UNIVERISTY OF PIRAEUS - DEPARTMENT OF INFORMATICS

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ – ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

MSc «Informatics»

ΠΜΣ «Πληροφορική»

MSc Thesis

Μεταπτυχιακή Διατριβή

Thesis Title: Τίτλος Διατριβής:	**Creating an AI Agent with human-like behavior using Maslow's hierarchy of needs and Goal Oriented Action Planning in Unity engine.** Δημιουργία ενός Πράκτορα ΤΝ με ανθρώπινη συμπεριφορά, χρήσημοποιώντας την ιεραρχία των αναγκών του Maslow και την αρχιτεκτονική σχεδιασμού δράσης με προσανατολισμό το στόχο (GOAP) στη μηχανή της Unity.
Student's name-surname: Ονοματεπώνυμο φοιτητή:	**Dimitrios Malliaris** Δημήτριος Μάλλιαρης
Father's name: Πατρώνυμο:	**Vasileios** Βασίλειος
Student's ID No: Αριθμός Μητρώου:	ΜΠΠΛ19034
Supervisor: Επιβλέπων:	**Themistoklis Panagiotopoulos, Professor** Θεμιστοκλής Παναγιωτόπουλος, Καθηγητής

July 2022/ Ιούλιος 2022

Acknowledgments/ Ευχαριστίες

Πρώτα από όλα, θα ήθελα να ευχαριστήσω τους γονείς μου, που με στήριξαν και συνεχίζουν να με στηρίζουν σε κάθε μου βήμα, χωρίς τη βοήθεια των οποίων, δεν θα μπορούσα να κυνηγήσω τα όνειρά μου. Επιπρόσθετα, θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή για αυτή τη διπλωματική εργασία, Θεμιστοκλή Παναγιωτόπουλο, του οποίου η καθοδήγηση και η βοήθεια ήταν υψίστης σημασίας κατά την εκτέλεση της παρούσας εργασίας. Τέλος, θα ήθελα να ευχαριστήσω από καρδίας, τη συμφοιτήτρια μου, Μαρία Τράγα, της οποίας η υποστήριξη και βοήθεια, έπαιξε αποφασιστικό ρόλο στο τελικό αποτέλεσμα αυτής της εργασίας.

First of all, I would like to thank my parents, who supported me and continue to support me every step of the way, without whose help I could not pursue my dreams. In addition, I would like to thank the supervising professor for this thesis, Themis Panayiotopoulos, whose guidance and assistance was of the utmost importance in the execution of the present work. Finally, I would like to thank from the bottom of my heart my fellow student, Maria Traga, whose support and help played a decisive role in the final result of this work.

Abstract

Η ανάγκη είναι κάτι που είναι απαραίτητο προκειμένου να ζήσει ένας οργανισμός μία υγιή ζωή. Οι άνθρωποι, όντας τα πιο έξυπνα ζώα, σύμφωνα με τα ανθρώπινα πρότυπα, έχουν πολύ πιο περίπλοκη ιεραρχία αναγκών από άλλα πλάσματα. Το 1943, ο Abraham Maslow πρότεινε την «Ιεραρχία των αναγκών του Maslow» σε μια προσπάθεια να περιγράψει τι οδηγεί και παρακινεί τους ανθρώπους. Αυτή η εργασία εστιάζει στην ιδέα μιας προσομοίωσης που εκτελείται στη μηχανή της Unity, χρησιμοποιώντας πράκτορες τεχνητής νοημοσύνης που «σκέφτονται» και «δρουν» με ανθρώπινο τρόπο. Αυτό επιτυγχάνεται χρησιμοποιώντας ένα σύστημα Δράσης/Στόχου, ακολουθώντας τις αρχές της αρχιτεκτονικής "Δράσης με προσανατολισμό το στόχο" ή GOAP. Η μετάβαση από τον θεωρητικό ορισμό κάθε ανάγκης στην ιεραρχία του Maslow, σε μια πιο απτή αναπαράσταση, δηλαδή σε μια μορφή που μπορεί να γίνει κατανοητή από έναν πράκτορα τεχνητής νοημοσύνης, είναι ένα πολύ δύσκολο επίτευγμα. Αυτό αποδεικνύεται πρόβλημα, ειδικά καθώς ανεβαίνουμε στην ιεραρχία, διότι οι ανάγκες γίνονται όλο και πιο αφηρημένες. Επομένως, σε αυτή την εργασία, η ιδέα της ανάγκης μεταγράφεται στην πολύ πιο απλοϊκή ιδέα ενός Στόχου, που ικανοποιείται μέσω μιας σειράς Δράσεων. Επιπλέον, λόγω των παραπάνω δυσκολιών, σε αυτή την εργασία, θα διερευνήσουμε μόνο κάποιες από τις βιολογικές ανάγκες της ιεραρχίας του Maslow.

Λέξεις-κλειδιά: Τεχνιτή Νοημοσύνη, Έξυπνοι Πράκτορες, Ιεραρχία των αναγκών του Maslow, Σχεδιασμός δράσης με προσανατολισμό το στόχο, Μηχανή Unity

A need is something that is necessary for an organism to live a healthy life. Humans, being the most intelligent animals, according to human standards, have a much more complex hierarchy of needs than other creatures. In 1943, Abraham Maslow proposed "Maslow's hierarchy of needs" in an attempt to describe what drives and motivates us, human beings. This project focuses on the idea of a simulation run on the Unity engine, using AI agents that "think" and "act" in a human-like way, by use of an Action/Goal system, following the principles of the "Goal Oriented Action Planning" or GOAP architecture. The transition from the theoretical definition of each need in Maslow's hierarchy, to a more tangible representation in a form that can be understood by an AI agent is a very challenging task. This proves to be a problem, especially as we move up the hierarchy because the needs become more and more abstract. Therefore, the idea of need is transcribed to the much more simplistic idea of a Goal, that is satisfied through a series of Actions. Additionally, due to the above difficulties, in this project, we will explore only needs from the "physiological" level of Maslow's hierarchy.

Keywords: Artificial Intelligence, Intelligent Agents, Maslow's hierarchy of needs, Goal Oriented Action Planning, Unity engine

Contents

List of Figures

List of Abbreviations

2D	Two-dimensional
3D	Three-dimensional
AI	Artificial intelligence
API	Application programming interface
CPU	Central processing unit
DOTA	Defense of the ancients
FSA	Finite state automata
FSM	Finite state machine
GOAP	Goal oriented action planning
GPU	Graphics processing unit
LSTM	Long short-term memory
MCTS	Monte Carlo tree search
ML	Machine learning
MOBA	Multiplayer online battle arena
NPC	Non-player character
OS	Operating system
STRIPS	Stanford Research Institute Problem Solver
UI	User interface
VR	Virtual reality

Introduction

Artificial Intelligence

Artificial intelligence (AI) is the ability of a digital computer or computer-controlled robot to perform tasks commonly associated with intelligent beings. The term is frequently applied to the project of developing systems endowed with the intellectual processes characteristic of humans, such as the ability to reason, discover meaning, generalize, or learn from past experience.

Artificial intelligence is a crossroads between multiple sciences such as computer science, psychology, philosophy, neurology, linguistics and engineering, with the aim of synthesizing intelligent behavior, with elements of reasoning, learning and adaptation to the environment, while usually applied in specially designed machines or computers. It is divided into symbolic artificial intelligence, which attempts to simulate human intelligence algorithmically using high-level symbols and logical rules, and sub-symbolic artificial intelligence, which attempts to reproduce human intelligence using elementary numerical models that synthesize inductively intelligent behaviors with the successive self-organization of simpler structural components ("behavioral artificial intelligence"), simulate real biological processes such as the evolution of species and the function of , or are the application of statistical methodologies to AI problems.

Artificial intelligence (AI) is intelligence demonstrated by machines, as opposed to the natural intelligence displayed by animals including humans. Leading AI textbooks define the field as the study of "intelligent agents": any system that perceives its environment and takes actions that maximize its chance of achieving its goals.

An agent is just something that acts (agent comes from the Latin agere, to do). But computer agents are expected to have other attributes that distinguish them from mere "programs," such as operating under autonomous control, perceiving their environment, persisting over a prolonged time period, adapting to change, and being capable of taking on another's goals. [Artificial Intelligence A Modern Approach Second Edition Stuart J. Russell and Peter Norvig, 2003]

We can define AI as the study of agents that receive percepts from the environment and perform actions. Each such agent implements a function that maps percept sequences to actions. In general, an AI system can be described as:

- “A system that thinks like humans”,
- “A system that acts like humans”,
- “A system that thinks rationally” or
- “A system that acts rationally”.

A.I. in video games

In video games, artificial intelligence (AI) is used to generate responsive, adaptive or intelligent behaviors primarily in non-player characters (NPCs) similar to human-like intelligence. Artificial intelligence has been an integral part of video games since their inception in the 1950s. [Grant, Eugene F.; Lardner, Rex (2 August 1952). "The Talk of the Town – It". The New Yorker]

AI in video games is a distinct subfield and differs from academic AI. It serves to improve the game-player experience rather than machine learning or decision making. During the golden age of arcade video games the idea of AI opponents was largely popularized in the form of graduated difficulty levels, distinct movement patterns, and in-game events dependent on the player's input. Modern games often implement existing techniques such as pathfinding and decision trees to guide the actions of NPCs. AI is often used in mechanisms which are not immediately visible to the user, such as data mining and procedural-content generation. [Yannakakis, Geogios N (2012). "Game AI revisited"]

However, "game AI" does not, in general, as might be thought and sometimes is depicted to be the case, mean a realization of an artificial person corresponding to an NPC, in the manner of say, the Turing test or an artificial general intelligence. The term "game AI" is used to refer to a broad set of algorithms that also include techniques from control theory, robotics, computer graphics and computer science in general, and so video game AI may often not constitute "true AI" in that such techniques do not necessarily facilitate computer learning or other standard criteria, only constituting "automated computation" or a predetermined and limited set of responses to a predetermined and limited set of inputs.

AI in gaming refers to responsive and adaptive video game experiences. These AI-powered interactive experiences are usually generated via non-player characters, or NPCs, that act intelligently or creatively, as if controlled by a human game-player. AI is the engine that determines an NPC’s behavior in the game world.

AI in gaming is all about enhancing a player’s experience. It is especially important as developers deliver gaming experiences to different devices. No longer is gaming simply a choice between console or desktop computer. Rather, players expect immersive game experiences on a vast array of mobile and wearable devices, from smartphones to VR headsets, and more. AI enables developers to deliver console-like experiences across device types.

Evolution of A.I. in video games

The origins of Artificial Intelligence in video games go back to before industry itself became a staple of popular culture worldwide. One of the best known precedents for using this technology in a game dates back to the 1950s, with computer chess titles developed based on the MiniMax algorithm. This software was able to analyze the position of the tracks on the dashboard to select the best possible movement. Beginning in the 1970s, with video games revolutionizing the market and the entertainment experience of users, Artificial Intelligence began to play a key role in conceiving and developing the titles of major brands.

In 1972 Pong, your opponent was already able to move based on the movement of the ball you hit to return the bump, while the Space Invaders gave clear clues as to how the enemies evolved and became smarter, challenging players. spending hours in the arcade trying to get through each of the levels.

For his part, in the eighties Pac-Man took another step, as it was the first video game to feature a path search system for enemies, who could more easily decipher the path the player took under certain conditions. In the same years, Donkey Kong boasted how far Artificial Intelligence had advanced, with a level of difficulty that pushed gamers of the time to the limit. However, Artificial Intelligence was still precarious enough to realize that the machine was not able to learn from its mistakes, nor to adapt to the player's behavior, so the latter had to decipher certain patterns to overcome the most demanding challenges.

Navigation in 3D space

Pathfinding is another common use for AI and is widely seen in real-time strategy games. Pathfinding is the method for determining how to get a NPC from one point on a map to another, taking into consideration the terrain, obstacles and possibly "fog of war". [Yannakakis, G. N. (2012, May). Game AI revisited]

Commercial videogames often use fast and simple "grid-based pathfinding", wherein the terrain is mapped onto a rigid grid of uniform squares and a pathfinding algorithm such as A* or IDA* is applied to the grid. [Abd Algfoor, Zeyad; Sunar, Mohd Shahrizal; Kolivand, Hoshang (2015). "A Comprehensive Study on Pathfinding Techniques for Robotics and Video Games". 2015][Yap, Peter. "Grid-based path-finding." In Conference of the Canadian Society for Computational Studies of Intelligence, pp. 44-55. Springer, Berlin, Heidelberg, 2002][Sturtevant, N. R. (June 2012). "Benchmarks for Grid-Based Pathfinding". IEEE Transactions on Computational Intelligence and AI in Games]

Instead of just a rigid grid, some games use irregular polygons and assemble a navigation mesh out of the areas of the map that NPCs can walk to. [Abd Algfoor, Zeyad; Sunar, Mohd Shahrizal; Kolivand, Hoshang (2015). "A Comprehensive Study on Pathfinding Techniques for Robotics and Video Games". 2015][Goodwin, S. D., Menon, S., & Price, R. G. (2006). Pathfinding in open terrain.]

As a third method, it is sometimes convenient for developers to manually select "waypoints" that NPCs should use to navigate; the cost is that such waypoints can create unnatural-looking movement. In addition, waypoints tend to perform worse than navigation meshes in complex environments.

Architectures

(FSM) Finite State Machines

A finite-state machine (FSM) or finite-state automaton (FSA, plural: automata), finite automaton, or simply a state machine, is a mathematical model of computation. It is an abstract machine that can be in exactly one of a finite number of states at any given time. The FSM can change from one state to another in response to some inputs; the change from one state to another is called a transition. [Wang, Jiacun (2019). Formal Methods in Computer Science. CRC Press. p. 34.]

An FSM is defined by a list of its states, its initial state, and the inputs that trigger each transition. Finite-state machines are of two types—deterministic finite-state machines and non-deterministic finite-state machines. ["Finite State Machines – Brilliant Math & Science Wiki". brilliant.org. Retrieved 2018-04-14.] A deterministic finite-state machine can be constructed equivalent to any non-deterministic one.

