

Uppsala University
Department of Informatics and Media

Decision-making AI in digital games

By Ahmad Al Shehabi

Information Systems: Master's Degree Project

Semester: VT2022

Date of the presentation seminar: 24 May 2022

Abstract

The field of artificial intelligence has gained much knowledge through the implementation of decision-making systems in video games. One of these systems was the Goal Oriented Action Planning system (GOAP) which directs the behavior of an AI-agent through multiple digital artifacts categorized as goals, actions, and plans. The aim of the thesis is to aid in the understanding and creation of GOAP driven AI-agents in a video game setting to promote research on this topic. The research question of this thesis was about finding out how the GOAP architecture compares to other video game decision-making systems. The theoretical framework introduces the concept of the illusion of intelligence in video games and presents a discussion focused on the different components which make up a GOAP system and other components that support it. Additionally, the theoretical framework explains the need for a comparison between different decision-making systems and explains the social impact of game AI research. The methods section introduces the criteria for the comparison between GOAP and other decision-making systems and presents a comparison process that was driven by a literature review. A GOAP system was designed for this thesis using the unified modeling language and concept maps. It was then implemented using C# code in a free-of-charge game engine called Unity. We present the pseudocode for the implementation of the GOAP system and show that this framework is a modular, customizable, and reusable system that enables AI-agents to create plans from a varied set of actions. Finally, the paper suggests further research within game decision-making AI and emphasizes the importance of game AI research for communities of game developers, hobbyists, and others who could benefit from game AI in their projects.

Keywords

Decision-making, Goal Oriented Action Planning, GOAP, Agent, AI, Artificial Intelligence, Behavior Trees, Finite State Machines, FSM.

Table of Contents

| | |
|---|----|
| Table of Contents | 3 |
| List of Tables | 6 |
| List of figures | 6 |
| Chapter 1 Introduction | 7 |
| 1.1 Aim | 8 |
| 1.2 Research question | 8 |
| 1.3 Disposition | 9 |
| Chapter 2 Background | 10 |
| 2.1 The significance of Decision-Making systems in video games | 11 |
| 2.2 Relevance to the field of information systems and contribution to the field | 12 |
| Chapter 3 Existing research | 14 |
| 3.1 Finite State Machines (FSM) | 14 |
| 3.2 Behavior trees | 16 |
| 3.3 Goal Oriented Action Planning | 16 |
| 3.4 The research gap | 17 |
| Chapter 4 Theoretical framework | 19 |
| 4.1 The significance of the illusion of intelligence | 19 |
| 4.2 Creating the illusion | 20 |
| 4.3 Planner system, A* algorithm and goal-driven (regressive) search | 21 |
| 4.3.1 The planner | 22 |
| 4.3.2 The A* algorithm and costs per action | 23 |
| 4.3.3 Heuristics and the set of relevant actions | 23 |
| 4.4 Vision in video games | 24 |
| 4.5 Working memory for AI-agents | 26 |

| | |
|---|----|
| 4.6 Different decision-making systems..... | 27 |
| 4.7 The social impact of game AI research..... | 28 |
| 4.8 Summary of the theoretical framework | 29 |
| Chapter 5 Methods and empirical data | 30 |
| 5.1 Design Science Research | 30 |
| 5.2 Analysis and Comparison | 31 |
| 5.3 Literature review | 33 |
| 5.3.1 Defining the alternatives | 33 |
| 5.3.2 Analyzing the alternatives..... | 34 |
| 5.3.3 Asking questions about the alternatives and making a result sheet | 36 |
| 5.3.4 Defining pros and cons | 37 |
| 5.4 Concept maps..... | 37 |
| Chapter 6 Ethics..... | 39 |
| Chapter 7 Presentation of results | 40 |
| 7.1 Project creation in the Unity engine | 40 |
| 7.2 Unified Modeling Language (UML) | 41 |
| 7.3 Implementation | 44 |
| 7.3.1 The Planner | 45 |
| 7.3.2 Goal..... | 47 |
| 7.3.3 Action..... | 48 |
| 7.3.4 Plan | 49 |
| 7.3.5 ActionSet..... | 50 |
| 7.3.6 World Representation | 51 |
| 7.3.7 Agent..... | 53 |
| 7.3.8 VisionSensor | 55 |

| | |
|---|----|
| 7.4 Comparison results..... | 58 |
| 7.4.1 Analysis of the comparison..... | 61 |
| Chapter 8 Discussion | 65 |
| 8.1 Summary of the work as a whole..... | 65 |
| 8.2 AI in games in design science – process and obstacles | 68 |
| 8.3 Ease of access to game development research | 71 |
| Chapter 9 Conclusion | 73 |
| 9.1 Main conclusions | 73 |
| 9.2 Limitations and future work..... | 74 |
| Chapter 10 References | 76 |

List of Tables

| | |
|--|----|
| Table 1: a table showing the requirements for the decision-making system that was developed | 32 |
| Table 2: table of analysis of alternatives..... | 35 |
| Table 3: Questions and results about the alternatives | 37 |
| Table 4: “bad, good and best” comparison table of the alternative decision-making systems..... | 61 |

List of figures

| | |
|---|----|
| Figure 1: The ghost’s FSM from the classic Pac-Man game. Adapted from figure 5-1 in (Source: Cossu, 2020b, p117-p139) | 15 |
| Figure 2: Vision cones. Green check marks show which targets can be seen and red crosses show which targets cannot be seen by the agent | 25 |
| Figure 3: Concept map of relations between the agent’s different components..... | 38 |
| Figure 4: UML diagram of the developed GOAP system | 42 |
| Figure 5: a UML diagram of the relations between the VisionCone and the Agent's WorkingMemory..... | 43 |
| Figure 6: the project’s SearchActions method from Planner class | 47 |
| Figure 7: Two scriptable action instances in the Unity engine’s interface | 49 |
| Figure 8: The hunter’s action set in the Unity engine’s interface..... | 51 |
| Figure 9: The Memory struct from the project | 53 |
| Figure 10: A code snippet that shows the method GrabKey from the PlanExecutor class..... | 55 |
| Figure 11: A “Hunter” character with two vision cones..... | 57 |
| Figure 12: a concept map of the relation between the vision sensor, working memory, planner, and the blackboard | 58 |
| Figure 13: a screenshot from the project showing a “hunter” AI-agent chasing a “hunted” AI-agent..... | 66 |
| Figure 14: a screenshot from the project showing a “hunted” AI-agent finding and grabbing a key..... | 66 |

Chapter 1 Introduction

This section introduces the reader to the general area of study that this thesis paper is about and explains the aim of the thesis in relation to the *Goal Oriented Action Planning* architecture. Furthermore, the research question of this paper is introduced in this chapter. Lastly, the disposition of the research paper is made available in the last part of this chapter.

The field of artificial intelligence (AI) is a large field that contributes to many industries and has many uses. AI has been used in facial recognition, search engines, robotics and even video games. One strange observation that some academics have noticed about people's view of AI is that once a certain kind of AI becomes mainstream, people stop viewing it as real intelligence (Haenlein & Kaplan, 2019). However, just because a certain AI technology is no longer deemed intelligent does not mean that it is any less useful. Video game AI is one of these AI fields that has become mainstream and is often expected to be part of any modern video game. But programming AI requires structure. It is not as simple as telling the computer to do "A" if condition "X" is fulfilled. Instead, a more organized system should be engineered to protect the AI's code from becoming unmanageable. One AI architecture that has been used in some video games is the *Goal Oriented Action Planning* architecture (GOAP). This thesis studied the GOAP architecture (Orkin, 2003) for AI decision making in video games and was accompanied by a development project of a GOAP system that was implemented in a digital game prototype. This was important because there is not enough accessible research about different AI-systems for game development. Smaller game studios and game development hobbyists do not have large budgets or enough time to do their own research and therefore, the game AI research community has a responsibility towards these kinds of developers (Cook, 2021). Additionally, the GOAP architecture (Orkin, 2003) is not as popular as other types of AI systems such as behavior trees and FSMs (Sweetser & Wiles, 2002), therefore it is also not as well documented either. Because of that, this research presents a valuable contribution to the field of game AI research.

1.1 Aim

The focus of this thesis was to understand the GOAP architecture (Orkin, 2003) and how it is built and then program a prototype of a GOAP driven AI-agent to be used in a digital game environment. By creating a prototype that uses GOAP AI and documenting that development process, this thesis aimed to advance the research within the fields of artificial intelligence and game development and make GOAP a better understood system with publicly accessible documentation. The chosen game type that was used was made to be similar to the basic mechanics of the game *Dead by Daylight* (2016). This meant creating a game prototype that had a type of character whose goal was to hunt other characters and another type of character whose goal was to escape. These types will be described as the hunter and the hunted in this paper. The prototype developed for this thesis used the GOAP architecture to create AI players that played against each other as opposed to having an online multiplayer game with real human players as is the case in *Dead by Daylight*. The reason for this choice of game category was that it presented two kinds of players with two main goals, hunt, or escape. This meant that the number of goals that the GOAP agent had, could be narrowed down to a smaller amount to fit the scale of this thesis paper and project.

1.2 Research question

The *Goal Oriented Action Planning* architecture (Orkin, 2003) is not the only decision-making type of system that is used in video game artificial intelligence. Therefore, it was important to understand its significance when compared to other systems that were used for similar purposes. The research question that this thesis wanted to answer was:

- How does the *Goal Oriented Action Planning* architecture compare to other artificially intelligent decision-making systems in video games?

1.3 Disposition

In Chapter 2 we present a background on the topic of Goal Oriented Action Planning. In Chapter 3 we cover existing research in the field of game decision-making systems. Chapter 4 is a theory-based chapter in which the theories around pertinent topics in artificial intelligence and video games are presented. Chapter 5 is the methods and empirical data chapter where the different methods that were used are presented followed by a brief chapter on ethical concerns about this research paper. Finally, in Chapter 7 we present the results of the research paper and provide a discussion in Chapter 8 and the conclusion of the research paper in Chapter 9.

Chapter 2 Background

This chapter briefly explains the concept of the *Goal Oriented Action Planner* (Orkin, 2003) and why it is an improvement compared to other methods of game character AI. Additionally, a motivation for the need for this kind of AI is introduced in relation to programming efficiency and an explanation of its relevance to the field of information systems is presented.

Video games have used several standards for non-player character AI, ranging from simple ones to more complicated ones that are harder to code. One of the most common methods of creating game AI is by using (FSMs) finite state machines (Sweetser and Wiles, 2002). Sweetser and Wiles explained that FSMs are the most used AI technique in video games because of how easy they are to program and get running. They elaborated that the way that FSMs work is that they divide behaviors into a limited number of states that a game character can alternate between. However, FSMs that are poorly structured can become too hard to manage the bigger the project becomes. Sweetser and Wiles noted that bigger games might require many states to be handled at the same time and having too many states in an FSM can easily become a problem. The *Goal Oriented Action Planning* architecture is one solution to this problem. Orkin (2003) said that GOAP does not replace the need for an FSM, but rather simplifies it so that it can be managed more easily. Orkin explained that in GOAP, instead of making a new state for each action in the FSM, a game character's actions can be grouped together in a state. That way, the total number of states in an FSM remains smaller. For example, he said that actions such as *dodge* and *reload weapon* can be grouped together in one state called *animate*. Furthermore, Orkin stated that the GOAP architecture helps create characters that are able to create plans and exhibit unique artificially intelligent behaviors. The characters become less repetitive and less predictable as he puts it.

Games that use AI to create game characters that are able to intelligently hunt, and chase down other game characters have seen a rise in popularity in recent years. Examples of those are the Alien from *Alien: Isolation* (2014) which chases the player around the game area and reacts to the player (searches for the player if they are hiding and attacks if they are visible) as well as the

characters Mr. X (Resident Evil 2, 2019) and Lady Demetrescu (Resident Evil Village, 2021) who exhibit similar behavior as they search for the player when they are hiding and chase them all through the game environment once they see them. This modern popularity of what could be called a *Hunter AI* in video games could make use of the GOAP architecture to enable creating intelligent hunter behaviors that can alternate between different states and create organized decision-making plans. Doing research on how this kind of AI can be created will help make this area of game development more accessible. This could be beneficial for smaller game studios that do not have big budgets to do research on AI for their games and would rather learn from publicly accessible thesis papers on the topic.

2.1 The significance of Decision-Making systems in video games

A game programmer could write long lines of code where they define the conditions for making a decision using “if this then that” lines of code. That could work perfectly fine for a game where the decisions to be made come from a small set of actions. Think for example about a game where an AI-agent needs to choose between the actions “Sleep” and “Eat”, where the need to sleep is directly related to the variable “Tired” and the need to eat is directly related to the variable “Hungry”. The logic to make a game character make a decision about those needs could look as such:

```
if(Hungry = true & Tired = false)
    //Do action 'Eat'
else if(Hungry = false & Tired = true)
    //Do action 'Sleep'
else if(Hungry = true & Tired = true)
    //Do action 'Eat' first
    //Wait until finished eating
    //Do action 'Sleep'
```

The above pseudocode makes a game character eat as long as it is not tired and sleep as long as it is not hungry and finally it makes it prefer to eat first and sleep second if it is both hungry and tired. This logic could work perfectly fine for an extremely simple video game. However, in a game where many needs are part of the game and there are many variables to consider, programming the game using simple “if this then that” statements creates very long lines of

unmaintainable code. Imagine for example a game where an AI-agent has the same two previously mentioned needs. This time however, fulfilling the “Sleep” action requires not only the agent to be tired, but that they also own a bed and know where it is located in the game’s 3D or 2D-space. Additionally, the agent should also be programmed to know that if they do not own a bed, then they should sleep on the couch. And then the same conditions would apply for the couch (the game character must own one and know where it is). In a game such as this, the simple action of going to sleep has so many conditions that affect it. This also applies to the “Eat” action and many other actions that the programmer would want to program into the game. Furthermore, the programmer might want to create different priorities for each action. Going to sleep while very hungry would be bad for the game character, so they should prioritize eating first. Programming action-priorities would create even longer lines of code that would have to account for every single combination of actions and which action of each pair of actions needs to happen before the other.

The previous example illustrates the need for Decision-making systems in video games. A well-structured decision-making system can eliminate long and unmaintainable lines of code. The GOAP architecture (Orkin, 2003) does that by using a combination of digital artifacts called actions, goals, and plans. Actions have preconditions and effects which help the system make decisions regarding prioritization of actions. Furthermore, the system uses costs per action to simulate decisions regarding preference.

2.2 Relevance to the field of information systems and contribution to the field

Video game designers need to collaborate using different information systems to create their games. Information system research is concerned with the interactions between individuals, systems, and organizations (Hirschheim & Klein, 2012) and information systems are described as systems that process data and provide information. Viewing the Goal Oriented Action Planning architecture (Orkin, 2003) through the lens of information systems, it can be seen that GOAP provides a way for game designers, programmers, and technical artists as separate individuals, to each do their part in a way that utilizes GOAP as a system. Ultimately, working together as an organization to produce a video game. Additionally, GOAP can be seen as an

information system because it can be provided with data from the designers. That data is in the form of definitions of individual actions as digital artifacts of the system. The GOAP system then processes that data using its planner and provides information to an AI-agent in the form of a plan. Furthermore, more design science research could be done for the field of information systems according to Peffers et al. (2007). They explained that a design science research project entails creating a digital artifact, evaluating it, and presenting it to an audience. This means that following a design science methodology, a GOAP system can be designed as part of a research project that would contribute to the field of IS. Additionally, considering that information systems are widely understood as tools for problem solving in organizations (Lyytinen, 1987), it can be understood that a decision-making AI system such as GOAP solves a problem for game development organizations. That problem being a lack of a common structure of managing AI-agent actions, which causes unmaintainable code and projects. GOAP provides a framework for game developers to understand an AI-agent's behaviors as being motivated by goals, actions, and plans. The three important digital components of the GOAP system. Additionally, Peffers et al. (2007) explained that the field of information systems has struggled with applying design science research as a component of research in the field of IS. Thus, conducting a design science project to develop a GOAP system and evaluate it contributes to the field of information systems and benefits game development organizations.

