can be used at any given time) as seen in 4.3.

The navigation region contains everything an NPC might need in order to walk around the world. This is achieved through the use of a feature that was improved in Godot 4.0, the tilemap editor, as well as other nodes that pose as either patrol points, or delivery points for items. In addition, there are items scattered around the world around patrol points.

Tile maps are very powerful when correctly set up, as they allow one to transform a set of tiles (tileset) into a fully fledged world. Using this/ tool, we can add navigation polygons specific to each tile, and when we place and connect these tiles, we are both generating the world, and it's navigation mesh. In our case, only the road tiles contain these meshes, so NPCs can only navigate on these tiles, as opposed to the player, who is free of all constraints. Due to this, there was very little code needed to implement the Godot pathfinding algorithm. The A* grid uses a similar approach, although we cannot use the navigation polygon itself, and such, I had to resort to read the entire tile map and create the grid in memory.

Finally, we have chickens. They do not do much, just move at random intervals, then idle.

The Actor

This is both the template for all humanoid NPCs and the player character. It contains only the bare minimum that allows player to control. It contains a *CharacterBody2D*, a type of specialized node that provides various functions and fields that make it easy to implement and control a game character. Its children contain various nodes regarding sounds, animation, model, and collision boxes. The player logic is controlled through two scripts: *Actor.gd*, *animator.gd*.

The actor script contains all the logic pertaining to all humanoids. The difference between an NPC and a player is a single boolean statement. The code is split into player and AI logic, with player logic processed inside since it's smaller and AI logic being delegated to another node, called a *Decider*.

When the actor is spawned, bet it a player or not, we choose a random model for each. If it's a player, we also target the remote transform towards the main camera. While processing, we check that our health is above 0, and if so, we listen to player input as described in 4.4. If an input was received, we update the currently playing animation, and play a sound effect where applicable. Additionally, if a humanoid is hit, they will play another animation, and will not be able to attack during it.

Additionally, the actor itself, and implicitly all NPCs, have a *hitbox* and a *hurtbox*. The hitbox is used when we attack, and when this interacts with the hurtbox of our opponent it will substract some amount of health.

The Bad Actor

The bad actor is the alter ego of an actor, and contains additional nodes that allow it to sense, navigate and think. It contains much more logic than a normal actor, and it is split into multiple scripts. Besides the processing that's happening inside *Actor.gd*, we have *Decider.gd* that deals with all decisions, *Navigator.gd* to deal with navigation, and $A+Agent.gd$ ("A*Agent" was not a valid name on Windows!) which is the custom implementation of A* made by the author.

This is by far the most complex part of the project, since there are hundreds of line of code making up the simple behavior of the in-game NPCs.

First, we have the processing that is happening in the main script, which for the NPC updates its internal world state, and passes it to the decider. This kind of implementation makes use of the concept of a blackboard, a dictionary type object in which we can store any kind of [key, value] pair. Although not necessarily efficient, it allows us to skip on processing the input, which would have been the case if we passed around integers instead.

The decider also keeps track of which pathfinding or decision-making technique we use. For the former, there isn't any distinguishable difference in code, and for decision-making the logic is clearly separated.

Secondly, we have the passive processing that is happening around the NPC. The NPC makes use of various sensors to gather data from the world. These are : vision, hearing and touch. These sensors will trigger a specific function in code, through which we can learn about the object we are sensing.

And finally, we have the pathfinding processing. The NPCs are almost constantly moving, so we need to make sure they know where to go, and how to reach that position. By default, all of them are patrolling, and they choose their patrol points at random. Once a point is selected, we check with our navigation agent if we have arrived, otherwise, we set the target (if it hasn't been already set) then ask it to generate a path. If we are using Godot's navigation system, that is where the story ends. If we are using our A* implementation, we ask the A* Grid for the map, and then we compute the path based on what it.

Listing 1 Part of the decision tree

```
if blackboard.get("target"):
    #have we arrived close enough to our target?
    #Patorl/Hunt
    if PursueTarget(actor,blackboard["target"],
            ↪blackboard["speed"]) == 0:
        #is this target a delivery point?
        if blackboard.get("hasItem") and
                ↪InGroup(blackboard["target"],"delivery"):
            actor.itemUnPick()
            #Idle
            blackboard["target"] = actor
            await get_tree().create_timer(randf_range(1,10)).
                ↪timeout
            blackboard["target"] = null
        #is this something we can attack?
        elif InGroup(blackboard["target"],"targetable"):
            #is it a player?
            if blackboard["target"].player and not
                    ↪blackboard.get("hasItem"):
                #Attack
                Attack(actor,blackboard["target"])
            #target's dead
            if blackboard["target"].health <=0:
                blackboard["target"] = null
else:
    #Patrol
    if PatrolPoints:
        var pointCount = PatrolPoints.size()
        var patrolPoint = PatrolPoints[randi()%pointCount]
        blackboard["target"]= patrolPoint
```