The FSM algorithm was introduced to video game design in the 1990s. In an FSM, a designer generalizes all possible situations that an AI could encounter, and then programs a specific reaction for each situation. Basically, a FSM AI would promptly react to the human player’s action with its pre-programmed behavior. An obvious drawback of FSM design is its predictability. All NPCs’ behaviors are pre-programmed, so after playing an FSM-based game a few times, a player may lose interest.

(MCTS)Monte Carlo Tree Search

In computer science, Monte Carlo tree search (MCTS) is a heuristic search algorithm for some kinds of decision processes, most notably those employed in software that plays board games. In that context MCTS is used to solve the game tree.

MCTS was combined with neural networks in 2016 for computer Go. It has been used in other board games like chess and shogi, games with incomplete information such as bridge and poker, as well as in turn-based-strategy video games (such as Total War: Rome II ‘s implementation in the high level campaign AI).

The focus of MCTS is on the analysis of the most promising moves, expanding the search tree based on random sampling of the search space. The application of Monte Carlo tree search in games is based on many playouts, also called roll-outs. In each playout, the game is played out to the very end by selecting moves at random. The final game result of each playout is then used to weight the nodes in the game tree so that better nodes are more likely to be chosen in future playouts.

The most basic way to use playouts is to apply the same number of playouts after each legal move of the current player, then choose the move which led to the most victories.[Brügmann, Bernd (1993). Monte Carlo Go] The efficiency of this method—called Pure Monte Carlo Game Search—often increases with time as more playouts are assigned to the moves that have frequently resulted in the current player's victory according to previous playouts. Each round of Monte Carlo tree search consists of four steps: [G.M.J.B. Chaslot; M.H.M. Winands; J.W.H.M. Uiterwijk; H.J. van den Herik; B. Bouzy (2008). "Progressive Strategies for Monte-Carlo Tree Search"]

- Selection: Start from root R and select successive child nodes until a leaf node L is reached. The root is the current game state and a leaf is any node that has a potential child from which no simulation (playout) has yet been initiated. The section below says more about a way of biasing choice of child nodes that lets the game tree expand towards the most promising moves, which is the essence of Monte Carlo tree search.
- Expansion: Unless L ends the game decisively (e.g. win/loss/draw) for either player, create one (or more) child nodes and choose node C from one of them. Child nodes are any valid moves from the game position defined by L.

- Simulation: Complete one random playout from node C. This step is sometimes also called playout or rollout. A playout may be as simple as choosing uniform random moves until the game is decided (for example in chess, the game is won, lost, or drawn).
- Backpropagation: Use the result of the playout to update information in the nodes on the path from C to R.

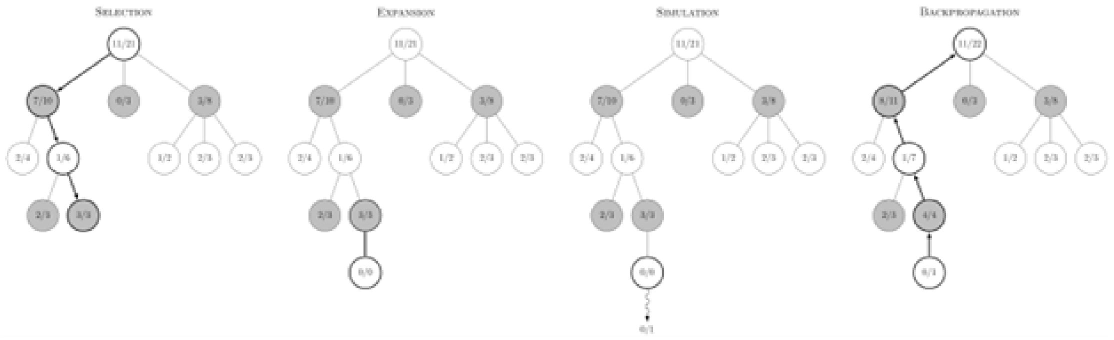


Figure 1 Monte Carlo Tree Search steps.

(ML) Machine Learning

Although AI designers worked very hard to make NPCs look intelligent, these characters lacked one very important trait: the ability to learn. In most video games, NPCs' behavior patterns are programmed and they are incapable of learning anything from players, e.g. they don't evolve based on human players' input.

Machine learning agents have been used to take the place of a human player rather than function as NPCs, which are deliberately added into video games as part of designed gameplay. Deep learning agents have achieved impressive results when used in competition with both humans and other artificial intelligence agents. [Silver, David; Hubert, Thomas; Schrittwieser, Julian; Antonoglou, Ioannis; Lai, Matthew; Guez, Arthur; Lanctot, Marc; Sifre, Laurent; Kumaran, Dharshan (2018-12-06). "A general reinforcement learning algorithm that masters chess, shogi, and Go through self]

In general, there is a variety of Machine Learning techniques that have been used for training ML agents. These techniques include but are not limited to: Deep learning, Convolutional neural networks, Recurrent neural networks, Long short-term memory, Reinforcement learning and Neuroevolution.

(OpenAI Five) DOTA2

Dota 2 is a multiplayer online battle arena (MOBA) game. Like other complex games, traditional AI agents have not been able to compete on the same level as professional human players. The only widely published information on AI agents attempted on Dota 2 is OpenAI's deep learning Five agent.

OpenAI Five utilized separate LSTM networks to learn each hero. It was trained using a reinforcement learning technique known as Proximal Policy Learning running on a system containing 256 GPUs and 128,000 CPU cores. Five trained for months, accumulating 180 years of game experience each day, before facing off with professional players. It was eventually able to beat the 2018 Dota 2 esports champion team in a 2019 series of games. [OpenAI Five] [Dota 2 with Large Scale Deep Reinforcement Learning, OpenAI, March 10, 2021]

The Drawbacks

Machine learning agents are often not covered in many game design courses. Previous use of machine learning agents in games may not have been very practical, as even the 2015 version of AlphaGo took hundreds of CPUs and GPUs to train to a strong level. [Silver, David; Hubert, Thomas; Schrittwieser, Julian; Antonoglou, Ioannis; Lai, Matthew; Guez, Arthur; Lanctot, Marc; Sifre, Laurent; Kumaran, Dharshan (2018-12-06). "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play"] This potentially limits the creation of highly effective deep learning agents to large corporations or extremely wealthy individuals. The extensive training time of neural network based approaches can also take weeks on these powerful machines.["AlphaStar: Mastering the Real-Time Strategy Game StarCraft II". DeepMind. Retrieved 2019-06-04]

The problem of effectively training artificial neural networks based models extends beyond powerful hardware environments; finding a good way to represent data and learn meaningful things from it is also often a difficult problem. Artificial neural networks models often overfit to very specific data and perform poorly in more generalized cases. AlphaStar shows this weakness, despite being able to beat professional players, it is only able to do so on a single map when playing a mirror protoss matchup. OpenAI Five also shows this weakness, it was only able to beat professional players when facing a very limited hero pool out of the entire game.[Dota 2 with Large Scale Deep Reinforcement Learning, OpenAI, March 10, 2021] This example shows how difficult it can be to train a deep learning agent to perform in more generalized situations.

Machine learning agents have shown great success in a variety of different games. However, agents that are too competent also risk making games too difficult for new or casual players. Research has shown that challenges that are too far above a player's skill level will ruin lower player enjoyment.[Sweetser, Penelope; Wyeth, Peta (2005-07-01). "GameFlow". Computers in Entertainment 3 (3): 3] These highly trained agents are likely only desirable against very skilled human players who have many hours of experience in a given game. Given these factors, highly effective deep learning agents are likely only a desired choice in games that have a large competitive scene, where they can function as an alternative practice option to a skilled human player.

(GOAP) Goal Oriented Action Planning

Goal-Oriented Action Planning refers to a simplified STRIPS-like planning architecture specifically designed for real-time control of autonomous character behavior in games.

GOAP is an artificial intelligence system for autonomous agents that allows them to dynamically plan a sequence of actions to satisfy a set goal. The sequence of actions selected by the agent is contingent on both the current state of the agent and the current state of the world, hence despite two agents being assigned the same goal; both agents could select a completely different sequence of actions.

Before we can discuss the benefits of GOAP, we first need to define some terminology. An agent uses a planner to formulate a sequence of actions that will satisfy some goal. We need to define what we mean by the terms goal, action, plan, and formulate.

Goal

A goal is any condition that an agent wants to satisfy. An agent may have any number of goals. At any instant, one goal is active, controlling the character’s behavior. A goal knows how to calculate its current relevance, and knows when it has been satisfied.

Plan

The plan is simply the name for a sequence of actions. A plan that satisfies a goal refers to the valid sequence of actions that will take a character from some starting state to some state that satisfies the goal.

Action

An action is a single step within a plan that makes a character do something. The duration of an action may be short or infinitely long. Each action knows when it is valid to run, and what it will do to the game world. In other words, an action knows its preconditions and effects. Preconditions and effects provide a mechanism for chaining actions into a valid sequence. For example, the RelaxSitting action has a precondition for the agent to be sitting. Consequently, the SitDown action followed by RelaxSitting action is a valid sequence of actions. Each action may have any number of preconditions and effects.

Plan Generation

A character generates a plan in real-time by supplying some goal to satisfy a system called a planner. The planner searches the space of actions for a sequence that will take the character from his starting state to his goal state. This process is referred to as formulating a plan. If the planner is successful, it returns a plan for the character to follow to direct his behavior. The character follows this plan to completion, invalidation, or until another goal becomes more relevant. If another goal activates, or the plan in-progress becomes invalid for any reason, the character aborts the current plan and formulates a new one.

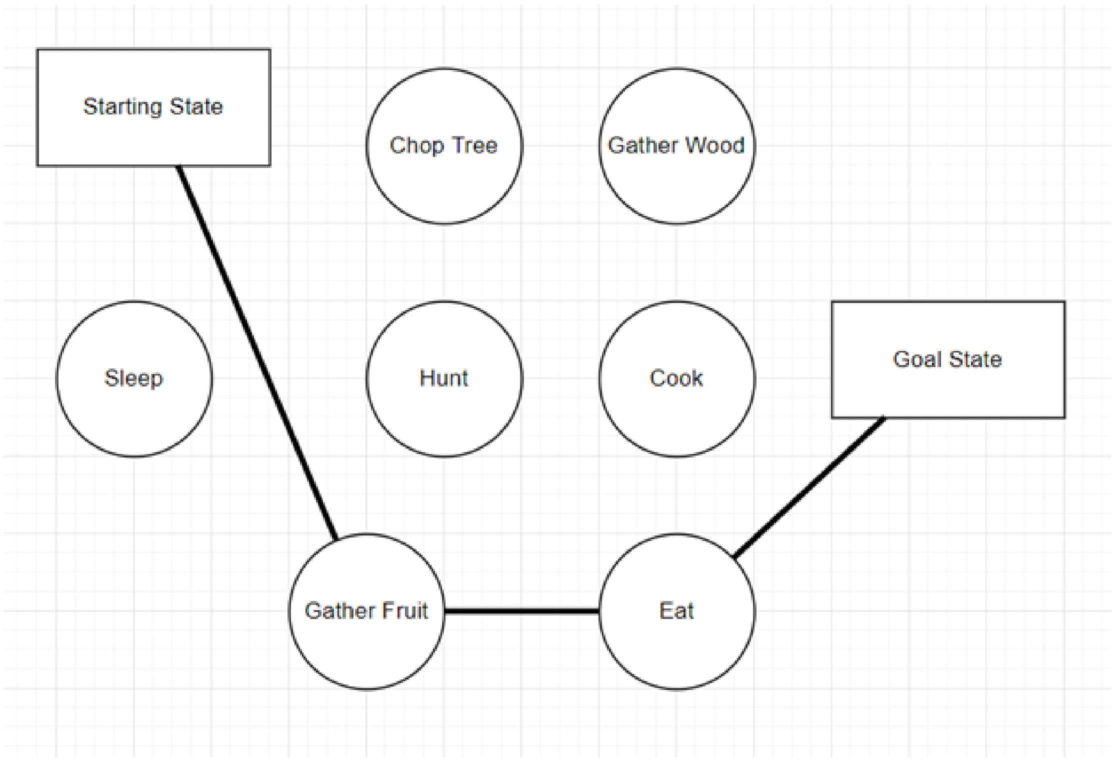


Figure 2 Plan generation example in GOAP

The above figure depicts an abstract illustration of the planning process. The rectangles represent the start and goal states, and each circle represents an action. The goal in this figure is for the agent to eat. The planner needs to find a sequence of actions for the agent that will take the world from a state in which the agent is hungry to a state in which the agent's hunger is satisfied.

In a sense, the above process is pathfinding. The planner needs to find a path through the space of actions that will take the agent from his starting state to some goal state. Each action is a step on that path that changes the state of the world in some way. Preconditions of the actions determine when it is valid to move from one action to the next.

In many cases, more than one valid plan exists. The planner only needs to find one of them. Similar to navigational pathfinding, the planner's search algorithm can be provided with hints to guide the search. For example, costs can be associated with actions, leading the planner to find the least costly sequence of actions, rather than any arbitrary sequence. [Applying Goal-Oriented Action Planning to Games, Jeff Orkin – Monolith Productions, http://www.jorkin.com]

Advantages

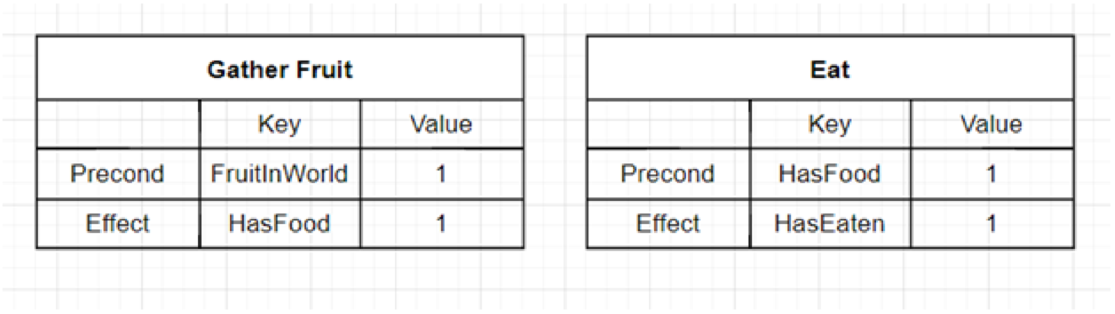
A GOAP system does not replace the need for a finite-state machine (FSM), but greatly simplifies the required FSM. A plan is a sequence of actions, where each action represents a state transition. By separating the state transition logic from the states themselves, the underlying FSM can be much simpler.