Chapter 3 Existing research

There are several ways that a game character's AI can be programmed to influence its decision-making abilities. However, some of them can quickly become complicated and unmanageable. This section will discuss some of the existing research on game AI and decision-making.

3.1 Finite State Machines (FSM)

A finite state machine is a commonly used technique in video game artificial intelligence. Sweetser and Wiles (2002) explained that the commonality of FSMs in the field of game development could be attributed to their ease of programming and debugging. They revealed that FSMs are used to divide a game object's behavior into several parts. Each part helps exhibit a certain behavior. Furthermore, they said that an example of how an FSM could be used was to represent a monster that expresses different emotional states such as berserk, rage, mad, annoyed, or uncaring. Each of these states are activated by different events in the game and each state makes the character exhibit certain behaviors.

In his book about game-AI programming, Cossu (2020a) explained that the classic arcade game Pac-Man used an FSM where the ghosts that chased the player had 3 states. Roam, chase and flee. Each of these states was triggered by an event in the game and each of them could make a transition to the other. For example, as Cossu explained it, the ghosts would be in the *Roam* state when Pac-Man (the player) was not in range, and their FSM would transition to the *chase* state when they saw Pac-Man. Additionally, the FSM would transition to the *flee* state if the player had picked up a pill, which made the ghosts run away from the player. The illustration below (Figure 1) (Cossu, 2020b) explains the FSM of Pac-Man.

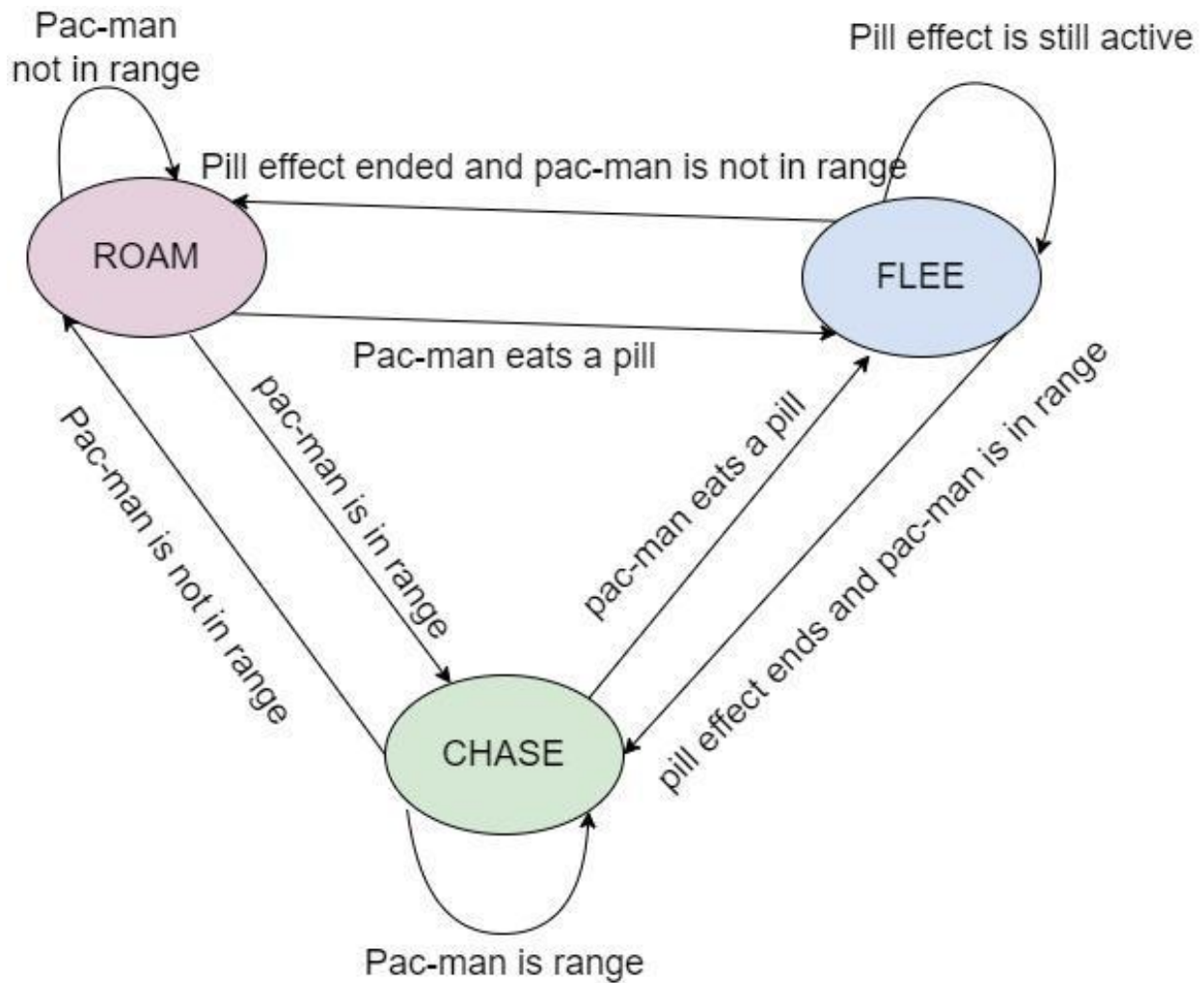


Figure 1: The ghost's FSM from the classic Pac-Man game. Adapted from figure 5-1 in (Source: Cossu, 2020b, p117-p139)

It can be seen in the illustration above how each state in the FSM uses three conditions to determine whether the FSM should stay in its current state or move to the next or previous states. A bigger game than Pac-Man can easily require more than 3 states which might imply the need to create a complicated FSM with a web of states, each requiring different conditions to transition from and to several other states. This is a well-known concern when developing game-AI using an FSM. Sweetser and Wiles (2002) said that game FSMs can become too large and grow out of control when needing to implement many states and state transitions.

3.2 Behavior trees

Another commonly used technique in video game artificial intelligence are behavior trees. Rabin (2017, Chapter 9) explained that behavior trees were a common architecture that is used to create control mechanisms for non-playable characters (NPCs). As he put it, a behavior tree has nodes that tell the NPC how to behave. Additionally, he claimed that behavior trees were easy to customize, because they were composed of smaller changeable behaviors each working as a small component.

The building blocks of a behavior tree (Rabin, 2014, Chapter 6) consisted of many components. The three most basic ones were behaviors, actions, and conditions. He explained that behaviors can be seen as an interface where actions and conditions create special implementations of the behavior interface. While actions were the leaf nodes of a tree, and they were responsible for gaining access to information about the game world and reacting to it. And finally, conditions were also leaf nodes in a behavior tree, which have the responsibility of checking for information about the game world.

Rabin warned however of some of the pitfalls of behavior trees. He said that because behavior trees were intended to aid in decision-making within larger systems, the tree could become too complicated if it is programmed to have too many responsibilities. Therefore, he explained that it is best to keep the following in mind:

- Avoid adding too many classes to the tree's decision-making architecture
- Do not build an entire programming language into the behavior tree prior to requiring such a feature
- Avoid using the blackboard as a point of communication for everything.

3.3 Goal Oriented Action Planning

The video game F.E.A.R (2005) was the first video game to use GOAP (Orkin, 2003) to plan decisions made by game characters. Orkin (2006) explained that the FSM for the characters in

F.E.A.R had only three states and that the A* algorithm was used to generate pathfinding plans as well as plans for sequences of actions. An A* algorithm is an algorithm which represents knowledge as a graph and finds the best path towards a certain goal in the graph (Hart et al., 1968). Orkin (2006) explained that the AI in this game had the ability to do things such as take cover, shoot fire blindly and even communicate with other AI teammates. The game used the states *GoTo*, *Animate* and *UseSmartObject* as Orkin explained. Furthermore, he elaborated that the *UseSmartObject* state was actually a version of the *Animate* state except that it holds an animation that is specific to the game object being used.

3.4 The research gap

The *Goal Oriented Action Planning* architecture for video game decision-making is a more advanced system compared to traditional finite state machines that are used in simpler video game AI. But there is a lack of sufficiently clear and easily accessible documentation and implementation examples of the GOAP system. Therefore, it is important to create an understanding of the GOAP architecture through a development and design project accompanied by good documentation and evaluation of the process and the developed artifact. Creating a GOAP system for any video game requires a planning process where the main components of the system must be outlined in order to direct the programming in the correct direction. There are some sources online that explain what GOAP is, but they do not show a clear overview of the system. Peffers et al. (2007) explained the design science research methodology which helps in doing research alongside a design project. They said this could be done by following some guidelines. They added that the most important of these guidelines was that the research must result in the creation of an artifact which acts as a solution for a problem. Furthermore, they said the artifact must be evaluated and the research should present a valid contribution while the development of that artifact must be based on previous research on the topic. Finally, they said that the research should be presented to a relevant community or audience. Therefore, to conduct a design science research project, this research paper will address that by developing a GOAP system which is the artifact of this research. To do that, it would be important to review the preexisting literature on the topic and create a clear blueprint that displays what the components

of the GOAP system is and how they interact with other objects in a game world. Additionally, each video game genre presents different challenges and uses for decision-making. To evaluate the produced artifact a comparison of the developed GOAP system to other decision-making systems is required. This helps in understanding when it is most useful to use GOAP and when to otherwise avoid using it and opt for a different decision-making system altogether. That will make this research paper a contribution to the field and the relevant communities of game developers.

Chapter 4 Theoretical framework

This section presents a theoretical framework for video game AI. Different concepts are explored. Some of which relate directly to video-game programming and others are more related to artificial intelligence in general. Together, these concepts are explained in relation to the GOAP planning system and how it all comes together as a video-game decision-making system.

4.1 The significance of the illusion of intelligence

An important part of video game AI programming is the illusion of intelligence as Rabin (2017, Chapter 1) explained. He said that even if an AI had human level intelligence, it could be perceived as unintelligent if it does not meet the player's expectation. Therefore, as Rabin explained, the illusion of intelligence in video games could be more important than real human-level intelligence. This point was further supported by Barrera et al. (2018) who stated that artificial intelligence for the purpose of video games is the illusion of intelligence. They added that intelligent game agents do not really need to learn things to be intelligent, they just have to convince the player that they are learning things. Additionally, they explained that intelligent game agents use different sets of sensors to react to their environment, similarly to how our brains use our eyes and ears but in a much less complex way. The reason why illusions work according to Rabin (2017, Chapter 1) comes from three aspects. The first aspect is that players want to believe in the illusion. Rabin said that players like to participate in the illusion that the video games they play have human-like qualities. He explained that as long as the AI does not make any visibly obvious mistakes, players will continue to participate in the illusion of the AI's intelligence.

The second aspect was that people are eager to anthropomorphize (Rabin, 2017). Rabin explained that people apply human traits to familiar behaviors to make sense of an ambiguous situation. Rabin believed that when this view is applied to video games, it can be said that people perceive an AI-agent's intelligence as human-like intelligence. Additionally, he added that video

games often have human-looking avatars that animate and move like human beings and therefore, the effect of anthropomorphism plays an important part in the illusion.

Finally, the third aspect according to Rabin (2017) was the power of expectation. He explained that people will believe certain things depending on what expectations they have. He gave an example from an experiment done by researchers in Caltech and Stanford where participants were given a 45\$ bottle of wine and a 5\$ bottle of wine and were asked to taste them. They used brain-imaging techniques and noticed that people's brains showed signs of experiencing more pleasure when the participants were tasting the more expensive wine. What the participants did not know was that both wines were actually the same. Rabin also added that the well known placebo effect in the medical field was another example of how people's expectations play an important role in what they experience. He concluded that managing player expectations in video-games was important for the illusion of intelligence in AI-agents.

These three aspects are an important point of interest in developing a decision-making AI. With these aspects taken into consideration, it becomes easier to understand that it is not enough to create an AI that is able to pick and choose from a set of different actions that lead to a goal. Rather, it is also important that this AI in one way or the other expresses to the human player that it is in fact actively making decisions that are influenced by the state of the environment around them. If the human player does not notice the game agent's artificial ability to make decisions and does not participate in the illusion, then the AI-agent's decision-making abilities would be lacking in significance.

4.2 Creating the illusion

After explaining the importance of illusion in the creation of artificial intelligence for video games, Rabin (2017, Chapter 1) revealed a few ways to create the illusion. Two of those are discussed in the following sections.

Firstly, Rabin (2017) started by explaining that a subtle way to manage expectations is to tell the player about the AI's abilities. He advised to use the loading screen in the game for example to mention an AI character's ability to make different decisions based on a given situation. This can

be done in a subtle way where that information could be framed as part of a gameplay tip, he added. Rabin warned however of how this could backfire if the game builds high expectations, but the AI and the game's code do not actually meet those expectations.

Secondly, Rabin highlighted that the use of animations and dialog advances the illusion of intelligence in games. He pointed out that despite character animations not explicitly being part of the AI programming, they play an important role in highlighting when the AI is making different actions. Rabin gave an example of how using head movement animations helps show the player that an AI-agent is aware. He explained that an AI-agent's ability to look at an object or character that it is pursuing, shows the player that the AI-agent is intelligent. In reality the AI does not need to look at (or move its head towards) an object in order to interact with it but using animations this way highlights the game character's artificial intelligence. In addition to animations, Rabin said that intelligence can be illustrated by programming the ability for an AI-agent to adjust its speed. Running faster when the situation calls for it and walking slowly otherwise. Furthermore, dialog that is related to what is happening in the game between AI characters emphasizes their intelligence as Rabin explained.

4.3 Planner system, A* algorithm and goal-driven (regressive) search

The *Goal Oriented Action Planning* architecture (Orkin, 2003, 2004, 2006) uses a combination of smaller programming concepts to create a system that is able to create plans of actions in real time. This section explains those concepts in further detail and how they work with each other.

The *Goal Oriented Action Planner* is a planner that can be used by non-playable characters (NPCs) to satisfy a specific goal (Orkin, 2004). Orkin explained that the planner looks for a suitable arrangement of actions that can be executed to fulfill the goal. He added that each action can have some preconditions that also need to be met for the planner to work correctly. Additionally, Orkin said that in the case of there being multiple actions which have the same effect, the planner deals with those actions in a way that allows higher priority actions to override others when that is suitable. Furthermore, Orkin explained that actions and goals in this case have no explicit connection to one another, but rather, the planner creates that connection by

creating plans during the execution of the program. This, according to Orkin, allows engineers to define what a goal is and what an action is during development. Meanwhile, designers get to define the data files which describe each individual action and goal.