4.6.4 The techniques and algorithms

The Decision Tree

Based on their definition, decision trees can be coded as a series of if/elif/else statements. As I have started defining the NPC behavior, it was much easier to keep adding instructions and expand the flow control structure. In the end, this became the basis of the finite state machine as well. A diagram for the main decision tree can be found in the Decision Trees subsection 3.2.4 here 3.5.

The whole AI logic can be found in the *Decider.gd* script, under the $simple(actor, blackboard)$ function. A part of this logic can be seen in Listing 1 1. We initially check if we can retrieve a target, and if we've reached it, we have multiple choices. If the NPC is carrying an item, and the target is a delivery point, then drop the item and idle for a bit, then discard the current target. Otherwise, if our target can be attacked, and we are not carrying an item, and it is a player, we attack. Finally, a target with no health can no longer be referenced, so we discard it.

Listing 2 Finite state machine; state selector and part of the patrol state

```
enum States {Idle,Patrol,Hunt,Attack,Talk,DeliverItem}
var currentState:States = States.Idle
func finite(actor,blackboard):
	match currentState:
		States.Idle:		Idle(actor,blackboard)
		States.Patrol:		Pat(actor,blackboard)
		States.Hunt:		Hunt(actor,blackboard)
		States.Attack:		Attk(actor,blackboard)
		States.DeliverItem:	Deliver(actor,blackboard)
```

```
func Pat(actor,blackboard):
	if blackboard.get("itemClose") and not
	    ↪blackboard.get("hasItem"):
		currentState = States.DeliverItem
	if blackboard.get("target"):
		if InGroup(blackboard["target"],"targetable"):
			currentState = States.Hunt
		if PursueTarget(actor,blackboard["target"],
 ↪  blackboard["speed"]) == 0:
			currentState = States.Idle
			blackboard["target"] = null
```

If a target cannot be retrieved, then we choose a random patrol point as our new destination. It should be noted that the function `PursueTarget(x,y,z)` deals with all navigation calls.

The Finite State Machine

The finite state machine used for controlling the AI has been entirely modelled after the decision tree. Although there are a few differences in regard to how the function calls are made, and the introduction of state changes, the logic has been kept mostly identical. In Listing 2 2 we can see how the FSM manages calls between states, as well as how states are selected inside the patrol function. The overall logic has also been exemplified in the Finite State Machine subsection 3.2.4 here 3.6.

It should be noted that the finite state machine code isn't an exact replica of the decision tree logic, and as such there are a few minor differences between the two. The main one regards the state selector exemplified in the first part of Listing 2. The other important difference is that, in fact, each state is a decision tree.

Pathfinding

In addition to the decision-making algorithms, the game also features two types of pathfinding techniques : Godot's built-in Navigation system and A*. Both algorithms use A* at their core. The main difference is their implementation. I have used a lot of Godot's functions to implement the pathfinding algorithm, which means that I have the option to change multiple things regarding how the path is manipulated, as well as how the target is processed. Though my implementation lacks a lot of the rich features that the built-in system offers (notably the debug feature), it was a pleasant experience, and a successful effort on my part, to have made it work almost the same as the original.

Listing 3 PursueTarget function

```
1  func PursueTarget(actor:Node2D, target:Node2D, speed:float):
2      var targetGlobal = navigator.GetGlobalPosition(target)
3      navigationSpeed = speed
4  
5      var Agent = navAgent if NavMode.NavAgent == NavigationMode
6              ↪else astarAgent
7      if Agent.get_target_position() != targetGlobal:
8          Agent.set_target_position(targetGlobal)
9          
10     var arrived = navigator.Navigate(actor,
11             ↪navigationSpeed, NavigationMode)
12     if arrived == 0:
13         actor.moveDir = Vector2.ZERO
14     return arrived
```

Godot Pathfinding

Godot offers quite a powerful tool for navigation in the for of Navigation Meshes and Navigation Agents. Although they are quite commonly used in various game engines, I was still surprised given how easy it was to implement.After creating the navigation mesh for each tile in the tile set, and adding those to the world, we attach a NavigationAgent2D node to our Bad Actor, and through a few calls, we can start walking towards our target, as seen in Listing 3 3 and Listing 4 4, we don't care about which navigation agent we use, but we still have to keep track in order to allow them to be switched, both at runtime and before compiling.