There are benefits at both development and runtime. Characters in the game can exhibit more varied, complex, and interesting behaviors using GOAP. The code behind the behaviors is more structured, re-usable, and maintainable. GOAP offers a much more elegant structure that better accommodates change. The addition of design requirements is handled by adding actions, and preconditions to related actions. This is more intuitive, and touches less code than revisiting every goal. Furthermore, GOAP provides the guarantee of valid plans. Hand-coded embedded plans can contain mistakes. A developer might code a sequence of actions that cannot follow each other. For example, a character might be instructed to fire a weapon, without ever being told to first draw a weapon. This situation cannot arise in a plan dynamically generated through a GOAP system, because the preconditions on the actions prevent the planner from formulating an invalid plan.

The structure imposed by GOAP is ideal for creating a variety of agent types who exhibit different behaviors, and even share behaviors across multiple projects. The planner is provided with a pool of actions from which to search for a plan. This pool does not necessarily have to be the complete pool of all existing actions. Different agent types may use subsets of the complete pool, leading to a variety of behaviors. [Applying Goal-Oriented Action Planning to Games, Jeff Orkin – Monolith Productions, http://www.jorkin.com]

World Representation

In order to search the space of actions, the planner needs to represent the state of the world in some way that lets it easily apply the preconditions and effects of actions, and recognize when it has reached the goal state. One compact way to represent the state of the world is with a list of KeyValue Pairs. Consequently, each action must contain a KeyValue List for its preconditions and one for its Effects as seen below: [Applying Goal-Oriented Action Planning to Games, Jeff Orkin – Monolith Productions, http://www.jorkin.com]



Gather Fruit		
	Key	Value
Precond	FruitInWorld	1
Effect	HasFood	1

Eat		
	Key	Value
Precond	HasFood	1
Effect	HasEaten	1

Figure 3 GOAP actions example

Unity Engine

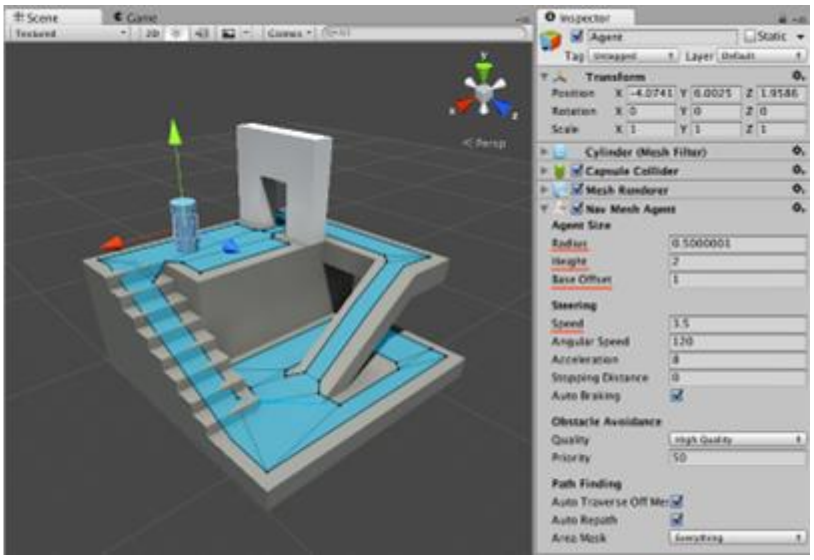
Unity is a cross-platform game engine developed by Unity Technologies, first announced and released in June 2005 at Apple Inc.'s Worldwide Developers Conference as a Mac OS X-exclusive game engine. The engine has since been gradually extended to support a variety of desktop, mobile, console and virtual reality platforms.

Figure 4 Unity engine logo.

The engine can be used to create three-dimensional (3D) and two-dimensional (2D) games, as well as interactive simulations and other experiences. The engine has been adopted by industries outside video gaming, such as film, automotive, architecture, engineering, construction, and the United States Armed Forces.

NavMesh Agent

Unity offers its own navigation tool through the use of the NavMesh Agent. The NavMesh Agent is a component that helps you to create characters which avoid each other while moving towards their goal. Agents reason about the game world using the NavMesh and they know how to avoid each other as well as other moving obstacles. Pathfinding and spatial reasoning are handled using the scripting API of the NavMesh Agent. [Unity Docs]



Agent Static
Tag Untagged Layer Default

Transform
Position X -4.0741 Y 0.0025 Z 1.9586
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Cylinder (Mesh Filter)
Capsule Collider
Mesh Renderer
Nav Mesh Agent

Agent Size
Radius 0.5000001
Height 2
Base Offset 1

Steering
Speed 3.5
Angular Speed 120
Acceleration 8
Stopping Distance 0
Auto Braking ✓

Obstacle Avoidance
Quality High Quality
Priority 50

Path Finding
Auto Traverse Off Me ✓
Auto Repath ✓
Area Mask Everything

Figure 5 NavMeshAgent example scene.

MonoBehaviour

MonoBehaviour is the base class from which every Unity script derives. When you use C#, you must explicitly derive from MonoBehaviour. Running a Unity script executes a number of event functions in a predetermined order. These event functions are called in a sequence called script lifecycle. The event functions that are more heavily used by developers and are referenced in this project, are the Awake(), Start(), Update(), FixedUpdate() methods. [Unity Docs]

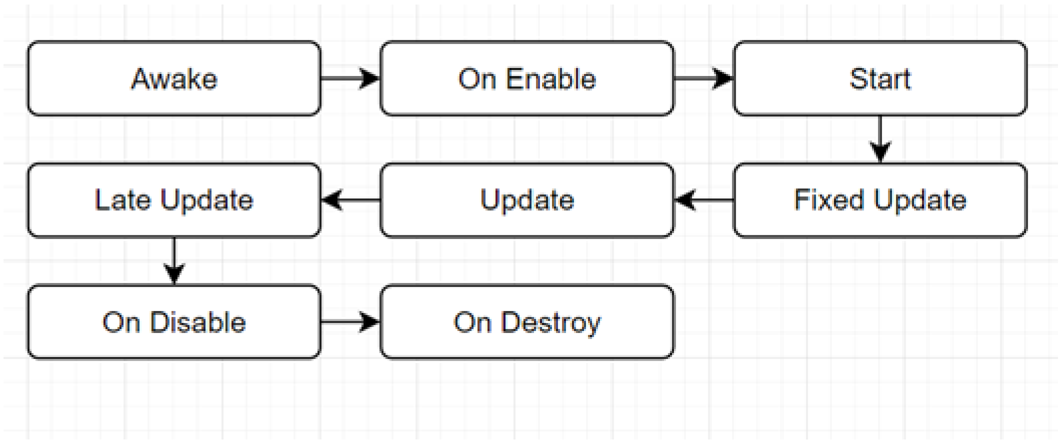


Figure 6 Some of the most common lifecycle events in Unity scripts

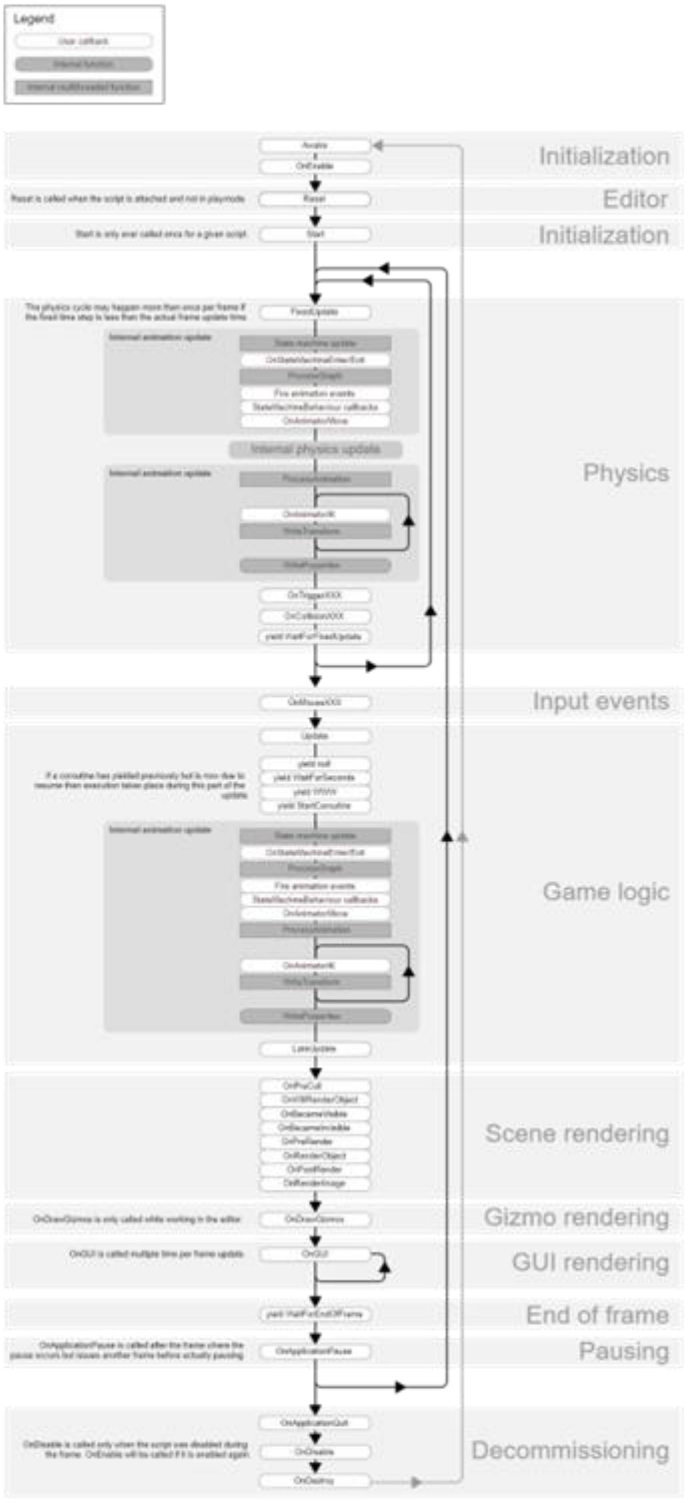


Figure 7 The full lifecycle event list of Unity.

ScriptableObjects

A ScriptableObject is a data container that you can use to save large amounts of data, independent of class instances. One of the main use cases for ScriptableObjects is to reduce your Project's memory usage by avoiding copies of values. This is useful if your Project has a Prefab that stores unchanging data in attached MonoBehaviour scripts. [Unity Docs]

Every time you instantiate that Prefab, it will get its own copy of that data. Instead of using the method, and storing duplicated data, you can use a ScriptableObject to store the data and then access it by reference from all of the Prefabs. This means that there is one copy of the data in memory. [Unity Docs]

Just like MonoBehaviours, ScriptableObjects derive from the base Unity object but, unlike MonoBehaviours, you can not attach a ScriptableObject to a GameObject. Instead, you need to save them as Assets in your Project. [Unity Docs]

Prefabs

Unity's Prefab system allows you to create, configure, and store a GameObject complete with all its components, property values, and child GameObjects as a reusable Asset. The Prefab Asset acts as a template from which you can create new Prefab instances in the Scene. [Unity Docs]

When you want to reuse a GameObject configured in a particular way – like a non-player character (NPC), prop or piece of scenery – in multiple places in your Scene, or across multiple Scenes in your Project, you should convert it to a Prefab. This is better than simply copying and pasting the GameObject, because the Prefab system allows you to automatically keep all the copies in sync. [Unity Docs]

Any edits that you make to a Prefab Asset are automatically reflected in the instances of that Prefab, allowing you to easily make broad changes across your whole Project without having to repeatedly make the same edit to every copy of the Asset. [Unity Docs]

Human Needs

A need is something that is necessary for an organism to live a healthy life. Needs are distinguished from wants. In the case of a need, a deficiency causes a clear adverse outcome: a dysfunction or death. In other words, a need is something required for a safe, stable and healthy life (e.g. air, water, food, land, shelter) while a want is a desire, wish or aspiration. When needs or wants are backed by purchasing power, they have the potential to become economic demands.

Basic needs such as air, water, food and protection from environmental dangers are necessary for an organism to live. In addition to basic needs, humans also have needs of a social or societal nature such as the human need to socialize or belong to a family unit or group. Needs can be objective and physical, such as the need for food, or physical and subjective, such as the need for self-esteem.

Needs and wants are a matter of interest in, and form a common substrate for, the fields of philosophy, biology, psychology, social science, economics, marketing and politics.

Maslow's Hierarchy of Needs

To most psychologists, need is a psychological feature that arouses an organism to action toward a goal, giving purpose and direction to behavior.