4.3.1 The planner

What made the FSM in GOAP different from other video game FSMs was that the logic to transition from one state to another was not embedded in the FSM itself but was rather embedded in a *planner* system (Orkin, 2006). Orkin explained that a planning system could tell the AI what the goal was and what actions were available and then let the AI create its plan in real time. This was contrary to regular FSMs which told the AI exactly what to do at any given situation, which meant that the FSM had to be designed to react to every possible situation and that made overly complicated designs.

A *planner system* according to Orkin (2003) is a system that searches the space of possible actions that could take a game character from a starting state to a goal state. If the planner succeeds in finding a plan, the character follows the plan until the goal is achieved or until a better plan becomes more relevant. If the plan becomes invalid during its execution for whatever reason, the planner tries to formulate a new plan for the character to follow. The planning system in F.E.A.R was based on the STRIPS planning system (Orkin, 2006).

STRIPS (STanford Research Institute Problem Solver) is a planning system that uses operators and goals (Fikes & Nilsson, 1971). Goals define a state of the world that the system wants to reach. Operators are the steps that are taken in order to reach that goal where each operator partially changes the state of the world (Fikes & Nilsson, 1971). For the STRIPS system to create a sequence of operators to reach its goal, it needs descriptions of the operators. These descriptions were separated in three main categories: 1. Name and parameters of the operator, 2. Preconditions, 3. Effects (Fikes & Nilsson, 1971). In GOAP, operators were referred to as *actions* (Orkin, 2006). Orkin revealed that in the game F.E.A.R (2005), different character types had different action sets. What this meant according to him was that in the game, when an assassin character and a rat character were given the same goal named *KillEnemy*, the assassin

would actually be able to form a plan consisting of the actions *patrol* and *attack*. Meanwhile, the rat did not have the *attack* action in its set of possible actions, therefore, if it tried to formulate a plan to kill the player it would only be able to go as far as to patrol the area (Orkin, 2006).

Another way that GOAP was different from STRIPS according to Orkin was that GOAP had the added functionality of *cost per action* and procedural *preconditions* and *effects*.

4.3.2 The A* algorithm and costs per action

Cost per action as explained by Orkin (2006) is where actions are assigned different values and the system gets to create a plan that is cost effective out of the set of available actions. That way, even if there are many different ways that a goal can be achieved, the AI system will only create a plan for the one way that is most cost effective (costs the least). He added that this functionality was implemented using the A* algorithm. The A* algorithm is a graph algorithm that uses an evaluation function which expands the node that has the least value (Hart et al., 1968). It repeats that function until it finds a full path of nodes with the least total cost to reach a given goal in the graph. For each node that the algorithm expands it marks that node as *open*. It searches for the next node with the smallest value (and at the lower level of the graph) and marks that as *open* and then marks the previous one as *closed* (Hart et al., 1968). Orkin (2003) suggested that the A* algorithm should be used for regressive search in video games. He explained that searching backwards meant that the search starts with the goal and finds the action that will satisfy that goal. Next, it will find the second action that satisfies the preconditions of the previous one. The algorithm will continue this way until it is able to formulate a full plan to satisfy the goal.

4.3.3 Heuristics and the set of relevant actions

In order for the GOAP planner to formulate a plan, it is not enough for it to simply pick whichever actions had the least costs. The chosen actions must also be heuristically and contextually relevant. Orkin (2003) explained that the planner needed to represent the state of the game world using a list of properties about the world. Each property in the list, as he said, would contain variables about that state of the world. An example he gave was how the *KillEnemy* goal could be represented in the game code. He showed that the goal could be represented with a world property structure that contained:

- **GAME_OBJECT-ID**: represents the object that this world property is concerned with. In the case of *KillEnemy* it's the human player (the AI's enemy)
- **WORLD_PROPERTY_KEY**: a key that represents the state of the world that this goal wants to achieve. *kTargetIsDead* for the goal *KillEnemy*.
- **Value**: some variable value (float, integer, boolean, etc.) that could describe the world property. *kTargetIsDead* needed a boolean variable with the value *true*.

Orkin added that the planner adds actions in the plan as each action adds a precondition which requires a different action. It does that until all preconditions for the goal state are met. The search for a viable plan successfully completes when the current state of the plan matches the goal state (Orkin, 2003).

4.4 Vision in video games

Video game vision is one part of game programming that really makes good use of the illusion of intelligence. In this section, game vision for AI-agents is discussed and explained in longer detail.

Every AI agent needs some way to perceive the environment around it. For AI agents in 3D video games, this is often done using what is called a vision cone (Rabin, 2017, Chapter 7.4.1). Rabin revealed that a vision cone determines the field of view (FOV) of a game character. As he put it, extending the length of a vision cone meant that the FOV also expanded in width. Furthermore, game vision is said to be composed of three vision checks, namely: distance, field of view and ray cast (Rabin, 2015, Chapter 4). Rabin explained that the distance check was used to restrict how far an AI agent could see. While the FOV check helped make sure that the agent only saw what was inside their view and not what was behind them for example. Lastly, the ray cast was said to be used to prevent an agent from seeing things that were behind walls or obstacles.

He further explained that ray casts are used to check if there is a clear line of vision between an AI-agent and their target (Rabin, 2014, Chapter 31). According to him, this is done by casting a ray from the AI-agent towards their target to check that the target is not obstructed by something

else. To limit the number of times an agent needs to do this check, Rabin elaborated that a ray cast and a vision cone can be used together. That helps an agent see their target using a vision cone first and then check if it is obstructed or not using a ray cast.

In some video games, an agent might be using several vision cones at once. In an article about the game *Alien: Isolation* (2014), the game's AI was explored, and they found out that all characters in the game had four different view cones (AIandGames, 2020). A normal vision cone for distant objects, another for objects that are straight ahead and in short distance, a third which was a peripheral vision cone and the last one which was for immediately close objects.

For a clearer explanation of these concepts see the figure below

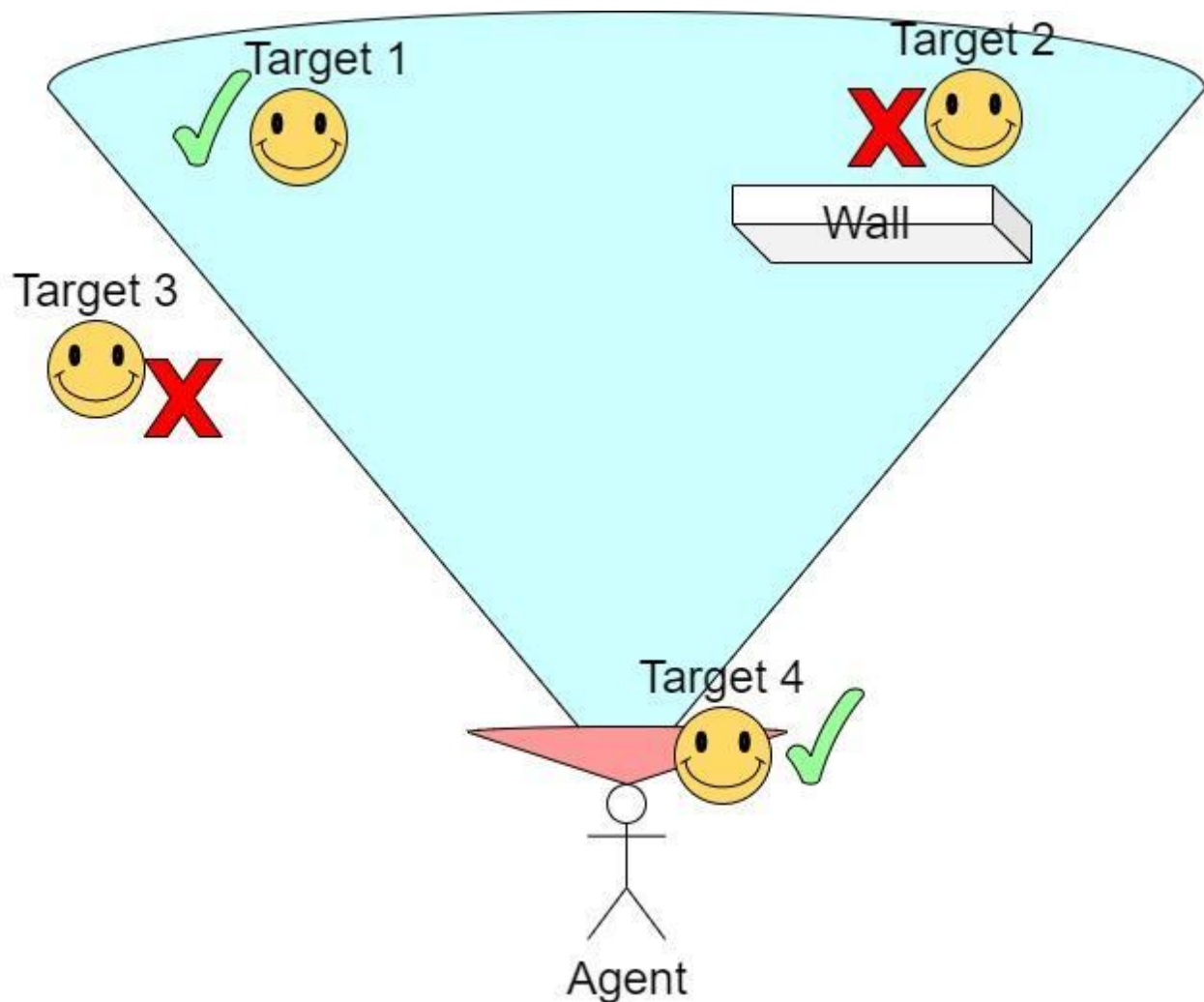


Figure 2: Vision cones. Green check marks show which targets can be seen and red crosses show which targets cannot be seen by the agent

Figure 2 is an example of an agent that uses two vision cones. It shows how target 1 can be seen by the agent because it is within the normal vision cone and there is a clear line of sight (a straight line/ray can reach from the agent to the target). Whereas target 2 cannot be seen because despite it being in the blue cone, it is obstructed by a wall. Target 3 cannot be seen, even though a clear line of sight can be established, the target is not inside either of the vision cones. And finally, target 4 can be seen by the agent because it is within the immediate vision cone (red cone).

To summarize, a programmer can equip an AI-agent with a combination of vision cones and ray casting to simulate vision. This further supports the idea that game intelligence is an illusion of intelligence, rather than a real human-like intelligence. The agent does not see in color nor in black and white. In fact, it does not really see anything at all, rather, it merely checks if a target is inside a vision cone object or not and if it is obstructed by a 3rd object. Simulating vision this way, does not require very complicated systems, yet it offers a crucial sensory ability for an agent's intelligence and aids in its decision-making abilities.

4.5 Working memory for AI-agents

In the field of artificial intelligence, there is a concept known as the “Working memory”. This section explains this concept and how it could be used in video game AI to further develop a decision-making system.

In their article about brain science and artificial intelligence, Fan et al. (2020) revealed that the working memory was discovered using functional magnetic resonance imaging (fMRI) from brain studies. This discovery aided in the creation of the artificial intelligence concept of long short-term memory (LSTM) which helped in many innovations within several fields such as, for example, natural language processing. Additionally, it was made clear that a working memory module can perform tasks that require complicated reasoning and inference.

Considering how useful the concept of a working memory is in computation and artificial intelligence, it was only a matter of time until the term started showing up in the field of game development. Orkin (2005) said that a working memory can be programmed for a game agent

such that the agent's sensory systems would be able to deposit acquired information in the working memory. Additionally, the agent's planner can be programmed to react to changes in the working memory, making it create a new plan when new information is acquired and added to the working memory. Furthermore, he added that sensors give perception facts to the working memory through a common format. In the paper, he described a project where he used a type which he programmed and called *WorkingMemoryFact*. This format was used as a record to save different attributes that represent things in the game world such as characters, objects, and their locations.

4.6 Different decision-making systems

The *Goal Oriented Action Planning* architecture (Orkin, 2003) is one of many decision-making systems that are used in game development. Therefore, looking into some other systems might create a better perspective into the prominence of decision-making AI-systems in video games. Rabin (2014) argued that behavior trees were amongst the most popular game AI architectures. He added that they were not as simple to develop as finite state machines, yet still somewhat easy to develop. It was also argued that behavior trees were helpful because they were modular and could be extended with new features and functionalities (Colledanchise & Ogren, 2017; Rabin, 2014). Sweetser and Wiles (2002) explained that finite state machines were used more often than any other AI systems in video games. They said that this was because of how easy they were to develop and how they give a clear structure of the different behaviors that a game character can exhibit by separating them into individual logical states.

The prevalence of these different decision-making systems in the field of game development warrants conducting a comparison between them. Because FSMs and behavior trees are the most popular system kinds used in game AI, they present a good base of comparison to see how GOAP compares against them. This will help understand how GOAP differs from the most popular solutions and in what ways they otherwise are similar to each other which would answer the paper's research question. Lastly, this will help in creating an understanding of what situation each of the three decision-making systems is best suited for.

4.7 The social impact of game AI research

When it comes to the value that game AI research can provide to society, Cook (2021) argued that game AI researchers had a responsibility towards society because of the impact that game AI research has. He explained that more increasingly, games are being used as a testbed for artificial intelligence and private labs like Google Deepmind and OpenAI have shown interest in video game AI research. Cook said that game AI researchers have a responsibility towards several groups of people. Firstly, he argued that they had a responsibility towards game developers because the researchers' work impacts developers' work opportunities and livelihoods. Meaning, that research in game AI impacts the scale of projects and work that is expected from game developers. He even warned that advancements in game AI research could lead to putting smaller developers out of work entirely.

Secondly, Cook (2021) added that researchers had a responsibility towards artists, hobbyists, and others. He explained that by focusing on game AI research as a tool for large commercial game companies, the research field would be excluding the kind of game developers that are making games as a hobby or ones who live in places that are internationally sanctioned and cannot sell their games on online global storefronts. Therefore, he said that research that results in software that is not open-source or that is incompatible with popular free tools makes it harder and costlier for many small developers to advance in the field of game development. As a result, game AI research becomes an exclusive tool only for larger wealthier companies which operate to benefit the global north (Cook, 2021).

Therefore, research on video game artificial intelligence, including decision-making systems for AI-agents, should be more easily accessible to better enable smaller game developers and hobbyists to participate in the creation of video games and to gain more control over their position as workers in the industry.

4.8 Summary of the theoretical framework

To summarize the theoretical framework of the topic, the flow of theories can be observed as such: First, video game AI does not need to be highly complicated to perfectly mimic human decision making. Instead, game intelligence can be thought of as an illusion of intelligence where smaller algorithms and systems work together to create an illusion of intelligent decision-making. Second, the *Goal Oriented Action Planner* is a decision-making planner system that makes use of the illusion of intelligence by creating an illusion that is the result of smaller components working together. Those components being an A* algorithm component that calculates the relevance of actions based on costs, and a world representation component that represents game world facts as a group of variables. This world representation component also informs the decision making because it gives heuristic meaning to the actions that an AI-agent can make. The GOAP planner interacts with and instantiates software objects called plans, actions, action sets and goals to create its effect. Third, a decision-making system is often supported by sensory systems such as vision cones and a working memory system where sensory systems can save game world facts as memories. Those memories/world facts are used by the decision-making system as previously mentioned. And finally, there are different types of decision-making AI systems in the field of game development and research on those topics has a social impact on people in and outside of commercial game development communities. Therefore, developing a decision-making system for games as part of this research paper's work, comparing it against other popular decision-making systems for games, and making it an accessible resource for future research is important. This would benefit the communities of game makers that do not have access to large budgets the same way that large commercial companies do.