At line 10 we can see a call to `navigator.Navigate` which tells whether we have arrived or not to our destination. Inside `Navigate` we again make sure to use the correct navigation agent object. We then ask the agent to provide the next point in the path, which we use to calculate in which direction should the actor move.

This way of coding the navigation agent allows seamless expansion of the system, as long as we make sure to use our desired agent and our new navigation agent implements the same function calls.

Listing 4 Navigate function

```
func Navigate(actor:CharacterBody2D, speed:float,
            ↪mode=Mode.NavAgent) -> int:
	var result = -1

	#simple shorthand to use the appropiate node
	    ↪#when computing navigation
	var Agent = navAgent if mode == Mode.NavAgent else AStarAgent
	#code adapted from here
	#https://www.youtube.com/watch?v=-juhGgA076E
	var currentLocation = GetGlobalPosition(actor)
	#this is why we need to have identical calls to
	    ↪#the navigation agent
	#otherwise we would need a switch (match) structure
	    ↪#in order to call functions
	var nextLocation = Agent.get_next_path_position()
	var newDir = currentLocation.direction_to(nextLocation)
	#this is crucial, we need to set our actors moveDir
	actor.moveDir = newDir
	#euclidean distance to the target
	var distance = currentLocation.distance_to
	        ↪(Agent.get_target_position())
	#this might also cause issues with AStar?
	if  distance >= searchRadius:
		result = 1 # running
	else:
		result = 0 #stopped
	return result
```

A* Pathfinding

Given that we are using identical calls for any navigation agent, the question now becomes, how do we implement one ourselves? One could take a peek inside Godot's code base, look up the appropriate functions, and then implement a similar logic. In our case, the folks over at Godot have provided an example implementation of a navigation system using A* through the 2D Navigation Astar Demo scene. I have used this scene in order to better understand how to make use of the tools Godot provides for A*.

The script *A+Agent.gd* contains the entire implementation of the A* navigation agent. All functions are eponymous to Godot's. This means that, as mentioned previously, the navigator doesn't care who is the current navigation agent. There isn't any code that discerns calls between the two, we always make the same calls to either agent!

Writing this implementation took a bit of trial and error to get right. Firstly, we cannot use the navigation polygons directly, since we are using a grid instead. Secondly, we have to make sure that we are only updating the path as we go along,

Listing 5 Get next path function

```
func get_next_path_position() -> Vector2:
	#we might lose some precision with this, but hopefully it's
	        ↪#not too much
	var localActor = Vector2(tilemap.local_to_map(actor.position))
	var next_point = tilemap.local_to_map(previousPoint)
	#are there at least two nodes in our path, and are we really
	        ↪#close to the next one?
	if (path && path.size() >= 2) and
	        ↪localActor.distance_to(Vector2(next_point)) < 0.1:
		#get rid of the first node in the path
		path.remove_at(0)
		#set the 'next" element as our new target
		next_point = path[0]
	#keep track of the previous point we visited
	previousPoint = tilemap.map_to_local(next_point)
	#use this only when there's one NPC,
	#this allows the visualisation of the path to some extent
	#debugTile.position = previousPoint
	return previousPoint
```

and not every call. And finally, we need to take care about our coordinates system, as the grid coordinates are loosely dependent on the global position of a tile.

In Listing 5 5 we can see the implementation of `get_next_path_position()` as seen in the project. Throughout this script, there are multiple calls to the tilemap object's functions `local_to_map and map_to_local`. These two allow us to correlate the global position of a tile to its local map coordinate. Since the A* grid is based on the map coordinates system, we need to process all positions so that an outsider will provide and receive only positions in the global space, whilst internally, we process on the map system. Internally, the agent also keeps track of the actor, and calculates the distance to the next point in the path. It should be noted that the path is computed whenever we change the target, as seen in Listing 3 3 in the `PursueTarget` function, in our implementation the path is computed as soon as `Agent.set_target_position` is called.

Chapter 5

Scrapped Ideas

The project went through multiple phases, spanning a little short of two years. Initially, there was an idea to make a hunting game, entitled "Fox Hunt", where the player would be the fox. The idea didn't get too far, as I've become doubtful of its validity in the long-term. The AI was supposed to use tactics and various strategies in order to corner the player, learn some of its behaviors and adapt. It then suddenly dawned on me that it might take too much time.