Maslow's Hierarchy of Needs The most widely known academic model of needs was proposed by psychologist, Abraham Maslow, in 1943. His theory proposed that people have a hierarchy of psychological needs, which range from basic physiological or lower order through to the higher order needs such as self-actualization. People tend to spend most of their resources (time, energy and finances) attempting to satisfy their basic physiological needs, before the higher order needs of belonging, esteem and self-actualization become meaningful. Maslow's approach is a generalized model for understanding human motivations in a wide variety of contexts, but must be adapted for specific contexts. The theory was further developed by over a course of almost thirty years into a broader theory on human behavior (Maslow, 1970), however the basic principles of the theory remained the same. [Nikos Avradinis*, Themis Panayiotopoulos and George Anastassakis, 2013 "Behavior believability in virtual worlds: agents acting when they need to"]

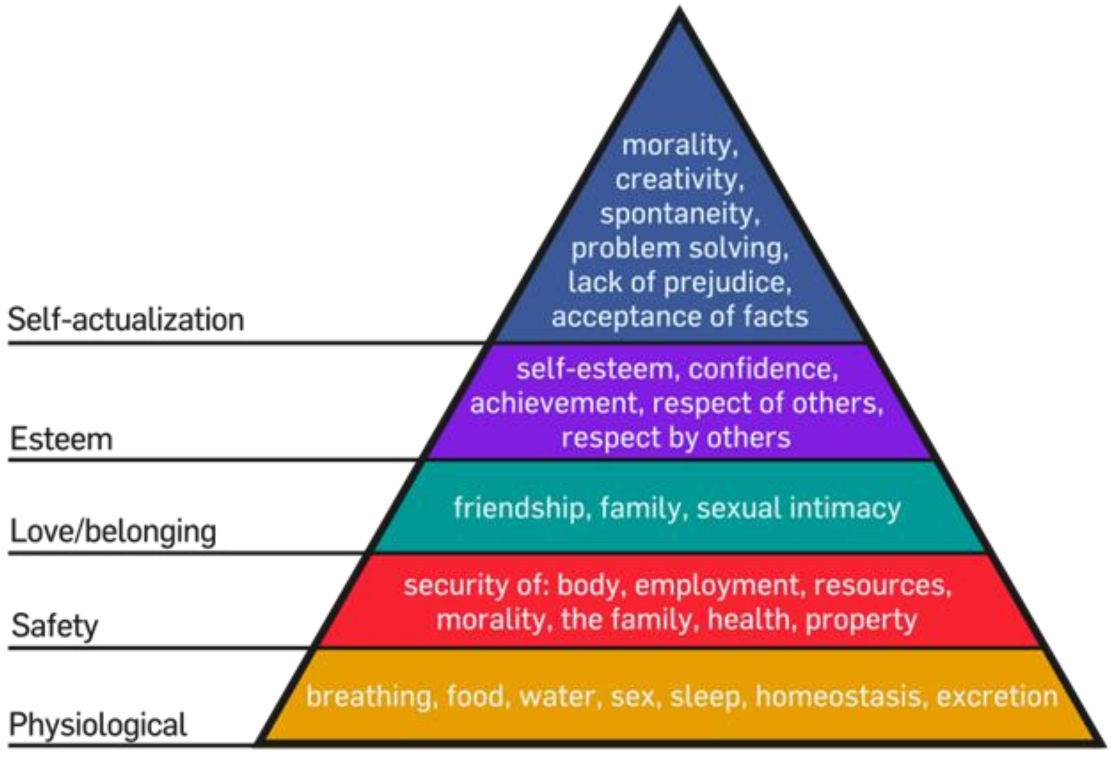

morality,
creativity,
spontaneity,
problem solving,
lack of prejudice,
acceptance of facts
Self-actualization
self-esteem, confidence,
achievement, respect of others,
respect by others
Esteem
friendship, family, sexual intimacy
Love/belonging
security of: body, employment, resources,
morality, the family, health, property
Safety
breathing, food, water, sex, sleep, homeostasis, excretion
Physiological

Figure 8 Maslow's hierarchy of needs, represented as a pyramid of needs

Hierarchy's Structure

Physiological

As mentioned before and as it showed in the last figure, at the lowest level of Maslow's hierarchy of needs lie one's physiological or biological needs. These correspond to basic needs such as hunger, thirst, need for air, sleep, sex etc. When these needs are not satisfied, they may result in emotional and physical discomfort and even threaten one's survival.

Safety

Right above physiological needs, lies the need for Safety, which includes physical safety, occupation and financial security.

Love & Belonging

Love and belonging needs refer to social needs such as belonging to a group, having friends, family and a partner.

Esteem

Esteem needs have both a social and a self-centered character. They concern self-esteem, which is related to knowledge, competence and mastery, as well as confidence which sources from others' respect, general attitude and appreciation towards the individual.

Self-Actualization

At the top of the pyramid is Self-Actualization needs. These needs are more abstract and are generally defined as the realization of one's full capacity and potential.

According to Maslow's theory, the hierarchy is characterized by three basic principles [Reeve J (2010) Understanding Motivation and Emotion. John Wiley & Sons, Hoboken, NJ]:

1. Needs are arranged according to their urgency and their intensity. Lower-level needs are felt stronger and more urgently than higher level ones.
2. Lower-level needs appear earlier in human development.
3. Needs are satisfied sequentially, one level at a time, in bottom to top order; higher level needs are not activated before lower-level needs are satisfied (a concept known as fulfillment progression).

Deficiency VS Growth Needs

Needs can also be distinguished into two categories, deficiency and growth needs [Reeve J (2010) Understanding Motivation and Emotion. John Wiley & Sons, Hoboken, NJ].

Deficiency needs indicate the lack of a basic resource, or an experience. They are like internal resources that have to be replenished regularly. Failing to satisfy a deficiency need puts one in a state of deprivation, which can threaten one's physiological or emotional state of well-being.

Growth needs differ to deficiency needs, in that they do not arise because of a deprivation experience, but rather from an innate drive to evolve and develop oneself. Growth needs emerge only when all deficiency needs have been satisfied, and a need to fulfill personal potential emerges.

Deficiency needs produce simpler and more stereotypical behaviors that are responses to an intense and urgent internal stimulus and aim towards satiation. Growth needs, on the other hand, are less urgent and intense but more elaborate, they can produce diverse behaviors consisting of potentially long sequences of actions and are constructive in nature. We could characterize deficiency needs as reactive and growth needs as generative. [Nikos Avradinis*, Themis Panayiotopoulos and George Anastassakis, 2013 "Behavior believability in virtual worlds: agents acting when they need to"]

Conclusion

In conclusion, Maslow's hierarchy of needs is used to study how humans intrinsically partake in behavioral motivation. Maslow used the terms "physiological," "safety," "belonging and love," "social needs" or "esteem," and "self-actualization" to describe the pattern through which human motivations generally move. This means that in order for motivation to arise at the next stage, each stage must be satisfied within the individual themselves. Additionally, this hierarchy is a main base in knowing how effort and motivation are correlated when discussing human behavior. Each of these individual levels contains a certain amount of internal sensation that must be met in order for an individual to complete their hierarchy. The goal in Maslow's hierarchy is to attain the fifth level or stage: self-actualization.

In-Game Representation

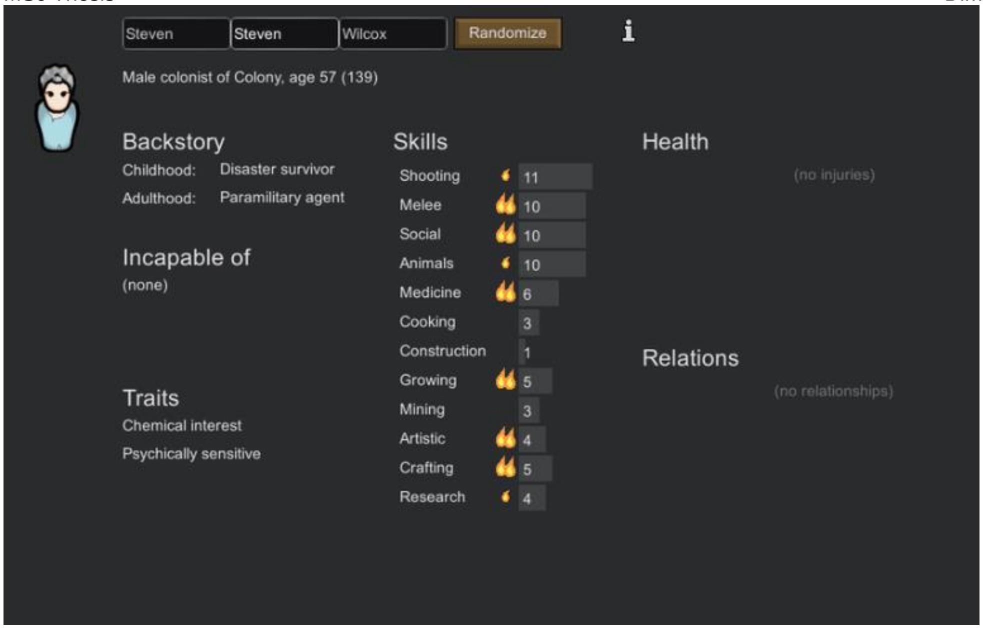
Representing Maslow's hierarchy of needs in a simulation using an AI agent is by no means an easy feat. First of all, we must find a way to transition from the theoretical definition of each need to a more tangible representation in a form that can be understood by an AI agent. This proves to be a problem as we move up Maslow's hierarchy because the needs become more and more abstract. The simplest way to do that, would be to create "Goals" that represent each need and a set of actions for the satisfaction of each goal. For that reason, the Goal Oriented Action Planning system was chosen for this project. As for the simulation's environment, the Unity engine is a very suitable platform as it provides a series of tools and components for making a highly representative simulation.

Rimworld

RimWorld is a sci-fi colony sim driven by an intelligent AI storyteller. Generates stories by simulating psychology, ecology, gunplay, melee combat, climate, biomes, diplomacy, interpersonal relationships, art, medicine, trade, and more. This title deserves to be mentioned as it proved to be a great reference and inspiration for this work. Rimworld was created by a single developer named Tynan Sylvester who founded Ludeon Studios. Furthermore Rimworld was created with the Unity engine.

Figure 9 Rimworld - Rimworld logo

The player has to manage the colonists' moods, needs, individual wounds, and illnesses. Colonists create structures, weapons, and apparel from metal, wood, stone, cloth, or futuristic materials. The player has to watch out for pirate raiders, hostile tribes, rampaging animals, giant tunneling insects and ancient killing machines. It is also possible to tame and train pets, productive farm animals, and deadly attack beasts. Colonists develop relationships with family members, lovers, and spouses. A new world is generated each time a new game is started. The game includes a variety of biomes like the desert, jungle, tundra, and more. Each colonist has his own unique backstories, traits, and skills that directly impact his behavior. In other words, colonists' needs and motivation defer. Colonists also have passion levels for each type of work. The colonist overview screen can be seen in the next figure.



Male colonist of Colony, age 57 (139)

Backstory

Childhood:	Disaster survivor
Adulthood:	Paramilitary agent

Incapable of

(none)

Traits

Chemical interest

Psychically sensitive

Skills

Shooting	11
Melee	10
Social	10
Animals	10
Medicine	6
Cooking	3
Construction	1
Growing	5
Mining	3
Artistic	4
Crafting	5
Research	4

Health

(no injuries)

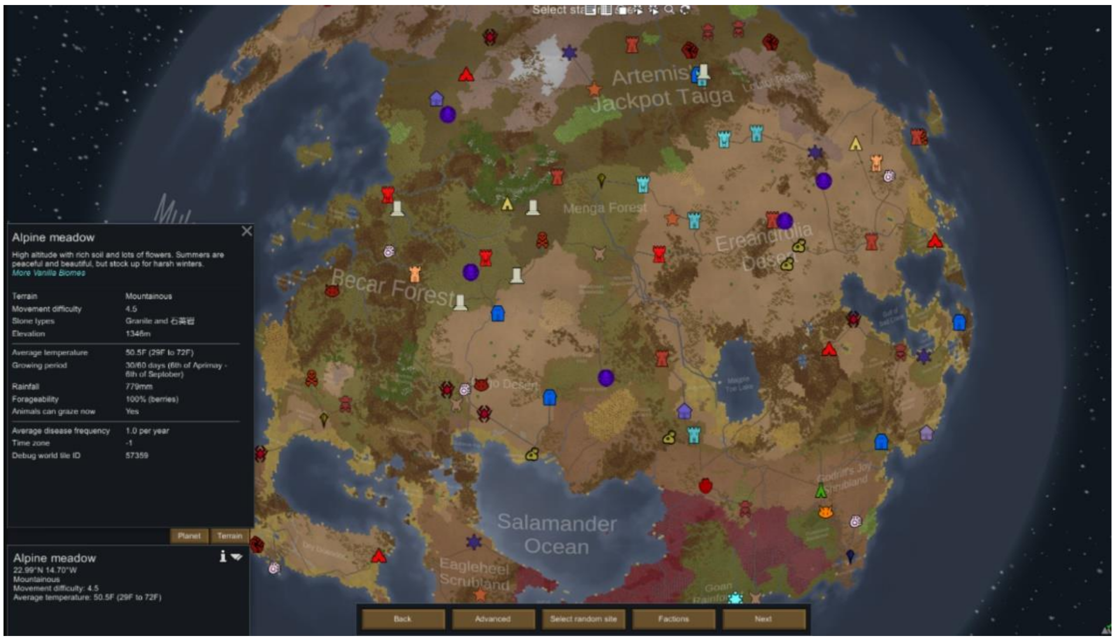
Relations

(no relationships)

Figure 10 Rimworld - Colonist's bio scene

Worlds in RimWorld are generated from either random or user-provided seeds that allow for a completely new experience every time. Each world is uniquely filled with multiple biomes that change what the world looks like and how it behaves. For example, some biomes feature a large number of mountains, while others are full of grasslands.

There are options for configuring the overall planet temperature and overall rainfall, with the result of changing them being true to what would happen in real life. Time zones are also modelled as the world actually physically exists and is fully interactive. Temperatures in regards to proximity from the equator are also correct; with hotter biomes being more central and colder biomes being closer to the poles. The seasonal differences are also less pronounced near the equator, with there being a permanent summer in it.