Chapter 5 Methods and empirical data

This section discusses the methods that were used during the execution of this research project and the data and information that was acquired using those methods. These methods included design science research which motivated a comparison between the different decision-making systems, a literature review as part of the comparison and methods relating to the practical implementation of the project.

5.1 Design Science Research

Design science has been used in the field of engineering more than it has in the field of information systems, but Peffers et al. (2007) argued that more design science research could be done for the field of information systems. They argued for a design science research methodology where there were six main activities. First, the *problem identification and motivation*, which aids in the development of the research artifact (a GOAP system in this case). Second, *defining the objectives for a solution*. This activity is used to infer the objectives that the research is trying to achieve. Third, *design and development*. Peffers et al. (2007) said that this includes designing an artifact that has an inherent contribution to the research topic. Fourth, *Demonstration*, which entails putting the designed artifact to the test in a relevant environment where its ability to solve a problem can be observed. Fifth, *evaluation*, which means inspecting how well the artifact solves the problem at hand. This inspection can be done by comparing the artifact to the original objectives that were outlined or using different ways of inspection. Sixth, *communication*, which is important for delivering information about the usefulness of the artifact to researchers and other audiences that can learn from it.

Peffers et al, (2007) said that these activities do not need to be done in a particular order. Different research projects might need to begin with different activities. Having already defined the research problem in section “3.4 The research gap”, the first activity of the “problem identification and motivation” activity can be considered finished. Additionally, the

“communication” activity is concerned with the publication of the research paper, which means this activity will be finished after this paper is published. The creation of this research paper and publicly publishing it will be helpful for many game AI developers and hobbyists (Cook, 2021). These groups are the relevant audience for this research paper and the paper will serve as learning material for them. The following sections of the paper will explain how each of the other design science research activities were conducted.

5.2 Analysis and Comparison

Comparing digital systems can be crucial in determining the choices that should be made during a development process, but they can also be useful in determining how an already developed system compares against other systems which fulfill the same goal. Lazarevich (2018), explained that software solutions and frameworks can be analyzed and compared by using a 5-step process. The purpose of Lazarevich’s analysis and comparison method was to decide which system out of the ones under question was the best to develop. In this paper however, the analysis and comparison were used to compare the GOAP (Orkin, 2003) system to other game decision-making AI systems. Which means that the chosen system that was developed was the GOAP system from the start, and then came the comparison step which is meant to contextualize GOAP as a decision-making system in comparison with other more popular systems (FSMs and Behavior Trees).

Before comparing the different AI-systems in question, a general understanding of the project’s requirements was needed. In design science research this was outlined as the “defining the objectives for a solution” activity (Peffer et al., 2007). In this case, the project was a game prototype where two types of AI-controlled characters (hunter and hunted), played against each other. The hunter character would have the ability to catch other characters and remove them from the play area, while the hunted character type would have the ability to hide and find and obtain keys to unlock a door through which they could escape the hunter. Therefore, the decision-making AI system that is developed for this project, must be able to manage many actions and allow a game designer to create new actions. It also needs to be able to automatically manage the actions’ priority in the decision-making process among other features. Therefore,

Table 1 below was created to explain what the project's requirements were and why they were needed:

| Decision-making system requirements/objectives | Motivation |
|--|--|
| manages many decisions | Each character type in-game has a set of several different decisions. Therefore, the system must be able to represent the variety of decision sets. |
| decisions are modular | To be able to create different decision sets, the system must allow for representing each decision as a separate object that can be added or removed from a decision set |
| automatic coordination of decisions | The system must have an ability to create plans that consist of multiple decisions that are played out in a correct order |
| automatic replanning | The system must be able to replan when a previous plan becomes invalid because of some change in the Gameworld. |
| reusable with different game character types | The system must be reusable for most AI-controlled character types |
| easy to set up | The system should not be very complicated to develop |
| Adopts a way for world representation | The game world must be represented somehow in the system in order to allow the AI-agents to react to different Gameworld changes |
| Ease of customization | The system should allow for easy customization for game designers, and give them freedom to create new decisions that a character is able to make |

Table 1: a table showing the requirements for the decision-making system that was developed

The comparison process (Lazarevich, 2018) was comprised of 5 steps:

- Define the alternatives
- Analyze the alternatives
- Ask questions to create a complete image of each alternative
- Make a result sheet
- Define pros and cons and make your choice

5.3 Literature review

To develop a project that uses the GOAP architecture, the literature on the topic was reviewed to base the project's code from the information that was described in the literature. Additionally, the literature review informed the analysis and comparison steps when comparing the GOAP architecture to other solutions. The literature review included the following sources for each of the different AI decision-making solutions:

- FSM: Cossu (2020a) and Sweetser and Wiles (2002).
- Behavior trees: Rabin (2014, 2017).
- GOAP: Orkin (2003, 2005, 2006).

5.3.1 Defining the alternatives

Going back to Lazarevich's 5-step comparison (Lazarevich, 2018), the alternatives to creating a decision-making AI for game characters (Step 1 in the comparison) could be described as such:

Goal Oriented Action Planning (Orkin, 2003): An AI system that uses multiple components to enable a game character to dynamically create plans while the game is running.

Finite State Machines (Cossu, 2020a): A system that has definitions for the different states that a game character could have. The transitions between the states are embedded in the game's code. Therefore, any plan that a character could make, must be pre-defined by the developers.

Behavior trees (Rabin, 2017): a common technique of developing game-AI. It is used when there are many states and transitions between them which could justify creating a tree-like hierarchy of states to enable a more complex decision-making logic.

5.3.2 Analyzing the alternatives

This step requires a comparison between the different alternatives by finding information about each alternative through different sources and learning how they work (Lazarevich, 2018). Additionally, this step includes creating a test project. For this purpose, the game prototype that was developed and used one of the alternatives, namely the GOAP architecture, was considered part of this comparison. Finally, the game prototype's GOAP implementation was analyzed in comparison with the other alternatives based on the literature available on them. As mentioned earlier in this paper, the comparison was to aid in understanding how GOAP (with its different components) compares to other popular systems.

| GOAP Components | GOAP | FSM | Behavior Tree |
|-----------------|--|---|--|
| Planner | Has a dedicated planner that uses an AI search algorithm to create plans | Does not have a planner. Plans are pre-designed by game designers and embedded in the state transitions | Does not have a planner. Has a "Selector" class, which tests different predefined "Sequences" and checks which one works |
| Goal | Is an object class. Instances can be created from it | Does not use a "goal" logic | Does not use a "goal" logic |
| Action | Is saved as a data file in ex. Json, xml or scriptableobject (for | Each state represents an action | Actions are implemented using a "Behavior" interface |

| | | | |
|----------------------|--|---|---|
| | unity3D) formats | | class. Actions are leaf nodes in the tree |
| Plan | Is an object class. Instances can be created from it. Is created dynamically by the AI (Planner) | Embedded in the FSM's state transition logic. Predetermined by game designers | "Sequences" allow an agent to follow a plan that is predetermined by game designers |
| Action set | Is saved as a data file in ex. Json, xml or scriptableObject (for unity3D) formats | Each agent type requires a different FSM due to a lack of "Action Set" logic | Each agent type requires a different behavior tree. Behaviors can be reused and recombined to create different behavior trees |
| World representation | World states can be represented as "struct" or "enum" type classes | Not represented in any particular way for regular FSMs | Using the "Condition" leaf node. Returns statuses that represent the success or failure of a behavior |

Table 2: table of analysis of alternatives

Table 2 lists the three different decision-making systems that the comparison was concerned with. It also lists different components that the GOAP system uses and describes what each of these components matches in Behaviour trees and FSMs.

5.3.3 Asking questions about the alternatives and making a result sheet

After analyzing the alternatives, Lazarevich (2018) suggested that questions should be asked about each of the alternatives to get a clearer image about each of them. In addition to the questions, a result sheet should be created in order to aid in the final “pros and cons” step of the comparison. The following is the result sheet:

| | GOAP | FSM | Behavior tree |
|------------------------------|---|---|--|
| How easy is it to customize? | Easy to customize because of its modular components (actions and action sets) | Not very customizable. Each type of AI-agent requires a different FSM | Easy to customize because of modular components such as “Behaviors”. New behavior trees can be created easily |
| What are the capabilities? | Dynamically creates decision-making plans during a game’s execution. Leads the decision-making of an AI-agent | Makes decisions based on hardcoded game logic. Best used with AI-agents that exhibit few states/actions | Leads the decision-making logic. Allows for creating AI-agents that exhibit many different behaviors |
| What are the limitations? | Each action requires preconditions and effects. If not carefully designed, these actions can become useless if their effects cannot chain with other actions’ preconditions | Not suitable for AI-agents that perform many actions. Creating FSMs that contain many states can easily become hard to maintain. Each different type of AI-agent requires a different FSM | Each different type of AI-agent requires a different behavior tree. This means a lot of work has to be done if there are many different types of game characters in the game |
| Is it easy to develop? | Requires good knowledge in game | Very easy to develop and does not require a very deep knowledge of game | Requires good knowledge in games and systems |

| | | | |
|--|--|---|--|
| | programming and systems programming | or systems programming. Is beginner friendly | programming. Not beginner friendly. |
|--|--|---|--|

Table 3: Questions and results about the alternatives

Table 3 is a result sheet for several questions that were produced to be asked about each of the decision-making system alternatives. The answers for each of the questions were inferred from the literature while the answers about GOAP specifically were also aided by the developed game prototype which used GOAP AI.

5.3.4 Defining pros and cons

The final step of the comparison is defining the pros and cons of each of the alternatives according to Lazarevich (2018). He added that these pros and cons can be checked against the scope and requirements of the project to put them into context. The Design Science methodology had the “evaluation” activity (Peffer et al., 2007) which meant inspecting how well the solution solves the given problem. This “evaluation” step is covered by the final step of the comparison. This will be presented in Chapter 7. Lazarevich (2018) also suggested that this step might lead to several viable options to choose from.

5.4 Concept maps

In their article about literature reviews, Rowley and Slack (2004) revealed that some researchers used concept mapping to map out key concepts within a research area and the relations between them. They also pointed out that there was no correct way to draw a concept map. What really mattered about a concept map according to Rowley and Slack was that it assisted a researcher in understanding their topic. Using this method, a concept map was developed to organize key concepts from the relevant literature. See Figure 3 below

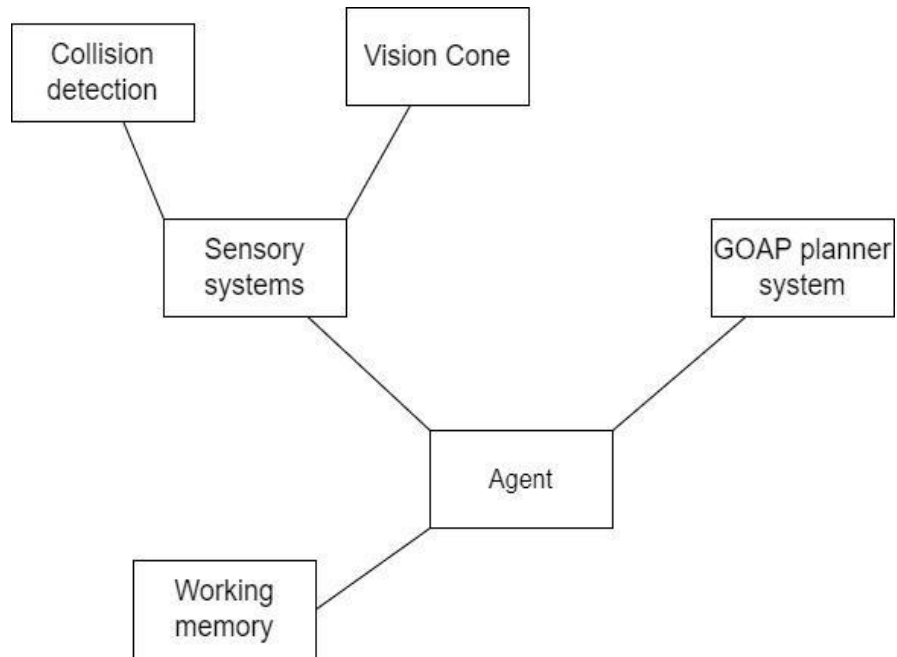


Figure 3: Concept map of relations between the agent's different components

The developed concept map (Figure 3) helped set into perspective the relation of the GOAP system to other components of a game's AI agent. This was helpful in coherently separating the different parts of the project's components from the GOAP system during development.

Chapter 6 Ethics

This paper compared different decision-making systems for video games. The comparison was based on information acquired through a literature review of the three types of systems (FSMs, behavior trees and GOAP). However, when it comes to GOAP, the acquired information was also aided by the prototype that was developed for this thesis. This might have caused a bias for GOAP compared to the other two systems because of the time spent on and the experience of developing a GOAP AI-agent. Therefore, the comparison could have possibly been more informed and fairer had a larger scale project been conducted which would have included developing prototypes of all three types of systems.

Chapter 7 Presentation of results

The following section presents pseudocode that shows how the GOAP prototype was programmed. The pseudocode shown in this section was for the classes of the planner, goal, action, action set, plan, goal, GOAP-agent, world representation, vision sensor and the blackboard. Finally, the results of the comparison between the GOAP, FSM, and Behavior Tree system types are presented followed by an analysis.

As previously mentioned, the problem with having simpler system such as FSMs or possibly not having any coherent system whatsoever is that as soon as the AI starts becoming more complex and is required to do many things and alternate between many decisions, the project's code becomes very long and unmaintainable. Therefore, it is important to have systems that can manage that complexity and accommodate a variety of decisions. This is where the GOAP system comes in.

Developing an artifact as part of the research project was important to understand GOAP through more than just the literature. This development process aided in the comparison step and was documented using pseudocode. This makes the paper an accessible resource for others who are interested in understanding or developing a GOAP system. The following sections describe how this research paper's artifact, (the GOAP system) was designed and implemented.

7.1 Project creation in the Unity engine

In order to program a functioning GOAP AI, a game engine was required to remove the bulk of unrelated work that did not directly have to do with the decision-making AI. A game engine is a collection of software modules which are responsible for simulation but do not directly specify the game's behavior, logic nor environment (Lewis & Jacobson, 2002). The Unity game engine (2021) was chosen for this project. Unity has built-in features that simulate important things that were needed in this project but were not the main focus of it. Such features were for example,

simulating the passing of time, physics simulation (gravity, collision, etc.), rendering 3D graphics, creating navigation meshes (places where a 3D object is allowed to move), lighting and shadows and other features.

Additionally, the Unity engine allows the use of C# code which was used to implement the GOAP decision-making system. Implementing the GOAP system in a relevant game engine was important based on the design science methodology's activity "demonstration" (Peffer et al., 2007) This activity required that the developed artifact should be demonstrated in a relevant context. The Unity engine fulfilled that purpose. The Unity engine is a free of charge game engine that is also very popular among hobbyist game creators. Cook (2021) emphasized the role of popular and free of charge software in advancing game AI research. He believes that it is necessary that research be accessible for a larger group of people and that this could be done by implementing the research in software that is free to use and known by many.