Before making the current project, I have started a few smaller projects in order to familiarize myself with Godot. Only one of these made it far enough, and I made full use of what I've learned. It was quite ugly, and very rough, but it had a functional AI and it could pathfinding around the map. That project became the basis for how decision-making should be implemented, and after learning about Godot's A* tools, the previous pathfinding became irrelevant.

In addition to the features related to code, there were also a few subsections cut regarding learning algorithms, drama managers, an entire section of implementations dissected. The implementations section was supposed to include code from the original half-life, possible implementations for the OpenAI Five bots, as well as various GDC talks for Rain World's ecosystem, Dwarf Fortress, Ultima Ratio Regum, and many more. This part was cut because of the exponential increase in research needed and the limited time left for finishing the thesis.

Everything was smooth, the app was finally starting to take shape, the countless days and nights spent building it were finally showing their colors.At some point, I realized that I had no backup of the code, which spelled disaster. And indeed, just before uploading the code to GitHub, some essential project files got corrupted. Thankfully, I was able to track down the issue after a few hours of frantic queries to Google and DuckDuckGo, and I was especially grateful for the fact that Godot was open source,as I could tinker with the files directly, and I did this multiple times throughout the project, even to the point of recompiling and rebuilding the entire executable because I had a few bugs which were not yet addressed. Quite an adventure!

Chapter 6

Challenges

There were a lot of factors that have slowed down the progress of this paper. It took longer than expected to write the actual code, both because of my unfamiliarity with GDScript and specific AI algorithms. Writing the code was not an easy task, as bugs creeped at every step, and making sure that every transition, decision, or goal. As mentioned in the previous chapter 5, there were multiple missed shots before finally hitting the target with the current project.

Due to the nature of developing the project,FSM's were tricky to implement as a lot of the logic had to be made into separate functions, and actions had to be delegated and moved, in the end resulting in a harder to read code.

Another challenging part was writing the A* implementation. Both resources and tutorials on how to make use of Godot's tools were not as widely available, and the documentation didn't exemplify too much. Only the demo from Godot regarding the AStar2D navigation proved to be a fruitful resource.

All in all, this thesis proved to be quite a challenge to write, both because I had to write such an extensive work, which was a first for me, and also because of the amount of research I had to do in order to accurately gauge this subject. It didn't help that I wasn't very familiar, at least in technical terms, to the field of AI in video games, and even AI in the academic sense, aside from the AI course I took during my final year. But when I decided to pick this subject last year, I thought it would be easy as I was well acquainted with video games in general, so it made sense that I shouldn't have problems with such a task. Given that it took me almost two years to write this paper (out of which one and a half was spent stressing over what should be included), it has proven to be the Goliath of my story. But I can't say I didn't enjoy it, as I finally got to learn a lot of new things regarding a domain I was so accustomed to, I took everything that came my way for granted, and video games do have their fair share of difficulties in designing and making, which I will explain briefly.

During the development of the first project, which was done whilst still in the 3rd year, I had trouble coming up with a theme, since I had a lot of bad experiences regarding demos and unfinished projects. The first draft failed miserably, both because I wasn't capable of learning how to use Unity and its tool-chain, and also because I was working in between courses. Not great!

The second time, was around the start of this year, where I had moderate success, but the code was made out of snippets from various sources. I didn't have a problem with pointing out where each snippet came from, the issue was that I couldn't understand why some pieces wouldn't fit together, and so I started a new one, the project currently attached to this paper.

The paper took quite a lot of effort, juggling between work life and personal life, whilst trying to write was quite challenging in and on itself. I wouldn't recommend anyone this kind of work environment, but it allowed for more time to ponder on various subjects, and implicitly, the expansion of multiple sections throughout this paper's development, which I took as bonus!

Chapter 7

Conclusion

A lot of what it means to develop AI for video games is about trickery, about speed and believability. Trading accuracy for speed, we have a lot of options to achieve any kind of result one might desire. From the use of Decision Trees to model simple behaviors, to Goal-Oriented Action Planning and it's promise of complex characters, there isn't any certainty that in the future, another type of algorithm or technique might be invented, but there is still the possibility of something great happening.

The main scope of this entire paper was to be an entry point for those enthusiastic enough to learn about what does AI imply when it comes to video games, and it is my sincerest hope that this paper provided you, the reader, with enough information to start your own journey. This paper would have served my past self in writing an even better paper, were this kind of compact treasure available to me at the time I started researching.

I have gained a lot of knowledge whilst reading the material I gathered, I stumbled upon a lot of things that I would consider treasures, and in my pursuit of compiling my research, I hope I have created a work worth reading. As AI is everywhere nowadays, video games are bound to evolve, many of the techniques and algorithms presented in this paper might become irrelevant in the future, but this work should serve as a checkpoint in time, to commemorate and record the achievements of AI in video games up to the present moment.