Alpine meadow

High altitude with rich soil and lots of flowers. Summers are peaceful and beautiful, but stock up for harsh winters.
More Vanilla Biomes

Terrain	Mountainous
Movement difficulty	4.5
Stone types	Granite and 石英岩
Elevation	1346m
Average temperature	50.5F (29F to 72F)
Growing period	30/60 days (6th of Aprimay - 6th of Septober)
Rainfall	779mm
Forageability	100% (berries)
Animals can graze now	Yes
Average disease frequency	1.0 per year
Time zone	-1
Debug world tile ID	57359

Alpine meadow
22.99°N 14.70°W
Mountainous
Movement difficulty: 4.5
Average temperature: 50.5F (29F to 72F)

Figure 11 Rimworld - Random generated world

Project Setup

The programming language used for all the scripts of this simulation is C# and the simulation will be running inside the Unity engine.

First of all, the simulation's requirements must be understood in order to construct the Unity Project. The AI agent:

- must be able to navigate in 3D space,
- must be able to interact with its environment,
- must be animated for a better representation of each performed action,
- must "remember" its own as well as the world's states and react when they are altered,
- must know all available Actions,
- must be aware of its Goals,
- must be able to orchestrate a Plan depending on the current Goal and available Actions
- must have a way to execute Actions in sequence and
- must have a way to determine if it is alive.

Prototyping

To better understand how to translate Maslow's Hierarchy of Needs into code, a prototype was created.

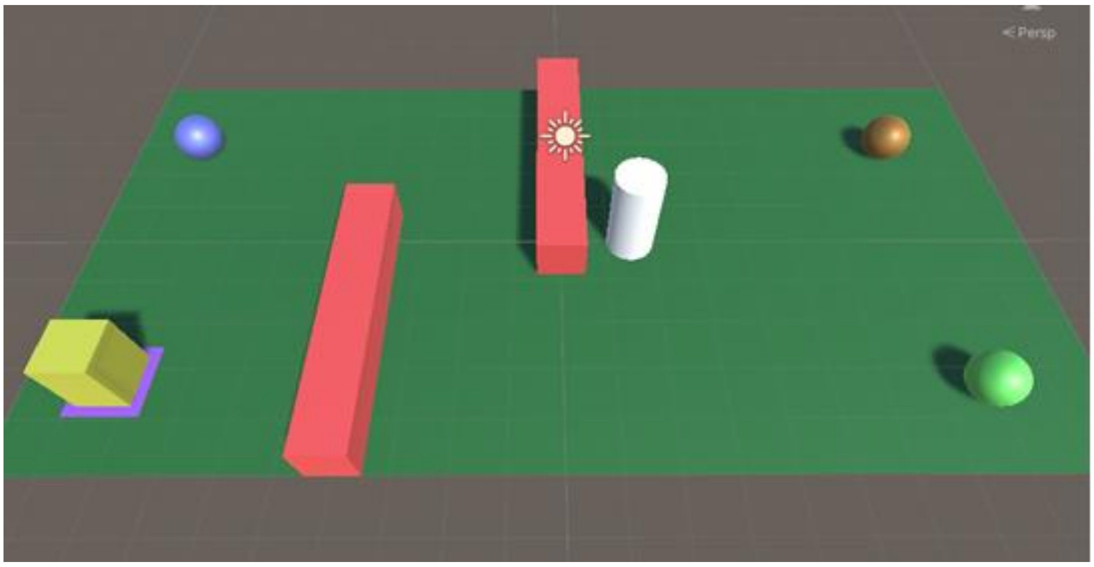
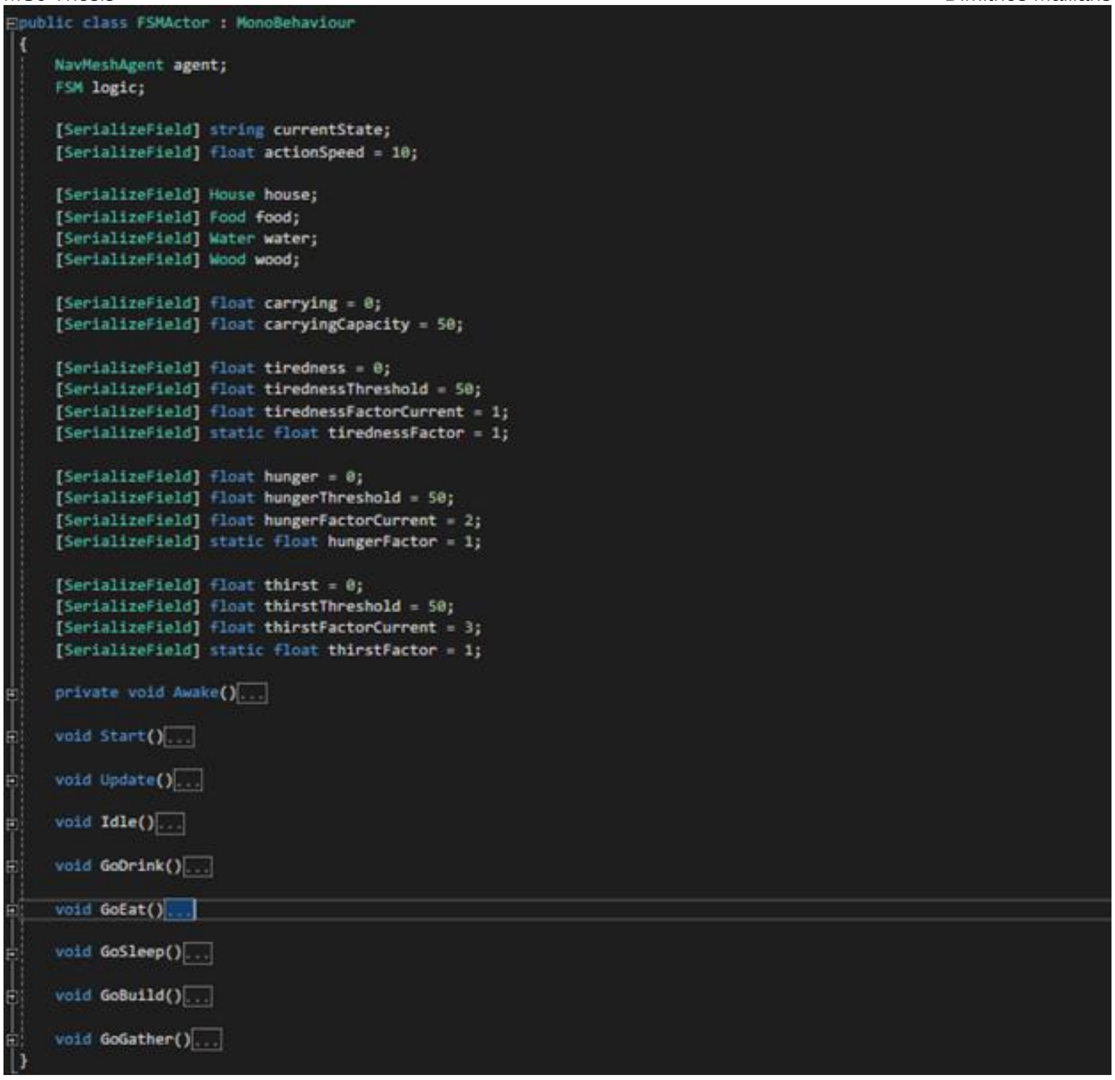


Figure 12 Scene from the prototype that uses Finite State Machine for logic

This prototype used a finite state machine as the core of its logic. The state machine possessed a stack in which needs were pushed as they were triggered when their priority reached a threshold. The Agent then proceeded to execute the action corresponding to the need at the top of the stack. When the action was successfully executed, the need is popped from the stack and the next need comes to the top. If no need is present in the stack, the Agent performs the idle Action which checks all needs ordered by their priority in Maslow's Hierarchy. In this project the Agent is represented by a script called FSMActor.

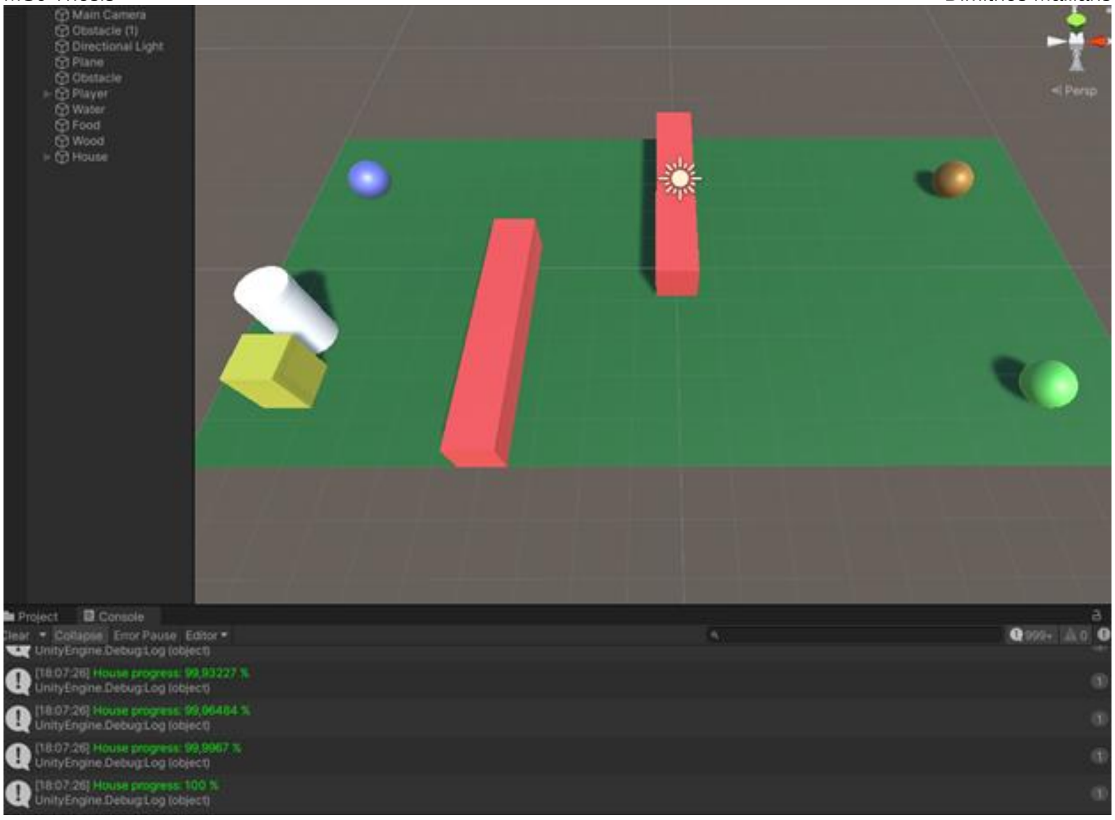


```
public class FSMActor : MonoBehaviour
{
    NavMeshAgent agent;
    FSM logic;

    [SerializeField] string currentState;
    [SerializeField] float actionSpeed = 10;

    [SerializeField] House house;
    [SerializeField] Food food;
    [SerializeField] Water water;
    [SerializeField] Wood wood;

    [SerializeField] float carrying = 0;
    [SerializeField] float carryingCapacity = 50;

    [SerializeField] float tiredness = 0;
    [SerializeField] float tirednessThreshold = 50;
    [SerializeField] float tirednessFactorCurrent = 1;
    [SerializeField] static float tirednessFactor = 1;

    [SerializeField] float hunger = 0;
    [SerializeField] float hungerThreshold = 50;
    [SerializeField] float hungerFactorCurrent = 2;
    [SerializeField] static float hungerFactor = 1;

    [SerializeField] float thirst = 0;
    [SerializeField] float thirstThreshold = 50;
    [SerializeField] float thirstFactorCurrent = 3;
    [SerializeField] static float thirstFactor = 1;

    private void Awake()...

    void Start()...

    void Update()...

    void Idle()...

    void GoDrink()...

    void GoEat()...

    void GoSleep()...

    void GoBuild()...

    void GoGather()...
}
```

Figure 13 The script that controls the prototype's Agent.

Actions here are represented by simple Methods that when executed remove the corresponding need from the stack when the conditions are met.

In this prototype, the idea of the Agent needing to build a house to sleep in, is also explored. When the Agent has no unsatisfied physiological need and there is no available built house, it prioritizes building the house. For the sake of simplicity, it can be assumed that this need could be the product of the safety needs described by Maslow and as such has a lower priority than physiological needs and alas it is examined last by the Agent.



```
UnityEngine.Debug:Log (object)
[18:07:26] House progress: 99,93227 %
UnityEngine.Debug:Log (object)
[18:07:26] House progress: 99,96484 %
UnityEngine.Debug:Log (object)
[18:07:26] House progress: 99,9967 %
UnityEngine.Debug:Log (object)
[18:07:26] House progress: 100 %
UnityEngine.Debug:Log (object)
```

Figure 14 Screenshot of the prototype's scene while running.

When the Agent completes the construction of his house it then prefers to sleep in it when the need to sleep arises.