7.2 Unified Modeling Language (UML)

After identifying the important concepts of a GOAP system through the literature and separating the non-GOAP components using a concept map (see Figure 3), the next step was to create a UML diagram to represent the GOAP components as UML classes (see Figure 4). It was important to keep the diagram simple in order to not restrict the actual code writing process later in the project.

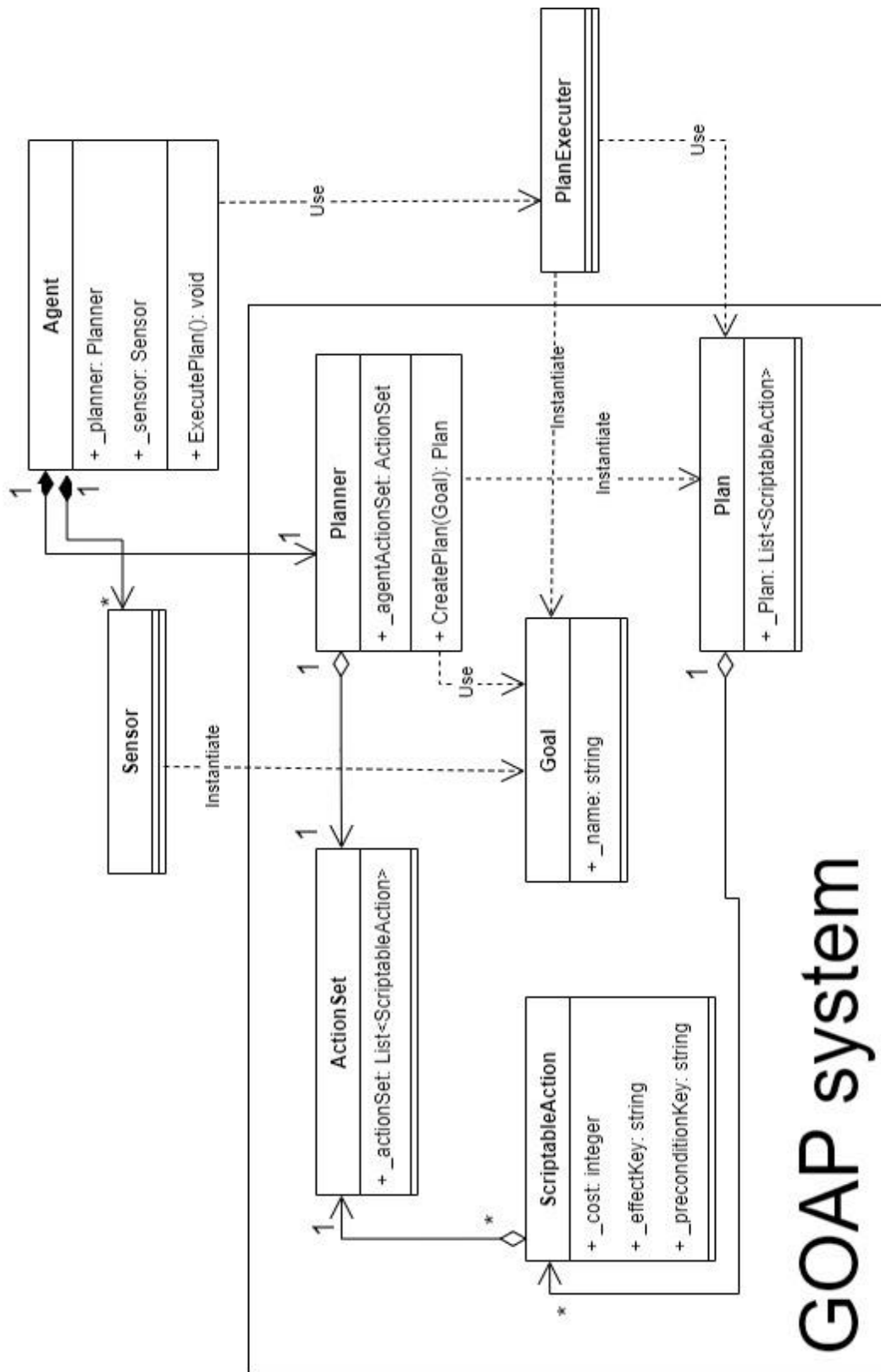


Figure 4: UML diagram of the developed GOAP system

Figure 4 shows that an AI-agent can have one or more sensors (a vision cone is an example of a sensor), a GOAP-planner system and a class that runs the planner's plans (named PlanExecuter in the diagram). Additionally, the GOAP system as seen in the diagram, shows the relationships between the previously mentioned parts of a GOAP system. Namely, a planner, goals, plans, action sets and actions (named ScriptableAction in the diagram). A planner uses a goal to create a plan, which is used by the plan executor. Both the plan executor and different sensors can create new goals. A plan consists of one or more actions. And finally, a planner contains an action set which in its turn contains a predefined list of actions.

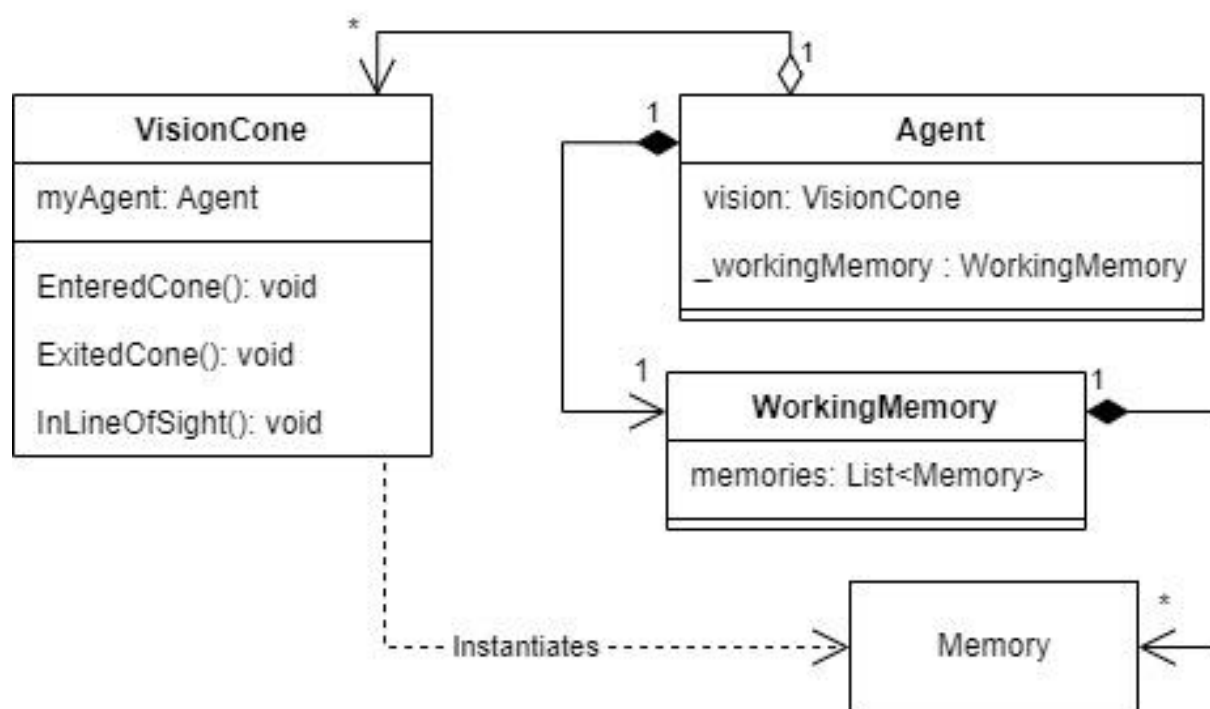


Figure 5: a UML diagram of the relations between the VisionCone and the Agent's WorkingMemory

Figure 5 shows that an agent can have one or more vision cones and also has a working memory. A vision cone can create memories which are then saved in the working memory.

Both concept mapping and using UML diagrams were part of the design process of this research paper's artifact. The design science activity "design and development" (Peffer et al., 2007) requires that an artifact which adds value to the research topic, should be developed for this kind

of research. Additionally, more documentation about the development of the GOAP system (the artifact) is presented in the following section (7.3 Implementation).

7.3 Implementation

This project was done in the UnityEngine (2021) and therefore, all the code for the project was written in C#, which is the main programming language used in the engine. The following section describes some of that code and how it works. The code in this paper is partially C# but is otherwise pseudocode. `MonoBehaviour` is a base class for much of the game code that is written in Unity. Therefore, when writing code for games in Unity, sometimes, creating a new class means it should be made to inherit from the `MonoBehaviour` class in order to use some of the engine's most basic functionalities.

Documenting the development process was important to communicate the results of what was previously designed and programmed in the project (Peffer et al., 2007). The project's code can otherwise be found on an online repository as it was uploaded incrementally during the duration of this research project (Al Shehabi, 2022). Keeping both an easily readable pseudocode version and the project's real code publicly accessible helps game developers, hobbyists and others who might be interested in experimenting or learning from this research paper. Doing that was important, because as Cook (2021) explained, researchers had a responsibility to make game AI research accessible to those communities who are involved in game development regardless of if it was related to their work or purely for fun or self-expression purposes. This was important for this research paper's aim in advancing research in the field.

Furthermore, the development of a GOAP system was important to further understand this system through more than just reading the literature. The developed artifact was also important for the comparison step which answers this paper's research question "How does the *Goal Oriented Action Planning* architecture compare to other artificially intelligent decision-making systems in video games?".

7.3.1 The Planner

The planner was programmed to create plans based on a provided set of actions that an AI character could make. Additionally, the planner's plan creation method required a goal. This way, anytime the AI character gained a new goal, it would give it to its planner and the planner creates a plan based on that goal. The AI character then follows that organized plan of actions. The code for this project's planner class looked approximately like this:

```
public class Planner
{
    private List<ScriptableAction> closed;

    public Plan CreatePlan(//Requires: goal)
    {
        //Create instance of 'Plan'
        //Execute 'SearchActions' using 'goal'
        //Save the output of 'SearchActions' in the instance 'Plan'
        //Output the 'Plan'
    }

    List<ScriptableAction> SearchActions(//Requires: goal, openList,
    actionSet)
    {
        private List<ScriptableAction> graphRow;

        //Create a row of actions that fulfill the provided goal
        //Add those actions to the 'graphRow' list
        //Add contents of 'graphRow' to the 'closed' list
        //Remove actions from the 'actionSet' list if they also exist in
        'closed'
        //Find the action with the smallest cost in the 'graphRow' list
        //Add cheapest action to 'openList'
        //Make the cheapest action's precondition into a 'goal'
        //Run 'SearchActions' using the new 'goal', 'openList' and
        'Actionset' (this implies recursion of this method inside
```

```

        itself)
    //Output the 'openList' after all recursions have executed
}
}

```

Notice that using a “graphRow” list in this manner means that a new list is created in each recursion of the *SearchActions* method, and it holds only the row of actions from that recursion. Meanwhile, the *openList* keeps expanding with actions added to it from each recursion. The *SearchActions* method uses an A* algorithm to find the solution. An A* search algorithm requires a graph of some sort to find the cheapest path in the graph from point A to point B. It is important to point out that *SearchActions* search algorithm runs a regressive search. Which means it runs backwards from the goal towards the first action that should be taken. In other words, it starts from point B and finds its way backwards to point A.

The *SearchActions* method starts by creating a row in a graph by examining the available set of actions and finding the actions which fulfill the end goal (see Figure 6). A recursion is a recurrence of a method that calls itself until it finds its desired solution. During each recursion of the *SearchActions* method, it first finds a suitable row of actions as mentioned, then it finds the action that has the smallest cost (this is where the A* algorithm is in effect) and adds it to the opened list. After that, the method runs itself again (recursion) to create the next row and find the cheapest action in it. This process continues to happen until a plan which fulfils the given goal has been found.

To explain this process with an example, imagine a scenario where a virtual character wants to buy and set up a new chandelier in their house and do it in the most effective way. The search algorithm for a situation like this could go on as such: First, it creates a row of suitable stores to buy chandeliers from. Meaning that it searches through a set of many stores and only adds the ones that sell chandeliers to the row. This is important, because when the algorithm starts looking for the cheapest chandelier, it should not be looking for it at stores that do not sell chandeliers. Second, it selects the one store from the row that is closest to where the character lives. Third, it searches through all the products at that store and creates a row of the products that are chandeliers. Fourth, it selects the cheapest chandelier available in the row.

```

//uses recursion to create a list of actions that can reach a goal and costs the least amount
List<ScriptableAction> SearchActions(Goal goal, List<ScriptableAction> openList, List<ScriptableAction> actionSet)
{
    List<ScriptableAction> graphRow = new List<ScriptableAction>();
    closedList = new List<ScriptableAction>();

    //Create a row in a graph
    foreach (ScriptableAction action in actionSet)
    {
        Memory fact = new Memory() { state = action.preconditionKey };

        if (action.effectKey == goal.goalState && myAgent.memory.ContainsMatchingMemory(fact))
        {
            graphRow.Add(action);
        }
    }
    closedList.AddRange(graphRow);

    //Removes closedList items from actionSet
    actionSet = RemoveActions(actionSet, closedList);

    int maxCost = 10;
    ScriptableAction cheapestAction = graphRow[0];

    //Find cheapest action in row
    foreach(ScriptableAction action in graphRow)
    {
        if(action.cost < maxCost)
        {
            maxCost = action.cost;
            cheapestAction = action;
        }
    }

    openList.Add(cheapestAction);

    Goal newGoal = new Goal(cheapestAction.preconditionKey);

    List<ScriptableAction> finalGraph = new List<ScriptableAction>();

    //Recursively finds the next suitable action
    //this check stops the recursion if the new cheapest action's precondition matches a memory in the working memory
    //and that represents the end node of a A* action graph
    if (!myAgent.memory.ContainsMatchingMemory(new Memory() { state = newGoal.goalState}))
        finalGraph.AddRange(SearchActions(newGoal, openList, actionSet));

    return openList;
}

```

Figure 6: the project's SearchActions method from Planner class

In essence, the previous example explains that this kind of search algorithm recursively takes two steps at a time and repeats. One step to create a row and one step to make the cheapest decision. The example could continue like this until the chandelier is up on the ceiling (take the shortest path home, find the most suitable tool to hang the chandelier, etc.).

7.3.2 Goal

The Goal class was programmed to contain a variable which represented what the state of the

world should be when the Goal was fulfilled by the AI agent. The planner needs a goal each time it creates a new plan. This goal acts as a starting point for the search algorithm that finds a suitable plan of decisions. As previously mentioned, the search algorithm runs regressively, so it does not try each action to see if it leads towards a goal. Instead, it starts with the goal and runs backwards to create a logical plan.

```
public class Goal
{
    public WorldState goalState;

    public Goal(//Requires: a new goalState)
    {
        //Set the value of the local goalState to equal the new
        goalState
    }
}
```

When an instance of this class is created (a new goal), it requires a new state of the world through its constructor and saves it in its local goalState variable. An instance of a Goal class is used by the planner to create a plan that fulfills that goal.