Bibliography

[1] https://github.com/JosephPrichard/Pathfinder/blob/main/src/pathfinding/algorithms/Heuristics.ts.

[2] The artificial intelligence for interactive digital entertainment conference. https://aaai.org/conference/aiide-2/.

[3] Astargrid2d. https://docs.godotengine.org/en/stable/classes/class_astargrid2d.html#enumerations.

[4] Game developers conference. https://www.youtube.com/@Gdconf.

[5] Godot. https://godotengine.org/.

[6] Unity. https://unity.com/.

[7] Thomas Holmes. Defining voice design in video games. Master's thesis, Aalto University. School of Arts, Design and Architecture, 2021.

[8] M. Hutson. *The 7 Laws of Magical Thinking: How Irrational Beliefs Keep Us Happy, Healthy, and Sane.* Hudson Street Press, 2012.

[9] Darren Korb & Greg Kasavin. Breathing life into greek myth: The dialogue of 'hades'. https://www.youtube.com/watch?v=m5KJSAj4afg, 2021.

[10] John-Paul Kelly, Adi Botea, and Sven Koenig. Offline planning with hierarchical task networks in video games. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 4(1):60–65, Sep. 2021.

[11] Tom Leonard. Building an ai sensory system: Examining the design of thief: The dark project. https://www.gamedeveloper.com/programming/building-an-ai-sensory-system-examining-the-design-of-i-thief-the-dark-project- 2003.

[12] Ian Millington and John Funge. *Artificial intelligence for games.* CRC Press, 2018.

[13] Jeff Orkin. Three states and a plan: The a.i. of f.e.a.r. 2006.

[14] Jeff Orkin. Applying goal-oriented action planning to games. In Steve Rabin, editor, *AI Game Programming Wisdom 2.* 2008.

[15] Joseph Prichard. Pathfinder. https://github.com/JosephPrichard/Pathfinder.

[16] Steve Rabin. *AI Game Programming Wisdom.* Charles River Media, Inc., USA, 2002.

[17] David Rosen. Animation bootcamp: An indie approach to procedural animation. `https://www.youtube.com/watch?v=LNidsMesxSE`, 2014.

[18] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach, Global Edition.* Pearson Education, 2021.

[19] J. Schell. *The Art of Game Design: A Book of Lenses, Third Edition.* CRC Press, 2019.

[20] Jaroslav Švelch. Should the monster play fair?: Reception of artificial intelligence in alien: Isolation. *Game Studies*, 20(2):243–260, 2020.

[21] Penelope Sweetser and Janet Wiles. Current ai in games : a review. *Australian Journal of Intelligent Information Processing Systems*, 8(1):24–42, 2002.

[22] Tommy Thompson. Ai and games. `https://www.youtube.com/@AIandGames`.

[23] J. Wang. *Formal Methods in Computer Science (1st ed.).* Chapman and Hall/CRC, 2019.

Video Games

[1] SCS Software. Euro Truck Simulator 2. SCS Software, 2012.

[2] Jackbox Games, Inc. The jackbox Party Pack. Jackbox Games, Inc., 2014.

[3] Codemasters. Dirt Rally 2.0. Codemasters, 2019.

[4] Rockstar Games. Grand Theft Auto V. Rockstar Games, 2013.

[5] Michael Mateas & Andrew Stern. Facade. Procedural Arts, 2005.

[6] Steve Russell. Spacewar!, 1962.

[7] Atari. Pong. Atari, 1972.

[8] Namco. Pac-Man. Namco, 1980.

[9] Looking Glass Studios. Thief: The Dark Project. Eidos Interactive, 1998.

[10] Valve. Half-Life. Valve, 1998.

[11] Bungie. Halo: Combat Evolved. Microsoft Game Studios, 2001.

[12] Monolith Productions. F.E.A.R. Vivendi Games, 2005.

[13] Creative Assembly. Alien: Isolation. Sega, 2014.

[14] Videocult. Rain World. Adult Swim Games, 2017.

[15] Bethesda Game Studios. Fallout 3. Bethesda Softworks, 2008.

[16] Wolfire Games. Overgrowth. Wolfire Games, 2017.

[17] Supergiant Games. Hades. Supergiant Games, 2020.

[18] Arkane Studios. Dishonored. Bethesda Softworks, 2012.

[19] High Moon Studios. Transformers: Fall of Cybertron. Activision, 2012.

[20] Maxis Software. The Sims. Electronic Arts, 2000.