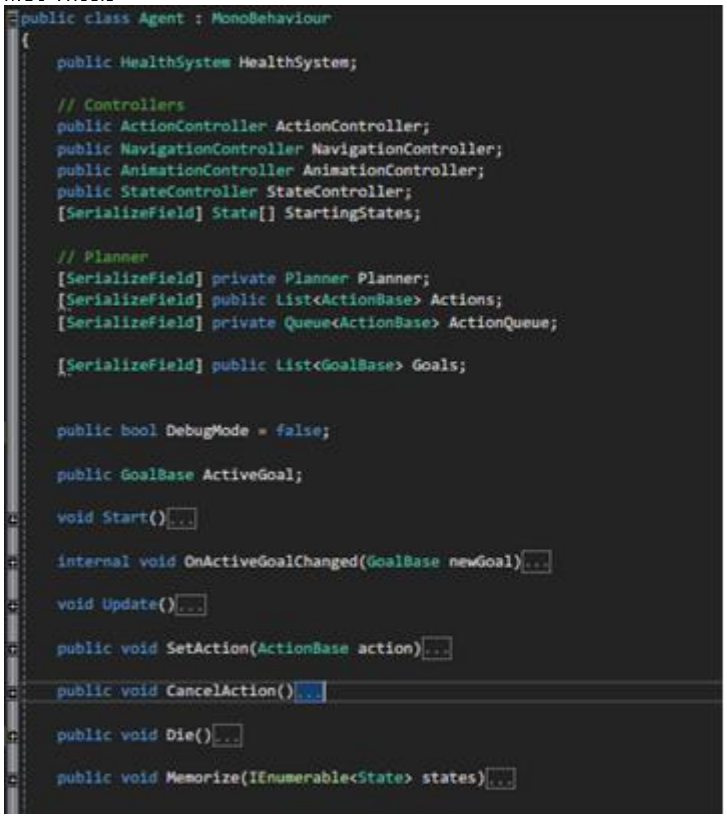
The implementation of an FSM provides a very simple, yet powerful way to represent Maslow's needs in the form of a simulation. A major drawback of this approach is that there is not much room for variation in the way that the Agent acts. Furthermore, the code needed for the representation of more of Maslow's needs will be very repetitive, unmaintainable and overall dense in a sense of readability. The GOAP architecture provides a much more efficient way of managing the Agent's Needs and Goals, in a more dynamic manner.

Creating the scripts - Implementation of GOAP inside Unity

In order to create an AI Agent that qualifies to all previously mentioned requirements, a series of system and controller scripts must be created. Initially, an Action and Goal base class were created as seen below.

Agent - Main Script

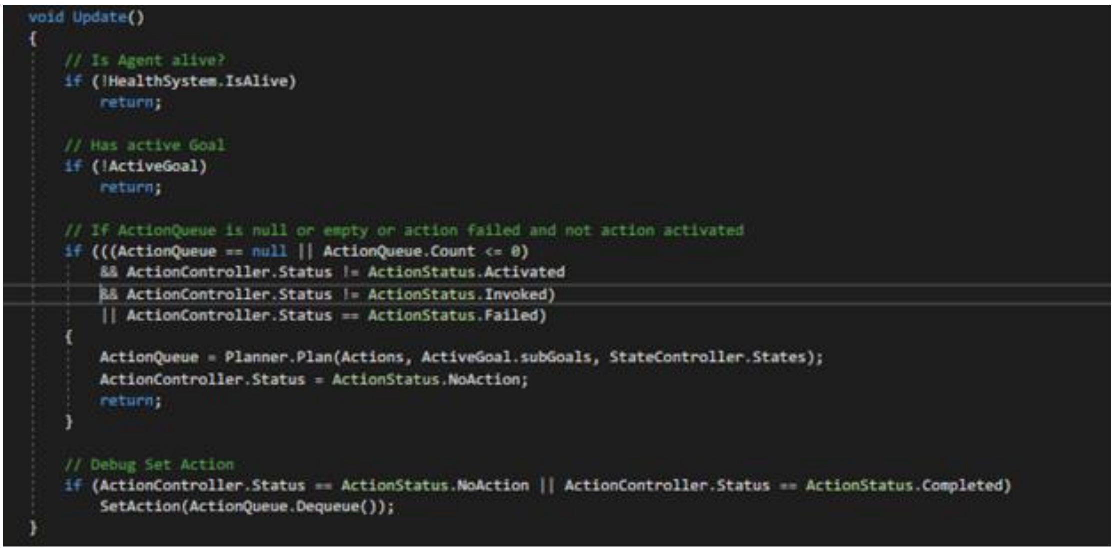
The Agent's main script is the script that stores a lot of information about the Agent. It stores its Goals, available Actions, actionqueue as well as all the controllers' instances. It also handles the Agent's death but more importantly, it provides a way for the Goals to affect the active Action and vice versa. It also facilitates an instance of the Planner script.



```
public class Agent : MonoBehaviour
{
    public HealthSystem HealthSystem;

    // Controllers
    public ActionController ActionController;
    public NavigationController NavigationController;
    public AnimationController AnimationController;
    public StateController StateController;
    [SerializeField] State[] StartingStates;

    // Planner
    [SerializeField] private Planner Planner;
    [SerializeField] public List<ActionBase> Actions;
    [SerializeField] private Queue<ActionBase> ActionQueue;

    [SerializeField] public List<GoalBase> Goals;


    public bool DebugMode = false;

    public GoalBase ActiveGoal;

    void Start()...

    internal void OnActiveGoalChanged(GoalBase newGoal)...

    void Update()...

    public void SetAction(ActionBase action)...

    public void CancelAction()...

    public void Die()...

    public void Memorize(IEnumerable<State> states)...
```

Figure 15 GOAP - Agent class script.

On each update the Agent scripts checks if the current action queue is invalid or completed and must be discarded and planned anew or just be planned if it was not initialized. This way the Agent "thinks" on every frame of the simulation. The update method can be seen below:



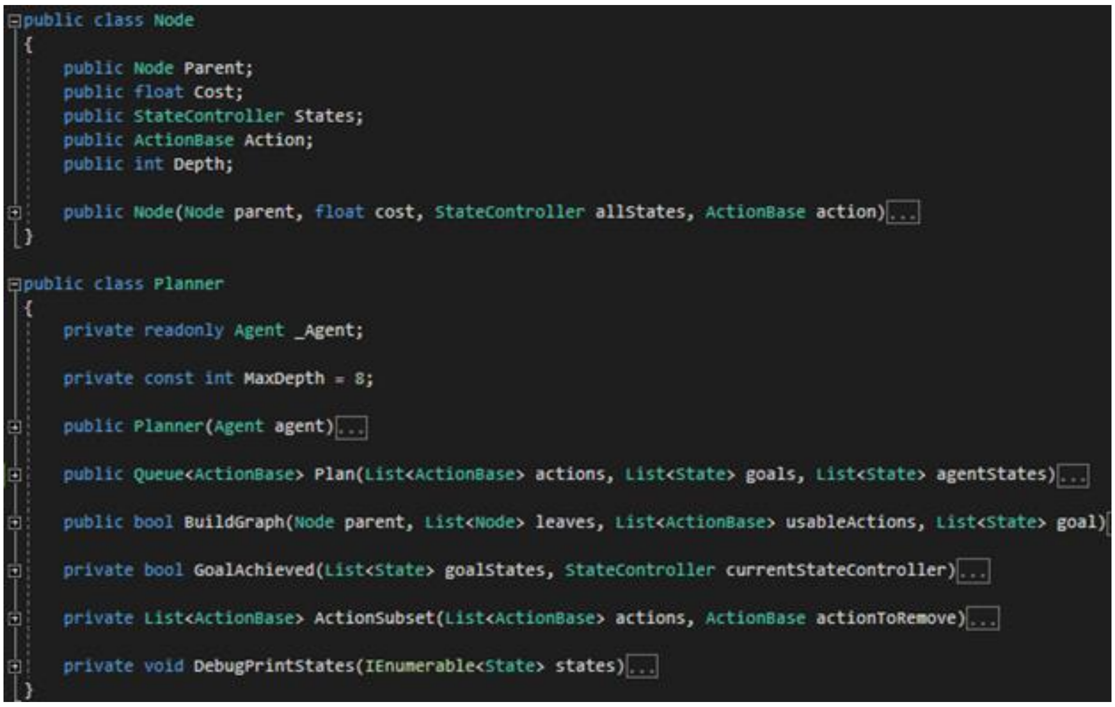
```
void Update()
{
    // Is Agent alive?
    if (!HealthSystem.IsAlive)
        return;

    // Has active Goal
    if (!ActiveGoal)
        return;

    // If ActionQueue is null or empty or action failed and not action activated
    if (((ActionQueue == null || ActionQueue.Count <= 0)
        && ActionController.Status != ActionStatus.Activated
        && ActionController.Status != ActionStatus.Invoked)
        || ActionController.Status == ActionStatus.Failed)
    {
        ActionQueue = Planner.Plan(Actions, ActiveGoal.subGoals, StateController.States);
        ActionController.Status = ActionStatus.NoAction;
        return;
    }

    // Debug Set Action
    if (ActionController.Status == ActionStatus.NoAction || ActionController.Status == ActionStatus.Completed)
        SetAction(ActionQueue.Dequeue());
}
```

Figure 16 GOAP - Agent class script - Update function.

Planner

The Planner class is responsible for the "thinking" part of the AI Agent. By consuming the Agent's current States, available Actions and desired Goal as input, it constructs a series of

Actions which result in a change of States that satisfies the Goal. In simpler terms, the Planner creates a graph consisting of nodes derived from the available Actions and runs a search algorithm against it to find the most effective path.

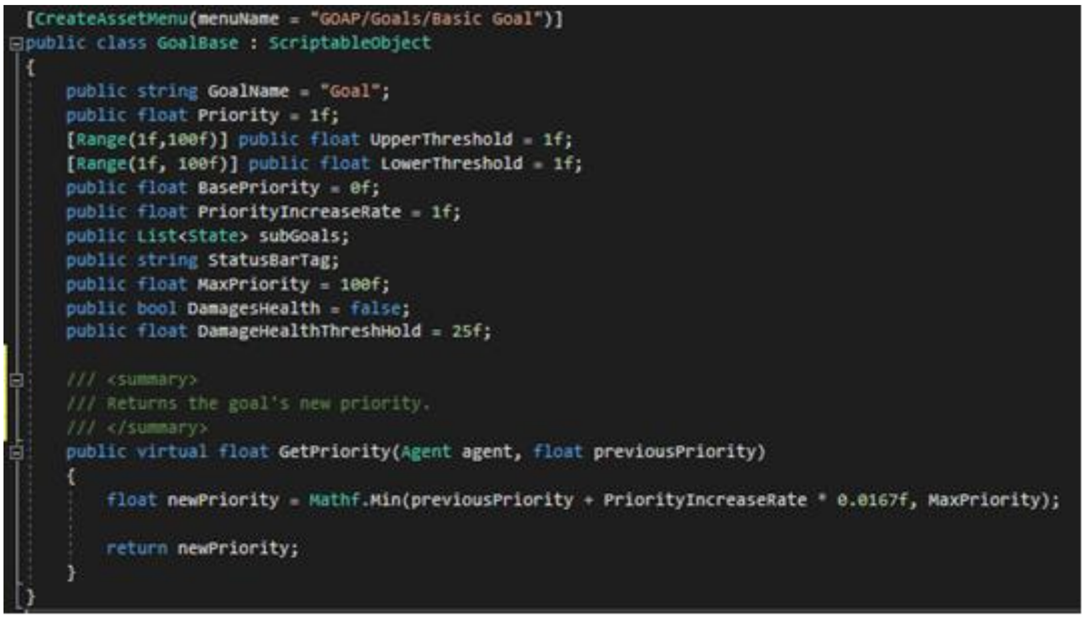


```
public class Node
{
    public Node Parent;
    public float Cost;
    public StateController States;
    public ActionBase Action;
    public int Depth;

    public Node(Node parent, float cost, StateController allStates, ActionBase action)...
}

public class Planner
{
    private readonly Agent _Agent;

    private const int MaxDepth = 8;

    public Planner(Agent agent)...

    public Queue<ActionBase> Plan(List<ActionBase> actions, List<State> goals, List<State> agentStates)...

    public bool BuildGraph(Node parent, List<Node> leaves, List<ActionBase> usableActions, List<State> goal)

    private bool GoalAchieved(List<State> goalStates, StateController currentStateController)...

    private List<ActionBase> ActionSubset(List<ActionBase> actions, ActionBase actionToRemove)...

    private void DebugPrintStates(IEnumerable<State> states)...
}
```

Figure 17 GOAP - Planner class script.

Goals

Goal classes inherit from the same base class named GoalBase. The GoalBase class contains all basic information for each goal such as its priority, thresholds and how it increases/decreases over time and also whether the deficiency of the Goal should damage the agent’s health. An overview of the class structure can be seen in the figure below:



```
 [CreateAssetMenu(menuName = "GOAP/Goals/Basic Goal")]
public class GoalBase : ScriptableObject
{
    public string GoalName = "Goal";
    public float Priority = 1f;
    [Range(1f,100f)] public float UpperThreshold = 1f;
    [Range(1f, 100f)] public float LowerThreshold = 1f;
    public float BasePriority = 0f;
    public float PriorityIncreaseRate = 1f;
    public List<State> subGoals;
    public string StatusBarTag;
    public float MaxPriority = 100f;
    public bool DamagesHealth = false;
    public float DamageHealthThreshHold = 25f;

    /// <summary>
    /// Returns the goal's new priority.
    /// </summary>
    public virtual float GetPriority(Agent agent, float previousPriority)
    {
        float newPriority = Mathf.Min(previousPriority + PriorityIncreaseRate * 0.0167f, MaxPriority);

        return newPriority;
    }
}
```

Figure 18 GOAP - Goal class script.