7.3.3 Action

Actions in this project were represented as a class which was named ScriptableAction. The ScriptableAction class was programmed to inherit from a Unity base class called ScriptableObject (Unity, 2018). According to Unity, scriptable objects are data containers that are intended to be used to save data that is unchanging. Therefore, this base class was useful because GOAP actions are meant to be defined during the design process and then remain unchanged during execution of the program.

```
public class ScriptableAction : ScriptableObject
{
    public int cost;
    public WorldState effect;
    public WorldState precondition;
```


}

The ScriptableAction class represents GOAP actions as previously mentioned. The information that was saved in each of these actions was cost, effect, and precondition. Each action has a precondition that needs to exist in the current state of the game world and an effect that it applies to the state of the game world. Preconditions and effects were important because the Planner class requires them in order to be able to tell if one action can be chained with another action. In other words, an action's effect needs to match the next action's precondition. This way, actions could be chained together to create a full plan of actions.

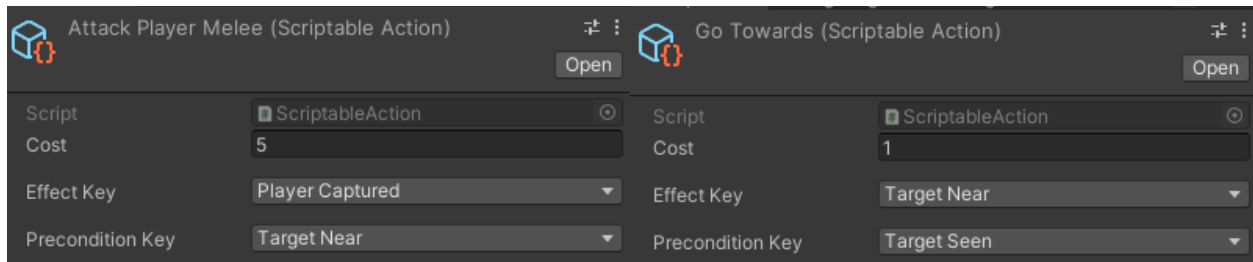


Figure 7: Two scriptable action instances in the Unity engine's interface

Figure 7 above shows two different actions in the unity engine's interface. The action on the right can heuristically chain to the action on the left. The Planner class does this when it sees that the action on the right has the effect "target near" which is a precondition in the action on the left. Assuming that the planner had created a plan which contained these two actions, if the final action in the plan was "Attack Player Melee", that would mean that the goal that the plan fulfilled was to have "Player Captured".

7.3.4 Plan

A plan according to Orkin's GOAP system (2003) is an array of actions that leads to satisfying a given goal. In this project, a plan was represented as a class that contained a variable list which contained a sequence of instances of the class ScriptableAction. This list of ScriptableAction instances comes from the planner when a plan is created. The plan receives a list of ScriptableActions through its constructor and saves it to its local list of ScriptableActions. After that, the plan would be ready to be received by a GOAP game agent and executed.

```

public class Plan
{
    private List<ScriptableAction> actions;

    public Plan(//Requires: list of ScriptableActions)
    {
        //Save the received list of actions as the local list of
        actions
    }
}

```

7.3.5 ActionSet

The concept of the ActionSet was programmed in this project as a subclass of Unity's built-in ScriptableObject (Unity, 2018). Similarly to a ScriptableAction, the contents of the ActionSet are defined during the design of the project only. They remain unchanged during the execution of the game. In this project, an ActionSet was programmed to contain a list of ScriptableActions which tells an AI-agent which actions it is allowed to choose from when making decisions (aka. when creating plans).

```

public class ActionSet : ScriptableObject
{
    public List<ScriptableAction> actions;
}

```

Orkin (2006) explained that different sets of actions should be used for different kinds of game characters. While some actions could be the same for all characters (such as a walking action), not all actions should be shared between all characters. A cat character for example, should not be able to pick up a firearm and use it. Therefore, a cat's ActionSet should not contain the exemplified action "Pick up firearm".

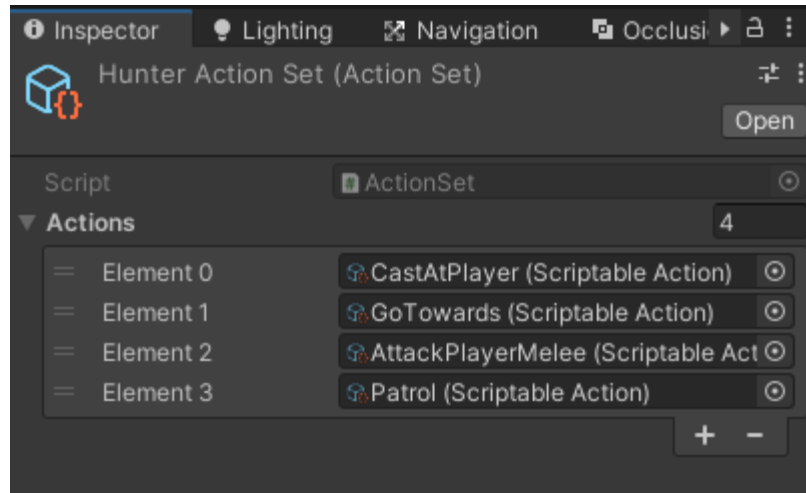


Figure 8: The hunter's action set in the Unity engine's interface

The above figure (Figure 8) shows an action set from the Unity project that was done for this paper. The action set in the image contains four different actions which were each defined using the ScriptableAction class.

7.3.6 World Representation

The WorkingMemory class was implemented to contain an agent's memories and run some operations on them. Several classes in the project use the WorkingMemory's methods. The planner class, for example, checks the WorkingMemory to see if there are world facts that could help it create new plans.

```
public class WorkingMemory
{
    private List<Memory> memories;
    private GoapAgent myAgent;

    public void AddMemory(Memory memory, Goal goal)
    {
        // Add a new memory to the working memory
        //Tell myAgent's planner to create a new plan using the given
        goal and execute it
    }
}
```

```

public List<Memory> GetMemories()
{
    return memories;
}

public void RemoveMemory(Memory memory, Goal goal)
{
    //Remove a given memory from the working memory
    //Tell myAgent's planner to create a new plan using the given
    goal and execute it
}

//Checks if there is a given worldfact that matches a worldfact in
memory
public bool ContainsMatchingMemory(Memory worldFact)
{
    //Check if the given worldFact already exists in the working
    memory
}

```

The WorkingMemory has four methods. AddMemory, which adds a new memory to the memories list and tells the agent's planner to create a new plan. The GetMemories method returns the list of memories from an agent's working memory. The method RemoveMemory removes a memory and tells the agent's planner to create a new plan. And finally, the ContainsMatchingMemory returns a true/false value about whether a given memory already exists in the agent's working memory.

Additionally, the WorldState enum class was programmed in order to represent the state of the game world at a given moment. This class contained a collection of values that were expanded on during the development of the project. Each value represented a state of the world that an AI-agent could know about. When a world state is fulfilled in the game, it gets added to the relevant AI-agent's WorkingMemory.

```

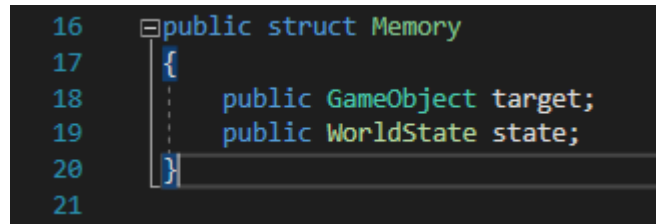
public enum WorldState
{

```

```

    playerCaptured,
    projectileAvailable,
    playerSeen,
    meleeAvailable
}

```



```

16 public struct Memory
17 {
18     public GameObject target;
19     public WorldState state;
20 }
21

```

Figure 9: The Memory struct from the project

Adding these WorldState values to an agent’s memory means that the planner gets new information to use when creating a plan. A WorldState is added to the WorkingMemory as part of a Memory instance (see Figure 9). The Memory struct was implemented to contain a WorldState value and a GameObject value which represented the target of that memory.

7.3.7 Agent

An agent as described by Orkin (2005) was composed of multiple parts and those were a blackboard, a working memory, some subsystems, and sensors. The role of the sensors was to detect game-world changes and save that information in the working memory. The blackboard acts as a bridge that communicates information between the agent and its subsystems. In this project, the GoapAgent class was created to communicate information between the planner, working memory, sensors, and other classes. The GoapAgent class looked like this:

```

public class GoapAgent : MonoBehaviour
{
    public WorkingMemory memory;
    public ActionSet agentActionSet;
    public Planner planner;

    public Plan ObtainNewPlan(Goal goal)
    {

```

```

        //Recieve a new goal
        //Ask the planner to create a new plan
    }

    public void ExecutePlan(Plan plan)
    {
        //Send the plan to the blackboard to be run
    }
}

```

The blackboard (Orkin, 2005) was implemented as a separate class which all game agents had access to. This class was named PlanExecuter

```

public class PlanExecuter : MonoBehaviour
{
    public void Execute(Plan plan)
    {
        //Take each action in the plan
        //Find a suitable method for that action from the implemented
        methods in this class
        //Run that method
    }

    void ActionA()
    {
        //Run an action A with custom code
    }

    void ActionB()
    {
        //Run an action B with custom code
    }

    void ActionEtc()
    {


```

```

        //Run custom code
    }
}

```

The PlanExecutor class was created to receive plans from the GoapAgent and run each individual action within that plan. The methods inside the class were not actually named ActionA, B, etc. Rather, they were given action specific names with each method running different lines of code. For example, see Figure 10 below which shows an action's custom code using a C# method named GrabKey.



```

105 void GrabKey()
106 {
107     currentAgent.GetComponent<Animator>().SetTrigger("key");
108
109     Memory newFact = new Memory() { state = WorldState.GrabbedKey, target = currentTarget };
110
111     currentAgent.GetComponent<GoapAgent>().memory.AddMemory(newFact, new Goal(WorldState.targetSeen));
112
113     currentTarget.GetComponent<Key>().GetTaken();
114 }
115

```

Figure 10: A code snippet that shows the method GrabKey from the PlanExecutor class

Other actions were for example things like: go to location, find other characters, cast projectile, hide, etc. Because the actions were many and their code changed during the creation of the project, they were not something that could be represented in detail in the pseudocode parts of this paper.

7.3.8 VisionSensor

The vision sensor class was implemented to represent an agent's vision cone. In this project, there were two different agent types, Hunter and Hunted. Therefore, the VisionCone class was an abstract class with two different implementations for each agent type.

```

public abstract class VisionSensor : MonoBehaviour
{
    protected GoapAgent agent;
    protected List<GameObject> FovTarget;

    private void OnTriggerEnter(Collider other)

```

```

{
    if (//If an enemy enters the borders of the vision cone)
    {
        //Add enemy to the FovTarget list
        //Run method StareAtTarget
    }
}

private void OnTriggerExit(Collider other)
{
    if (//If an enemy leaves the scope of the vision cone)
    {
        //Remove the enemy from the FovTarget list
        //Remove from the agent's working memory, the memory that
        contained the WorldState "targetSeen" for the current
        target
    }
}

public abstract IEnumerator StareAtTarget(GameObject target);
//Continuously cast a line of sight towards a target to make sure it is
not behind an obstacle
//Add a memory to the agent's working memory about which target was seen
in the vision cone and was not behind an obstacle
}

```

The `OnTriggerEnter` and `OnTriggerExit` methods are two methods that come from the `MonoBehavior` class (a base class in the unity engine). These methods check if something entered or exited a 3D trigger. A 3D trigger can be any 3D shape and, in this case, it is the vision cone (see Figure 11). In the vision sensor class, these two methods were used to check if an enemy character entered or exited the borders of a vision cone.

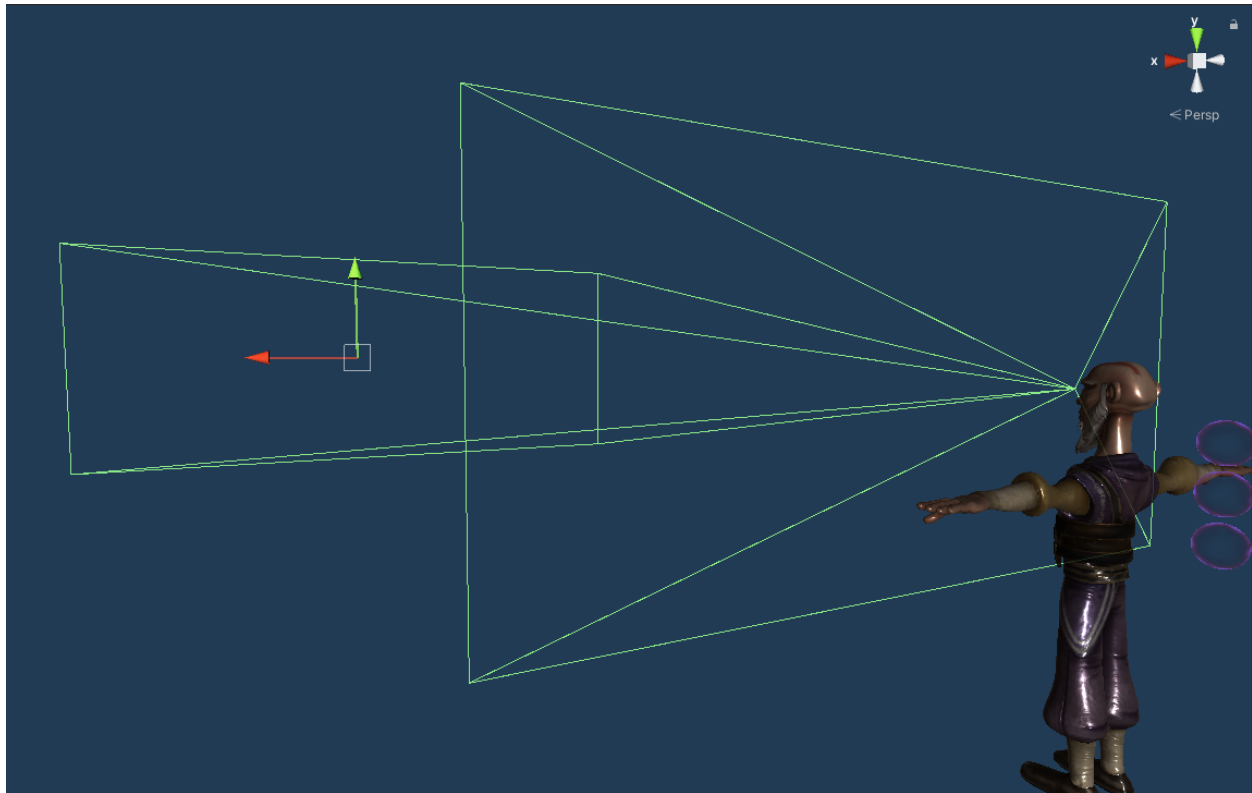


Figure 11: A “Hunter” character with two vision cones

The figure above (Figure 11) shows the hunter character in the project. It has two vision cones, one for near sight and one for far sight. If another character walks into this agent’s vision cones, a ray will be cast to check that they are not behind an obstacle (wall). This is done by the `StareAtTarget` method in the `VisionSensor` class. That method also adds the target of the vision cone to its agent’s working memory.

And finally, Figure 12 below is a concept map that shows the relations between the vision sensor, working memory, planner, and the blackboard

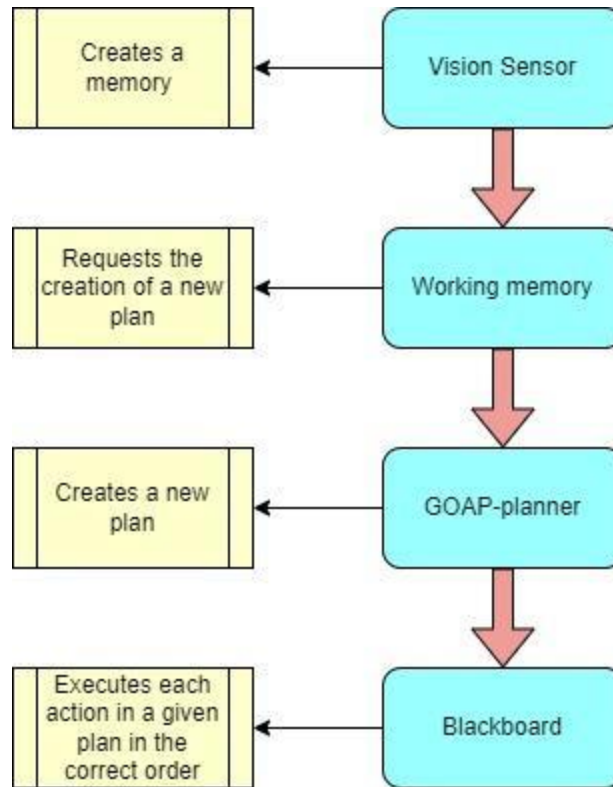


Figure 12: a concept map of the relation between the vision sensor, working memory, planner, and the blackboard

As seen in the concept map above, a game agent’s vision sensor (or any other sensor for that matter) can create a memory about certain game events that the sensor witnesses. This memory is then added to the working memory which prompts the working memory to tell the GOAP-planner to create a new plan. The new plan is then given to the *PlanExecuter* (the blackboard) which makes the AI-agent do the actions that were in the plan.

7.4 Comparison results

As stated earlier in this paper, a comparison was made between the developed GOAP system and two other decision-making types of AI. The two other types of AI in question were the “behavior tree” and the “finite state machine”. The final step of Lazarevich’s comparison (2018) was to define the pros and cons of each system to find which one is the most viable solution for the intended purpose. In the following table however, a “bad, good, best” categorization was used. This allows representing the comparison with more nuance. Only one system can be considered

best at a certain task, but multiple systems can be good or bad at certain tasks. What the intended purpose is and what counts as a good, bad or best was based on the previously stated requirements of the system. The results of the comparison were the following (see Table 4):

| | FSM | Behavior tree | GOAP |
|-------------------------------------|--|---|---|
| manages many decisions | Bad Having many decisions in the FSM makes it unmanageable | Good Manages many decisions using different tree nodes | Good Manages many decisions using different actions |
| decisions are modular | Bad The logic for each decision is embedded in the states of the FSM. Which makes them hard to configure or reconfigure without needing to reprogram the logic | Good Designers need to design new behavior trees by adding different leaf nodes to a new instance of a tree | Best ActionSets allow game designers to create different configurations of possible decisions for an AI-agent. This is easily done by adding actions to a new list of actions |
| automatic coordination of decisions | Bad Unable to automatically create plans. Plans consisting of different decisions are implemented as FSM states which must be designed manually | Bad Unable to automatically create plans. Plans consisting of different decisions are implemented as tree branches which must be designed manually | Best Done automatically by the planner class in real time (while the game is running) |
| automatic replanning | Bad Does not have a dedicated planner. The FSM's state transitions are the only way for this system to move from one decision to another (replan) | Bad The predesigned tree branches are the only way for the system to change plans. That is if designers have accounted for all possible situations in the game and created suitable | Best Done by the Planner class. Each new memory that is added to the working memory prompts the planner to create a new plan of actions |

| | | | |
|--|---|---|--|
| | | tree branches for them | |
| reusable with different game character types | Bad New FSMs must be designed and programmed for different character types | Good New behavior trees must be designed for different character types. This can be done using preprogrammed leaf nodes. Requires redesigning transitions between leaves | Best Different action sets allow reusability for different character types. No need to redesign the planner nor the individual actions to fit different characters |
| easy to set up | Best Beginner friendly and easy to set up. Many games use this system in their AI-agents. Does not contain many components and is therefore easier to program | Bad Requires good programming knowledge. Several types of leaf nodes need to be programmed before the system becomes functional | Bad Requires a good understanding of different algorithms and several components need to be programmed before the system becomes functional |
| Adopts a way for world representation | Bad Does not support a particular way to represent changes in the game world | Good Uses condition leaves to represent the success or failure states of different behaviors | Best Uses a dedicated working memory to save information about different game objects and game world events. |
| Ease of customization | Bad Game logic is embedded in the states which means different states and their transition logic need to be hardcoded making the game designer's work harder | Good Easy to customize because it allows game designers to design new behavior trees, tree branches and leaves without needing to write new code. Requires redesigning transitions between leaves | Best Easy to customize because it allows game designers to create new actions and action sets without needing to write new code |

Table 4: “bad, good and best” comparison table of the alternative decision-making systems

If a winner had to be selected based on which system had a higher number of *Good* and *Best* in the table, the obvious winner of the comparison would be the GOAP architecture. However, making different comparisons that take different criteria into account would probably yield different results. Therefore, this comparison should be viewed only within the scope of the criteria listed above. It is not a general comparison of the three systems. Ultimately, the comparison was made to contextualize the GOAP system in comparison with other popular decision-making systems. It was not made to determine which AI system was the best one. Every project has different requirements and those are what really matter when selecting an AI decision-making system to work with. To further motivate these results, an analysis of the comparison had to be made:

7.4.1 Analysis of the comparison

The three different types of systems represent decision-making in different ways. GOAP’s decisions are represented as actions, while behavior trees represent decisions as leaf nodes that are called behaviors. And finally, FSMs represent decisions as states.

Managing many decisions:

The Goal Oriented Action Planning architecture had the ability to manage many actions in its planner. Each action was represented as its own object and for this thesis’ project, this was done by developing the *ScriptableAction* class. In comparison, behavior trees had behavior nodes and FSMs had states. FSMs presented a disadvantage when it came to the management of many decisions because decisions were represented as states and an FSM with many states was considered to be an unmanageable system.

Modularity of decisions:

As mentioned earlier, GOAP had a way of representing decisions as actions. This allows the game designer to pick and choose which actions go inside which action sets and that means that the decisions in the system are modular and can be configured in different ways. This was implemented in this thesis’ project in the *ScriptableAction* and *ActionSet* classes in such a way that a character type can have an individual *ActionSet* that contains a list of *ScriptableAction*

instances that represent what decisions that character could make in the game. In comparison, a game designer could configure different behavior trees by adding or removing behavior leaf nodes to a tree. FSM systems fall behind in this domain because they represent decisions as states. Those states are not modular which means they are hardcoded in the FSM's code. Removing or adding new states requires writing more code or removing code from the program.

Automatic coordination of decisions:

GOAP has a dedicated planner that creates plans consisting of heuristically relevant actions and sets them in a correct order for them to be played out. The developed project contained the *Planner* class which was able to receive a goal and create a plan based on that goal and information that is saved in the working memory. The planner examines the available action set and using a regressive search algorithm it finds each relevant action beginning with the action that fulfills the goal back towards the first action that an agent can start a plan with. Neither FSMs nor behavior trees have a similar ability to GOAP's planner. Instead, what could be perceived as plans are predesigned connections between the states in FSMs and between the tree's leaves in behavior trees. The two systems (FSM and behavior tree) are not able to automatically create plans at runtime.

Automatic replanning:

This feature relies on the *Planner* class as well. In the thesis' project, whenever a new memory is added to the AI-agent's working memory, a request is also automatically sent to the planner to create a new plan that takes the newly added memory into consideration. Behavior trees and FSMs do not have planners as previously mentioned; therefore, they fail to match the GOAP architecture in this feature as well.

Reusability with different character types:

The developed GOAP system allowed reusability with a very simple implementation of the *ActionSet* class. Each instance of this class contained a list of actions. Which meant that different characters could very easily be assigned different sets of actions that they were allowed to make. No changes to the planner or other parts of the code were needed to ensure the system's compatibility with each character type. Behavior trees and FSMs do not have a matching way to

easily reuse the system to the game designer's advantage. Instead, the designer has to design different behavior trees for each different character type in their game. The same applies to FSMs where a game designer has to design new FSMs for each different character type.

Ease of setting up:

While the GOAP architecture had many advantages, it could not be compared with the ease of set up of an FSM. By the time a programmer could finish programming their version of a GOAP system without designing any actions, they could already have programmed a finite state machine with its different actions and transitions between them. FSMs are known to be easy to program because their logic is simple. An FSM switches between each state provided some condition was met (ex. If $X = \text{true}$, transition to State 2). On the opposite side, the GOAP system in this thesis project required programming a planner class which took some time to get up and running. It also required programming all the other parts of the GOAP architecture that were described in the literature. Behavior trees are not easy to set up either, considering they have many classes that need to be programmed to represent each type of leaf in the tree (leaves can represent behaviors or conditions among other types).

World representation:

The developed GOAP project used memories to represent the state of the world. These memories were saved in the *WorkingMemory* whenever something relevant happened in the game (ex, another character was seen, key was found, etc.). Behavior trees do not use the same kind of logic, but they do have what is called a Condition leaf inside the tree. These condition leaves represent whether a certain behavior was successful or not. FSMs do not have any particular way of representing changes/statuses in the Gameworld.

Ease of customization:

The developed project allowed for easy customizability of the scope of decisions that could be made. This was because of the modularity of the actions, as new actions could be easily instantiated and assigned costs, effects, and preconditions. This way, a game designer could customize the actions in a game based on costs and context without needing to change the code for the planner to accommodate new actions that are created later in the project. Behavior trees

were also easy to customize because they contain definitions of what a behavior is and what a condition is (among other leaf types). The game designer gets to chain different behaviors and conditions to create new tree branches that could exhibit a desired line of action. FSMs on the other hand do not allow for easy customizability. Game designers need to hardcode every new state in the FSM.

Chapter 8 Discussion

This section presents a concluding discussion about this thesis paper. It summarizes the work as a whole. It discusses AI in games in relation to the design science methodology and discusses the issue of accessibility to game development research.

8.1 Summary of the work as a whole

This thesis began with the aim of understanding the *Goal Oriented Action Planning* architecture and developing a GOAP system. This was to advance the research within the fields of artificial intelligence and game development and make GOAP a better documented system with publicly accessible research. The developed GOAP system was used in a game prototype where there were two different character types (Hunter and hunted) where the hunter type had the goal of catching as many other characters as possible (see *Figure 13*) and the hunted type had the goal of finding special keys that allowed them to escape the game area (see *Figure 14*).



Figure 13: a screenshot from the project showing a “hunter” AI-agent chasing a “hunted” AI-agent



Figure 14: a screenshot from the project showing a “hunted” AI-agent finding and grabbing a key

The thesis also wanted to answer the research question “How does the GOAP architecture compare to other decision-making AI systems in video games? ”. As mentioned earlier, two other decision-making systems were chosen. Those two systems (FSMs and Behavior Trees) were chosen because they were both commonly used in video game development and were very well documented in the literature. The content of the comparison was based on the acquired knowledge about all three systems through reviewing the literature on the topic. Additionally, the developed GOAP system also aided in gaining knowledge for the comparison.

The background section of the thesis introduced the reader to the topic of artificially intelligent game agents in video games. Additionally, it briefly explained why there was a need for dedicated decision-making systems in video games and how those were better than simply using basic “if this then do that” logic. Furthermore, the existing research section introduced the three decision-making systems that this thesis was concerned with comparing. It roughly explained the logic behind them and finally, it introduced the research gap.

The theoretical framework introduced the theoretical discussions that were relevant in the area of artificially intelligent decision-making systems in video games. The idea of the illusion of intelligence was proposed by several professionals as a way of thinking about developing AI for games. This idea helped create boundaries for the scope of the project and eliminated the need to think about game-AI with too much complexity. Furthermore, more concrete theoretical ideas were introduced and those were planner systems, the A* algorithm, heuristics, vision in video games and working memory systems. Finally, the theoretical framework introduced the need to conduct a comparison between different AI-systems in video games and motivated the need for game AI-research because of its social impact on society.

Afterwards, the paper presented the methods and empirical data section where the design science research methodology was explained, multiple steps for an analysis and comparison were introduced and included a literature review for data collection. These were concerned with establishing a way to compare the three different decision-making systems, namely, GOAP, behavior trees and FSMs. And finally, the usefulness of concept maps was explained. Furthermore, the ethics section brought up the possibility of bias in the comparison, due to the

fact that the developed prototype was only concerned with developing a GOAP system and not the two other ones (Behavior tree or FSM).

Finally, the results section was presented. It presented the designed unified modeling language (UML) diagrams and explained the use of game engine software (Unity Engine) where the prototype was programmed and developed. A pseudocode presentation of the main components of the developed GOAP system and other supporting components was shown and an explanation of what each of them was responsible for performing was presented. Those components included the planner, goal, action, plan, actionset, vision sensor and the blackboard. Subsequently, the results of the comparison were presented using a “bad, good, best” table that is based off of the system requirements which were previously outlined in the method section. In addition to that, a more detailed analysis of the comparison results was presented.

8.2 AI in games in design science – process and obstacles

The thesis had the aim of understanding the goal oriented action planning architecture by programming a prototype which implements that architecture. That required setting the prototype in a video game environment in order to truly understand its advantages and limitations as a decision-making system for video game AI-agents. Decision-making systems like GOAP are important because they present a structured solution for game AI. Without such decision-making systems, game programming would easily become unmaintainable and troubled with too many “if this do that” long lines of code. In the literature, it was easy to find documentation for behavior trees and finite state machines and how to program and implement them in video games. The same could not be said about the GOAP architecture however. But that was part of the research gap, and this thesis was written to address that issue. The literature on GOAP that was reviewed for this paper, was more open to interpretation. This meant that it became easier to program the prototype in a way that best fit the chosen game engine (Unity Engine). The purpose of creating a GOAP system for this research paper was to further understand GOAP. This was done from a design science perspective where the research was centered around six different activities. Those were “problem identification and motivation” which included the identification of the research gap, that being a lack of enough clear and easily accessible documentation and

implementation examples of the GOAP system. The second activity was “defining the objectives for a solution” which was done in Table 1 where multiple requirements were outlined for the GOAP system before developing it. The third activity was “design and development” which was done by designing concept maps and UML diagrams which in their turn served as a base for the development of the GOAP system. The fourth activity was “Demonstration” which was done by demonstrating that the developed GOAP system was functional in a virtual game setting for AI game agents in the Unity engine. The fifth activity was “evaluation” and this was done through a comparison between the GOAP system and Behavior trees and FSMs (the latter two being some of the most common AI decision-making systems for games). And finally, the sixth activity in the design science methodology was “communication” which is presented by the culmination of information and implementation documentation of the produced artifact (GOAP system) in this research paper. Making this paper an accessible resource for game developers, hobbyists, and other relevant audiences. And a resource which contributes to the information systems field using a design science approach while also advancing research in game development and AI research.

Furthermore, programming a decision-making system for game AI-agents is not as much of a complicated endeavor as it might seem. The literature supported this idea in the discussion about how game intelligence is an illusion of intelligence. A decision-making system in this case only needed to take into account the least possible information that could help it achieve its goals. The GOAP system does this by using a regressive search algorithm in its planner to find a heuristically relevant chainable line of actions that fulfills a given goal. The set of actions is up for the game designer to develop. Each action in the set has a precondition which must be fulfilled to allow this action to be considered as a part of a plan. Actions also have effects, which are used to create a chain with the next action’s precondition. Meaning that an effect matches the next action’s precondition in the plan. And finally, an action has a cost, which helps the planner choose the cheapest action if there are more than one action with the same effect.