Goal State

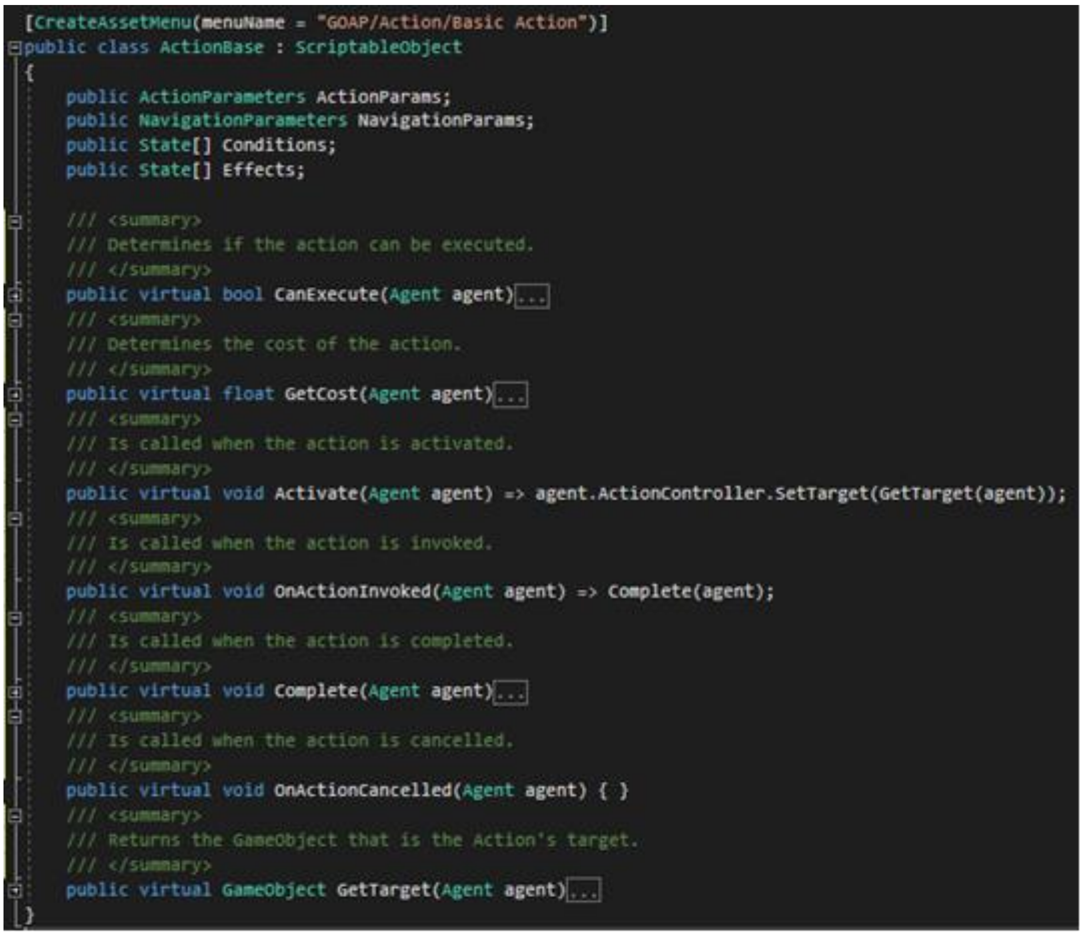
The States that satisfy each goal differ and may be more than one. For that reason these States are stored in a list as sub goals.

Priority

Each Goal's priority is calculated based on the previous priority's value. Also, it cannot exceed the MaxPriority variable set to each Goal.

Actions

All Actions inherit from the same class, the ActionBase. This class contains all the information (parameters & instruction) for the execution of an Action. The ActionBase class can be seen in the figure below along with its Methods and their summary:



```
[CreateAssetMenu(menuName = "GOAP/Action/Basic Action")]
public class ActionBase : ScriptableObject
{
    public ActionParameters ActionParams;
    public NavigationParameters NavigationParams;
    public State[] Conditions;
    public State[] Effects;

    /// <summary>
    /// Determines if the action can be executed.
    /// </summary>
    public virtual bool CanExecute(Agent agent)...
    /// <summary>
    /// Determines the cost of the action.
    /// </summary>
    public virtual float GetCost(Agent agent)...
    /// <summary>
    /// Is called when the action is activated.
    /// </summary>
    public virtual void Activate(Agent agent) => agent.ActionController.SetTarget(GetTarget(agent));
    /// <summary>
    /// Is called when the action is invoked.
    /// </summary>
    public virtual void OnActionInvoked(Agent agent) => Complete(agent);
    /// <summary>
    /// Is called when the action is completed.
    /// </summary>
    public virtual void Complete(Agent agent)...
    /// <summary>
    /// Is called when the action is cancelled.
    /// </summary>
    public virtual void OnActionCancelled(Agent agent) { }
    /// <summary>
    /// Returns the GameObject that is the Action's target.
    /// </summary>
    public virtual GameObject GetTarget(Agent agent)...
}
```

Figure 19 GOAP - Action class script.

Conditions & Effects

The preconditions and effects of each action are stored inside arrays as States. These arrays are kept public in order for the Planner to have access to that info.

Parameters

The ActionParameters class contains information about the Action, for example, its cost, duration, animation, action range and information about the target of the action. Similarly, the

NavigationParameters class contains parameters used for the Agent’s Navigation such as navigation speed, navigation cost etc.

Cost

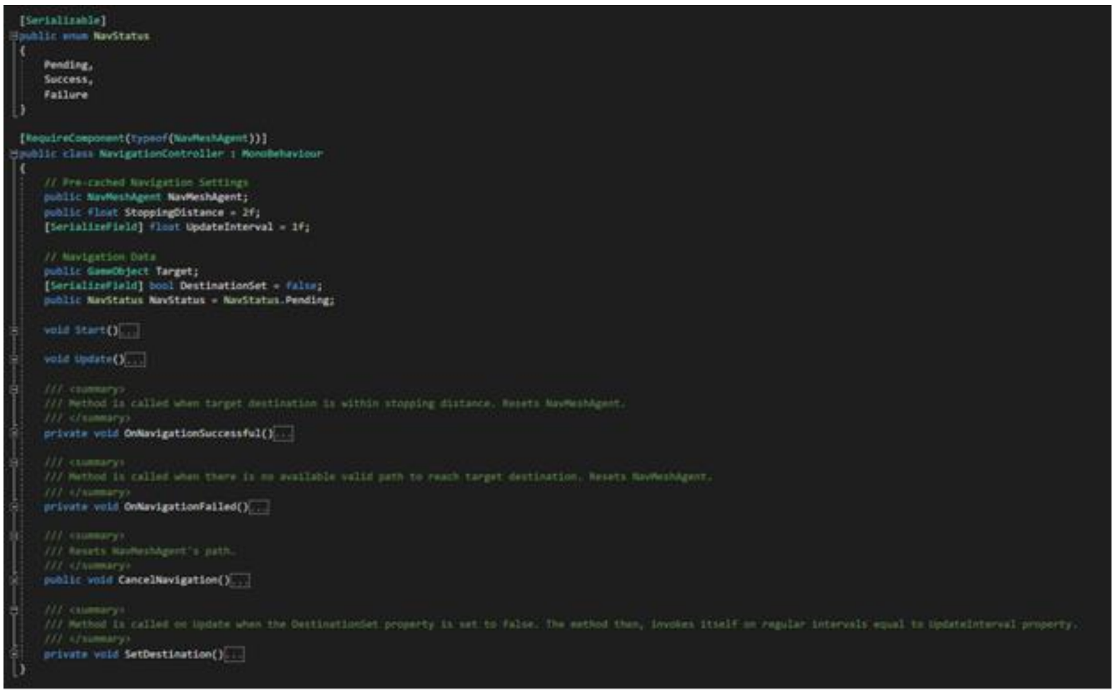
The cost is calculated using the GetCost() method. This method returns the sum of the base cost when added to the navigational cost which is calculated taking into account the distance from the Agent's position to the target GameObject. Obviously, the navigational cost amounts to zero if the target is the Agent itself or there is no target for the Action.

Controllers

Almost all changes that occur to the Agent’s GameObject are handled by a group of controller scripts. These scripts inherit from the MonoBehaviour, allowing them to be attached to the Agent. These controllers’ operation is tied to the Agent’s main script from which they access information about the Agent’s state. It is important to state that some of these controllers rely on one another for their operation. For example, the action controller needs the navigation controller to know when the Agent has reached its destination.

Navigation Controller

The navigation controller is responsible for handling the Agent’s navigation in the 3D space. This script communicates with Unity’s NavMeshAgent and in order to steer and guide the Agent’s GameObject. The controller uses an enum to describe its state. If the navigation is in a pending, successful or failure state.



```
[Serializable]
public enum NavStatus
{
    Pending,
    Success,
    Failure
}

[RequireComponent(typeof(NavMeshAgent))]
public class NavigationController : MonoBehaviour
{
    // Pre-cached Navigation Settings
    public NavMeshAgent NavMeshAgent;
    public float StoppingDistance = 2f;
    [SerializeField] float UpdateInterval = 1f;

    // Navigation Data
    public GameObject Target;
    [SerializeField] bool DestinationSet = false;
    public NavStatus NavStatus = NavStatus.Pending;

    void Start()[...]

    void Update()[...]

    /// <summary>
    /// Method is called when target destination is within stopping distance. Resets NavMeshAgent.
    /// </summary>
    private void OnNavigationSuccessful()[...]

    /// <summary>
    /// Method is called when there is no available valid path to reach target destination. Resets NavMeshAgent.
    /// </summary>
    private void OnNavigationFailed()[...]

    /// <summary>
    /// Resets NavMeshAgent's path.
    /// </summary>
    public void CancelNavigation()[...]

    /// <summary>
    /// Method is called on Update when the DestinationSet property is set to False. The method then, invokes itself on regular intervals equal to UpdateInterval property.
    /// </summary>
    private void SetDestination()[...]
}
```

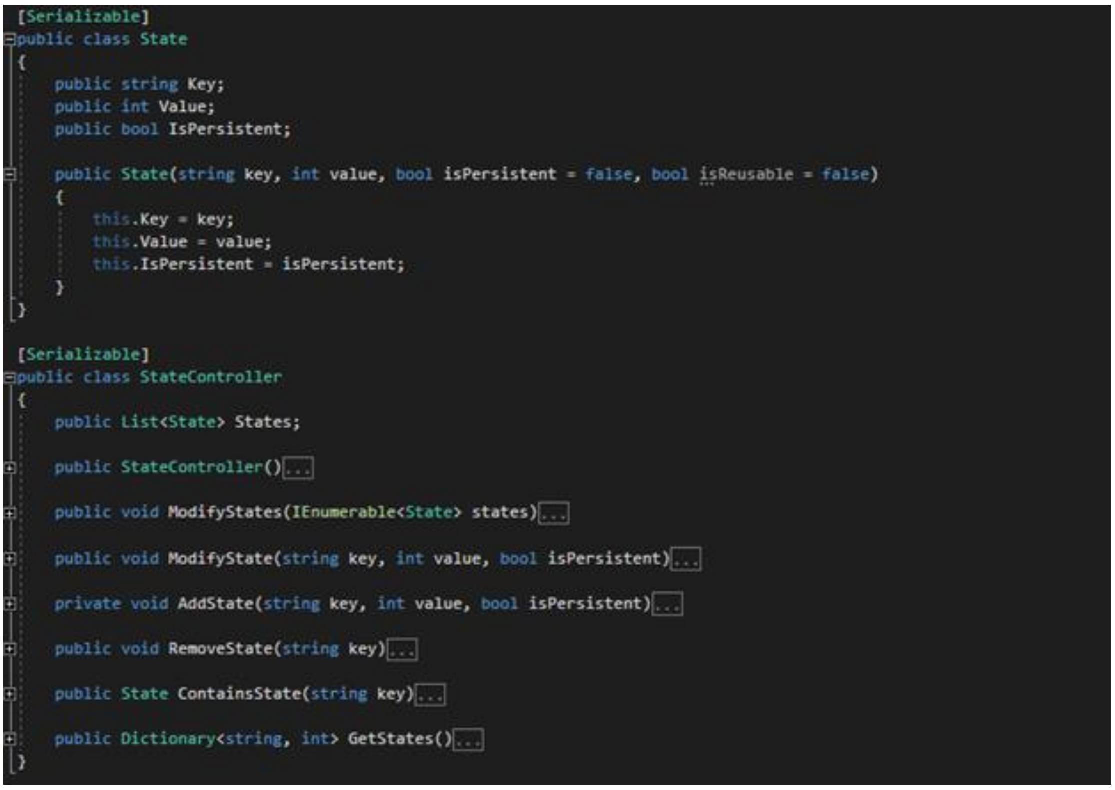
Figure 20 GOAP - Navigation controller script

Animation Controller

The animation controller is the script that manages the animation that is currently “played” by the Agent’s model. It is instructed by the action controller when, for how long and what animation it should be “playing”.

State Controller

The state controller is a helper script that is used to handle the Agent's current States. In the next figure the definition of the State class can also be seen as it is closely tied to this controller. This controller is able to store new States and modify or remove existing ones.

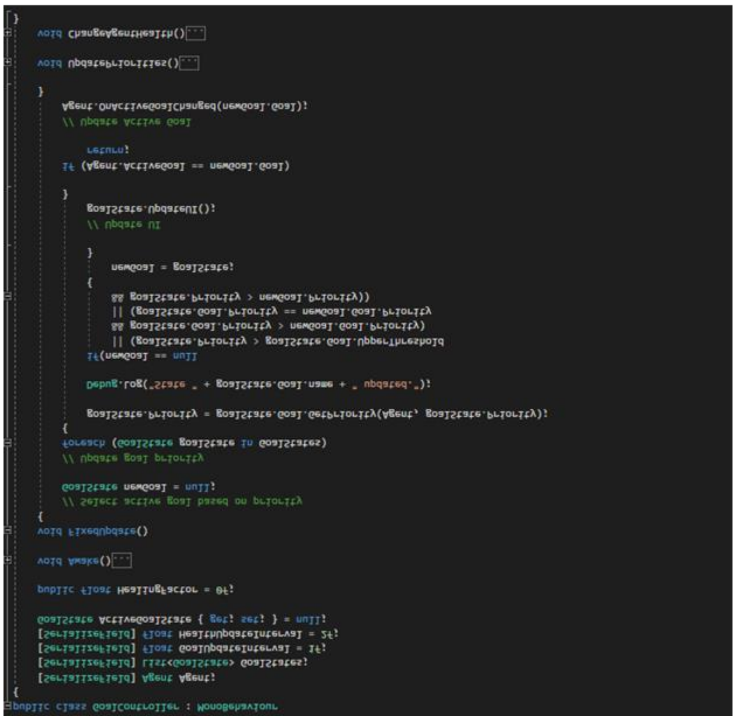


```
[Serializable]
public class State
{
    public string Key;
    public int Value;
    public bool IsPersistent;

    public State(string key, int value, bool isPersistent = false, bool isReusable = false)
    {
        this.Key = key;
        this.Value = value;
        this.IsPersistent = isPersistent;
    }
}

[Serializable]
public class StateController
{
    public List<State> States;

    public StateController()...

    public void ModifyStates(IEnumerable<State> states)...

    public void ModifyState(string key, int value, bool isPersistent)...

    private void AddState(string key, int value, bool isPersistent)...

    public void RemoveState(string key)...

    public State ContainsState(string key)...

    public Dictionary<string, int> GetStates()...
}
```

Figure 21 GOAP - State controller script.