After developing the GOAP planner and its components, it became clear that the rest of the game prototype also had to be programmed in a new way. The idea that a game’s AI-agent’s actions would be chosen by an automated planner presented a new way of game programming. Instead of writing code that immediately tells an AI-agent to do a certain behavior if a certain event was

witnessed, the code for the game prototype had to be written in such a way where there was a high focus on the agent's working memory. The significance of the working memory for such a project was not highlighted well enough in the literature. The working memory acted as a middleman between the agent's planner and the blackboard. This meant that when an agent witnessed an event through their sensory systems, those sensors would add a memory about that event into the working memory. The planner could then do with that knowledge what it sees best and create a plan of actions which are then invoked from the blackboard. This reflects Orkin's idea about seeing the game's agents as those who are in the best position to make their own decisions (Orkin, 2003).

Another important aspect of GOAP powered AI-agents are their sensory systems. These are also not discussed as well enough as they should be in the literature concerning GOAP. However, they were discussed in many other sources of literature because they are important components of video game AI-agents regardless of which decision-making system the game characters have. What is meant by sensors here is for example, vision sensors, audio sensors, proximity sensors, etc. Any game object that can be used to relay Gameworld events to the working memory can be seen as a sensor in this case. In this thesis' game prototype, the developed sensors were a long-range vision sensor, a short-range vision sensor and a proximity sensor. The vision sensors were programmed to create a memory about if a target was seen, and which target it was. It then sends that memory to the working memory so that the planner could reprioritize which actions it should be making.

The comparison part of the paper concluded by illustrating the contextual differences of the three decision-making systems. This comparison was not intended to show which system was the best out of the three but was rather intended to show what things the GOAP system could do that the other decision-making systems could not do. The conclusion was that the GOAP architecture was a system that could support a combination of features, and these were: managing many decisions, having decisions be represented in a modular way, a planner that creates and recreates correct sequences of decisions, a system that can be reused in multiple game character types, a system that has a heuristical way of representing Gameworld events and is easy to customize. On the other hand, if a game designer needed an AI system where they could design their game characters' sequences of actions manually but still be able to create characters that are able to

make many decisions, then a behavior tree could be more suited for that purpose. And finally, if the designer wanted to be able to quickly prototype a game agent's AI and did not need it to do too many actions, then a finite-state-machine AI (FSM) would be the best choice.

Lastly, there are many different artificial intelligence frameworks one could choose from when building a project that utilizes decision-making. Every framework has its pros and cons and no one framework should be deemed the best without considering the requirements of the project in question. The *goal oriented action planning* architecture shines best when used to design agents that are capable of making many choices and decisions. And it truly becomes useful when the need to implement new actions arises, considering its *Planner* class, which does not need to be reprogrammed every time a new action is implemented in the agent's blackboard.

8.3 Ease of access to game development research

Independent game studios that are composed of a small team of developers usually have to work on many things at the same time. This is contrary to bigger companies which might have divisions for every part of a video game's lifecycle going from research to development and even marketing. That is why it is important that more papers are written on video game artificial intelligence systems in a digestible and easy to understand manner. Design science research presents a very appropriate model to do such research where an understanding of a certain artifact is best done by creating one such artifact as part of the research. More research in the field of game AI would enable smaller developers and hobbyists to easily learn from them and implement them in their own projects. This is not only limited to game development however. As was discussed earlier in this paper in "the social impact of game AI research" section, video games have been used as a testbed for artificial intelligence research to test certain software patterns before using them in other domains for example in robotics or smart home products.

Additionally, many organizations use domain specific information systems that are employed to solve problems and process information. Game development studios are among these organizations. They use different information systems such as the GOAP system. This gives them a unified understanding in the organization of what input individual developers can give to

the system and what kinds of output it will produce. Thus, enabling the organization to develop games through a well-structured system.

This paper could help some developers understand the *Goal Oriented Action Planning* architecture, or perhaps introduce it to them for the first time. Advanced decision-making AI systems can help developers create new kinds of games or games that are not usually released by smaller game studios. For example, GOAP can be used to create games or virtual environments where many agents/characters are in one environment and are each taking individual courses of action to achieve separate goals or one common goal. For example, in a game where players can build houses together from materials found in the game, there could be AI-agent characters that help the player by collecting materials to build the house. Finding a material would be set as a goal by the agents' GOAP planners and each agent would go and collect a different material to build the same house.

Finally, doing more accessible research on artificial intelligence systems in games is important because it gives new opportunities to game developers, hobbyists and other communities who do not have access to expensive scientific journals nor the ability to hire their own research teams. These new opportunities could create new jobs, artistic expressions or even lead the path towards new technologies.

Chapter 9 Conclusion

This section presents the main conclusions of the thesis. And then discusses the implications and limitations of this paper and lists possible related work that could be done on the topic in the future.

9.1 Main conclusions

The Goal Oriented Action Planning system presents a useful framework for games that have AI-agents which need to create multi-action plans and alternate between many different actions.

Unlike an FSM or a behavior tree, the GOAP system frees the developer from the need to write code that specifies how each action may precede or proceed the next action in a plan.

Additionally, the GOAP system frees developers from predefining sets of plans altogether.

Instead, the system creates plans during the execution of the program using an algorithm that uses preconditions, effects, and costs to decide which actions could create a suitable plan to achieve a given goal. GOAP uses a set of main technical concepts that should be implemented and those were: Goal, action, action sets, plans, planner. Additionally, a working memory and several sensors can be implemented to give the GOAP system the ability to react to different events in the game world and create plans based on memories.

Different video games require different AI systems depending on the level of variety of actions or decisions that an AI-agent is expected to display. This is an important factor in selecting a type of AI-system to implement in a game. A traditional FSM works best for simple AI-agents with a small handful of actions they can make. Behavior trees are better for more complex behaviors consisting of several actions or decisions. And finally, a GOAP system is good for AI-agents that dynamically create their own plans based on the decisions they need to make to achieve certain goals in a game.

Lastly, doing research using a design science approach allows the researcher to create an artifact of the phenomena or technology that is being researched and thoroughly document it so that different researchers and other relevant communities can benefit from the research. This makes

this research paper not just a source of information that was collected at a certain time, but rather, it serves as a guide for future research and for developers who are interested in developing their own version of the research's artifact. Namely, a GOAP system.

9.2 Limitations and future work

As previously mentioned, this paper only addresses three decision-making systems for game agents' artificial intelligence. In reality, there are more of them in the industry than just the three. Different kinds of projects or games require different solutions of course. Therefore, the takeaway from this thesis paper should not be that the GOAP architecture is the best solution for game AI. Instead, game programmers should carefully consider their game's requirements and choose the best system for the job based on that.

Additionally, a future comparison of the three systems (GOAP, FSM and Behavior Trees) should be done after having programmed three different digital artifacts of each, instead of developing an artifact of only one of them (GOAP). This could further improve the understanding of these systems and how they differ from each other. Furthermore, it could expose more advantages in behavior trees and FSMs which might not have been visible through this paper's literature review.

Because the GOAP architecture makes very good use of working memory and sensory systems, it becomes very intriguing to think of the applications it could have in microelectronics. Perhaps, smaller robotics that have a specific set of actions could make use of a decision-making system such as GOAP. This could be a much better choice than complicated machine learning solutions which might not be suitable for the low-powered processors that microelectronics usually have.

Finally, the domain of decision-making systems continues to make new advancements through experimenting on and developing new video games. Two more decision-making systems that are already used in some modern video games are the *Utility AI* (Walkup, 2021) which makes different calculations based on a given situation and decides the best course of action in a given moment and *Director AI* (Middler, 2021) which monitors different variables in the game such as the player's health, shooting accuracy, etc. and based on that makes decisions about for example,

how many enemies should go after the player, how strong those should be and in which parts of the game they should exist, etc. More research on these systems and especially research that uses a design science approach can be done to expose their internal logic and make them more easily available to learn from for smaller game development teams, hobbyists and others who might be interested. And lastly, this paper did not cover the different ways where multiple kinds of decision-making AI can be combined in a project to achieve different goals. Creating new AI-models based on a combination of multiple decision-making systems could aid in developing more robust and useful solutions.

Chapter 10 References

- AIandGames. (2020, May 20). *Revisiting the AI of Alien: Isolation*,. AI and Games.
<https://www.aiandgames.com/2020/05/20/revisiting-alien-isolation/>
- Al Shehabi, A. (2022). *GitHub - sonic6/GOAP-project* [C# Classes for a Unity project].
<https://github.com/sonic6/GOAP-project/tree/main/GOAP-project/Assets/Scripts>
- *Alien: Isolation*. (2014). [Video game].
https://store.steampowered.com/app/214490/Alien_Isolation/
- Barrera, R., Kyaw, A. S., & Swe, T. N. (2018). *Unity 2017 Game AI Programming - Third Edition: Leverage the power of Artificial Intelligence to program smart entities for your games* (Rev. ed.). Packt Publishing. <https://learning.oreilly.com/library/view/unity-2017-game/9781788477901/>
- Colledanchise, M., & Ogren, P. (2017). How Behavior Trees Modularize Hybrid Control Systems and Generalize Sequential Behavior Compositions, the Subsumption Architecture, and Decision Trees. *IEEE Transactions on Robotics*, 33(2), 372–389.
<https://doi.org/10.1109/tro.2016.2633567>
- Cook, M. (2021). The Social Responsibility of Game AI. *2021 IEEE Conference on Games (CoG)*, 1–8. <https://doi.org/10.1109/cog52621.2021.9619090>
- Cossu, S. M. (2020a). Behaviors. In *Beginning Game AI with Unity: Programming Artificial Intelligence with C#* (1st ed., pp. 117–139). Apress.
https://doi.org/10.1007/978-1-4842-6355-6_5

- Cossu, S. M. (2020b). Pac-Man ghost FSM [Illustration]. In *Beginning Game AI with Unity: Programming Artificial Intelligence with C#* (pp. 117–139).
- *Dead by Daylight*. (2016). [Video Game]. Behaviour Interactive.
https://store.steampowered.com/app/381210/Dead_by_Daylight/
- Fan, J., Fang, L., Wu, J., Guo, Y., & Dai, Q. (2020). From Brain Science to Artificial Intelligence. *Engineering*, 6(3), 248–252. <https://doi.org/10.1016/j.eng.2019.11.012>
- *F.E.A.R.* (2005). [Video Game]. <https://store.steampowered.com/app/21090/FEAR/>
- Fikes, R. E., & Nilsson, N. J. (1971). Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3–4), 189–208.
[https://doi.org/10.1016/0004-3702\(71\)90010-5](https://doi.org/10.1016/0004-3702(71)90010-5)
- Haenlein, M., & Kaplan, A. (2019). A Brief History of Artificial Intelligence: On the Past, Present, and Future of Artificial Intelligence. *California Management Review*, 61(4), 5–14. <https://doi.org/10.1177/0008125619864925>
- Hart, P., Nilsson, N., & Raphael, B. (1968). A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2), 100–107. <https://doi.org/10.1109/tssc.1968.300136>
- Hirschheim, R., & Klein, H. (2012). A Glorious and Not-So-Short History of the Information Systems Field. *Journal of the Association for Information Systems*, 13(4), 188–235. <https://doi.org/10.17705/1jais.00294>
- Lazarevich, V. (2018, October 25). *How to Compare Software Products, Solutions or Frameworks*. Digiteum. <https://www.digiteum.com/how-to-compare-software-solutions-frameworks-libraries-and-other-components/>

- Lewis, M., & Jacobson, J. (2002). GAME ENGINES IN SCIENTIFIC RESEARCH. *Communications of the ACM*, 45(1), 27–31.
<https://www.cse.unr.edu/~sushil/class/gas/papers/GameAIp27-lewis.pdf>
- Lyytinen, K. (1987). Different perspectives on information systems: problems and solutions. *ACM Computing Surveys*, 19(1), 5–46. <https://doi.org/10.1145/28865.28867>
- Middler, J. (2021, November 17). ‘Left 4 Dead’ veterans reveal AI Director 2.0. NME.
<https://www.nme.com/news/gaming-news/left-4-dead-veterans-reveal-ai-director-2-0-3097693>
- Orkin, J. (2003). *Applying Goal-Oriented Action Planning to Games*. MIT Media Lab.
https://alumni.media.mit.edu/~jorkin/GOAP_draft_AIWisdom2_2003.pdf
- Orkin, J. (2004). Symbolic Representation of Game World State: Toward Real-Time Planning in Games. *Proceedings of the AAAI Workshop on Challenges in Game AI*.
<http://alumni.media.mit.edu/~jorkin/WS404OrkinJ.pdf>
- Orkin, J. (2005). Agent Architecture Considerations for Real-Time Planning in Games. *AIIDE’05: Proceedings of the First AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 105–110.
<https://www.aaai.org/Papers/AIIDE/2005/AIIDE05-018.pdf>
- Orkin, J. (2006). Three States and a Plan: The A.I. of F.E.A.R. *Proceedings of the Game Developer’s Conference (GDC)*. Game Developers Conference.
http://alumni.media.mit.edu/~jorkin/gdc2006_orkin_jeff_fear.pdf
- Peffers, K., Tuunanen, T., Rothenberger, M. A., & Chatterjee, S. (2007). A Design Science Research Methodology for Information Systems Research. *Journal of*

Management Information Systems, 24(3), 45–77. <https://doi.org/10.2753/mis0742-1222240302>

- Rabin, S. (2014). *Game AI Pro: Collected Wisdom of Game AI Professionals* (1st ed.). CRC Press. <https://doi.org/10.1201/b16725>
- Rabin, S. (2015). *Game AI Pro 2: Collected Wisdom of Game AI Professionals* (1st ed.). A K Peters/CRC Press. <https://doi.org/10.1201/b18373>
- Rabin, S. (2017). *Game AI Pro 3: Collected Wisdom of Game AI Professionals* (1st ed.). A K Peters/CRC Press. <https://doi.org/10.4324/9781315151700>
- *Resident Evil 2*. (2019). [Video Game]. Capcom.
https://store.steampowered.com/app/883710/Resident_Evil_2/
- *Resident Evil Village*. (2021). [Video Game]. Capcom.
https://store.steampowered.com/app/1196590/Resident_Evil_Village/
- Rowley, J., & Slack, F. (2004). Conducting a literature review. *Management Research News*, 27(6), 31–39. <https://doi.org/10.1108/01409170410784185>
- Sweetser, P., & Wiles, J. (2002). Current AI in games : a review. *Australian Journal of Intelligent Information Processing Systems*, 8(1), 24–42.
https://eprints.qut.edu.au/45741/1/AJIIPS_paper.pdf
- Unity. (2018, October 15). *Unity - Manual: ScriptableObject*. Unity3D.
<https://docs.unity3d.com/Manual/class-ScriptableObject.html#:~:text=A%20ScriptableObject%20is%20a%20data,your%20Project%20has%20a%20Prefab>
- *Unity*. (2021). [Game engine]. Unity Technologies. <https://unity.com>

- Walkup, M. (2021, July 24). *AI Made Easy with Utility AI - Morgan Walkup*. Medium.
<https://medium.com/@morganwalkupdev/ai-made-easy-with-utility-ai-fef94cd36161>