Goal Controller

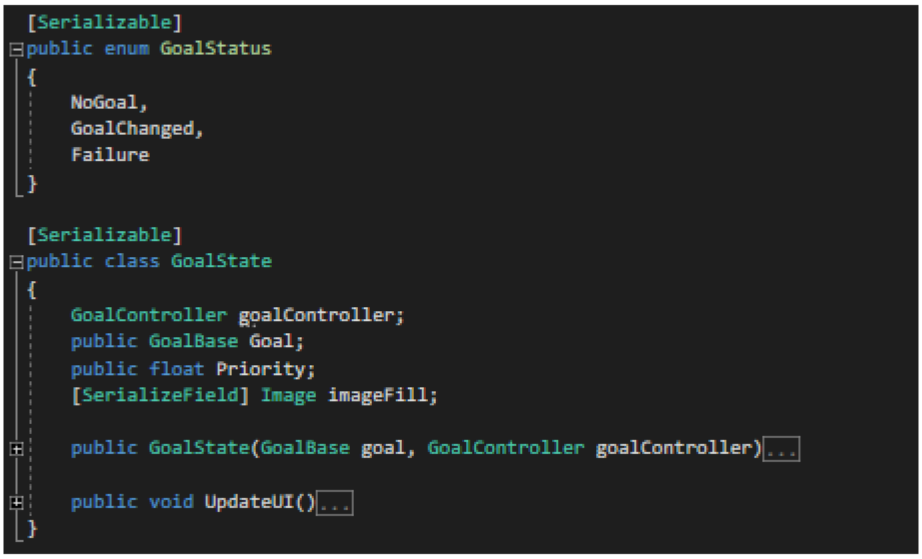
The goal controller is the script that is responsible for handling the Agent's Goals. Every fixed update, it updates each Goal's priority and decides if a new Goal should be activated instead of the one currently active. Additionally, it refreshes the UI to represent the altered data, as well as interacts with the Agent's health based on the Goals' settings.



```
public class GoalController : MonoBehaviour
{
    [SerializeField] Agent Agent;
    [SerializeField] List<GoalState> GoalStates;
    [SerializeField] float GoalUpdateInterval = 1f;
    [SerializeField] float HealthUpdateInterval = 2f;
    GoalState ActiveGoalState { get; set; } = null;

    public float HealingFactor = 9f;

    void Awake()[...]

    void FixedUpdate()
    {
        // Select active goal based on priority
        GoalState newGoal = null;

        // Update goal priority
        foreach (GoalState goalState in GoalStates)
        {
            goalState.Priority = goalState.Goal.GetPriority(Agent, goalState.Priority);

            Debug.Log("State " + goalState.Goal.name + " updated.");

            if(newGoal == null
                || (goalState.Priority > goalState.Goal.UpperThreshold
                && goalState.Goal.Priority > newGoal.Goal.Priority)
                || (goalState.Goal.Priority == newGoal.Goal.Priority
                && goalState.Priority > newGoal.Priority))
            {
                newGoal = goalState;
            }

            // Update UI
            goalState.UpdateUI();
        }

        if (Agent.ActiveGoal == newGoal.Goal)
            return;

        // Update active goal
        Agent.OnActiveGoalChanged(newGoal.Goal);
    }

    void UpdatePriorities()[...]

    void ChangeAgentHealth()[...]
}
```

Figure 22 GOAP - Goal controller script.

This controller also utilizes one helper class called GoalState which describes each Goal's state and helps transfer that info to the UI. Furthermore, the controller's status is described to other components by the use of an enum called GoalStatus. These can be seen in the next figure.

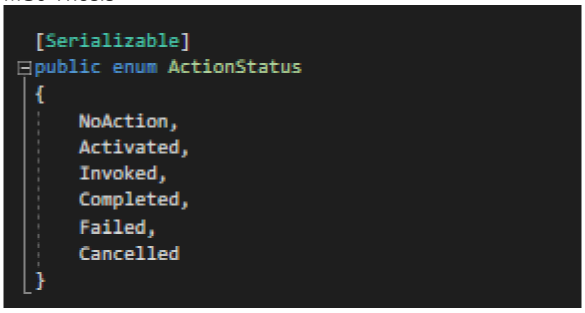


```
[Serializable]
public enum GoalStatus
{
    NoGoal,
    GoalChanged,
    Failure
}

[Serializable]
public class GoalState
{
    GoalController goalController;
    public GoalBase Goal;
    public float Priority;
    [SerializeField] Image imageFill;

    public GoalState(GoalBase goal, GoalController goalController)[...]

    public void UpdateUI()[...]
}
```

Figure 23 GOAP - Goal controller script 2.

Action Controller

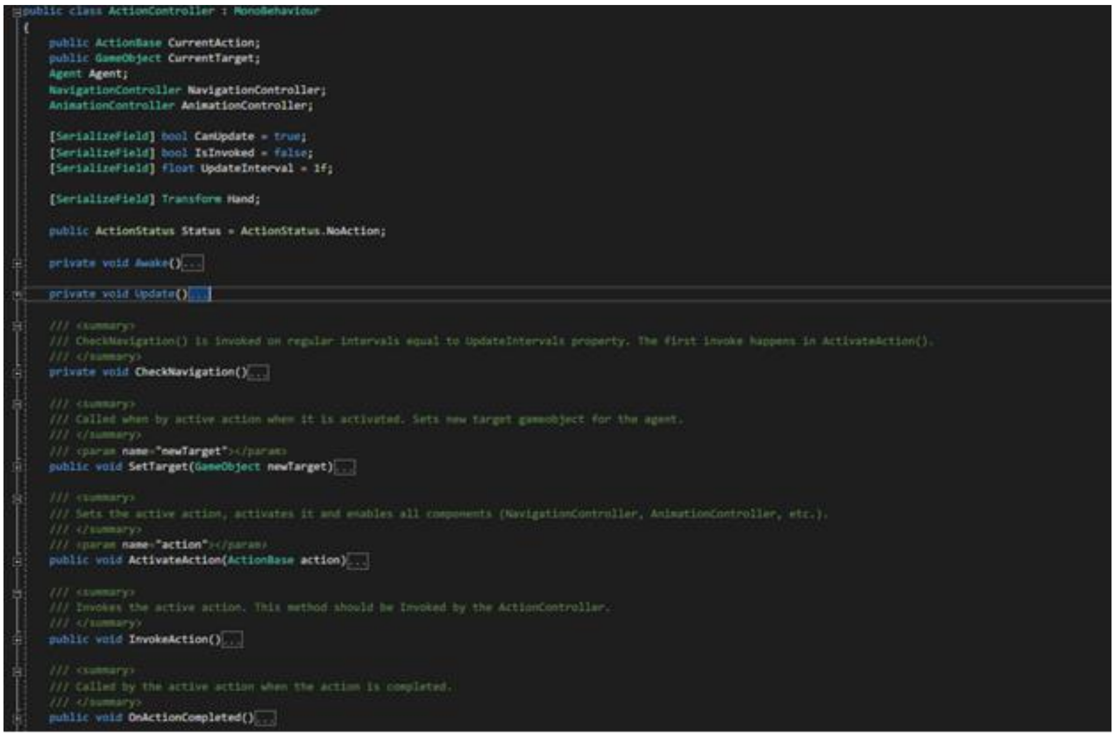
The action controller is the controller responsible for the execution of an Agent's Action. It describes the state of an Action by the use of an enum called ActionStatus as seen below:



```
[Serializable]
public enum ActionStatus
{
    NoAction,
    Activated,
    Invoked,
    Completed,
    Failed,
    Cancelled
}
```

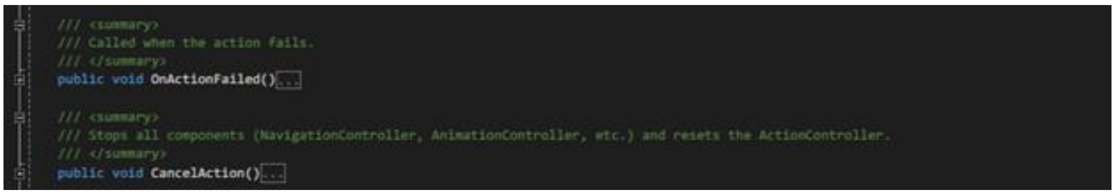
Figure 24 GOAP - Action controller script.

This controller reads the currently active Action and acts according to its parameters. It sets the Agent's target, the Agent's destination through the navigation controller, activates an animation sequence through the animation controller and lastly calls the Action's methods.



```
public class ActionController : MonoBehaviour
{
    public ActionBase CurrentAction;
    public GameObject CurrentTarget;
    Agent Agent;
    NavigationController NavigationController;
    AnimationController AnimationController;

    [SerializeField] bool CanUpdate = true;
    [SerializeField] bool IsInvoked = false;
    [SerializeField] float UpdateInterval = 1f;

    [SerializeField] Transform Hand;

    public ActionStatus Status = ActionStatus.NoAction;

    private void Awake()[...]

    private void Update()[...]

    /// <summary>
    /// CheckNavigation() is invoked on regular intervals equal to UpdateIntervals property. The first invoke happens in ActivateAction().
    /// </summary>
    private void CheckNavigation()[...]

    /// <summary>
    /// Called when by active action when it is activated. Sets new target gameobject for the agent.
    /// </summary>
    /// <param name="newTarget"></param>
    public void SetTarget(GameObject newTarget)[...]

    /// <summary>
    /// Sets the active action, activates it and enables all components (NavigationController, AnimationController, etc.).
    /// </summary>
    /// <param name="action"></param>
    public void ActivateAction(ActionBase action)[...]

    /// <summary>
    /// Invokes the active action. This method should be Invoked by the ActionController.
    /// </summary>
    public void InvokeAction()[...]

    /// <summary>
    /// Called by the active action when the action is completed.
    /// </summary>
    public void OnActionCompleted()[...]
```

Figure 25 GOAP - Action controller script 2.



```
    /// <summary>
    /// Called when the action fails.
    /// </summary>
    public void OnActionFailed()[...]

    /// <summary>
    /// Stops all components (NavigationController, AnimationController, etc.) and resets the ActionController.
    /// </summary>
    public void CancelAction()[...]
```

Figure 26 GOAP - Action controller script 3.

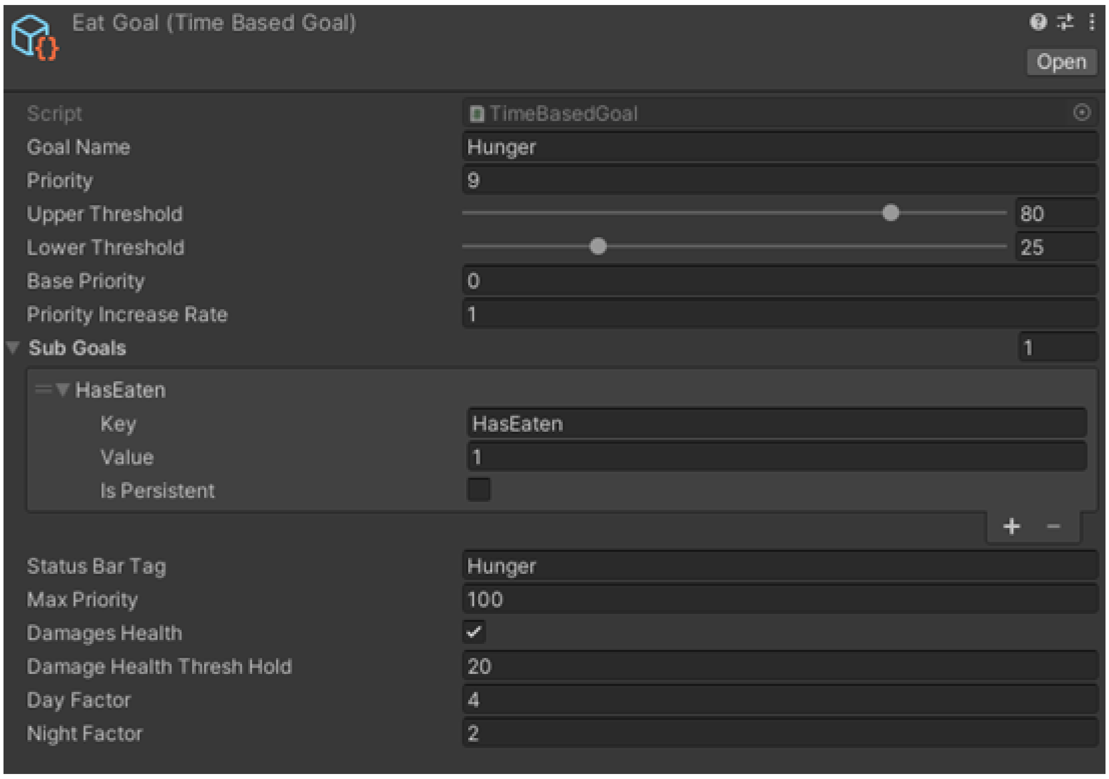
By the above figures we can understand the importance and the complexity of this controller. It is the script that ensures that an Action is handled properly and without errors.

Implementing Physiological Human Needs

For this demonstration, from Maslow's hierarchy of needs, only some of the physiological needs were chosen to be recreated. This does not mean that it is impossible to convert all of Maslow's needs to Goal scripts but rather that it is not required as this project is just an example used to prove that it can be done.

From Needs to Goals

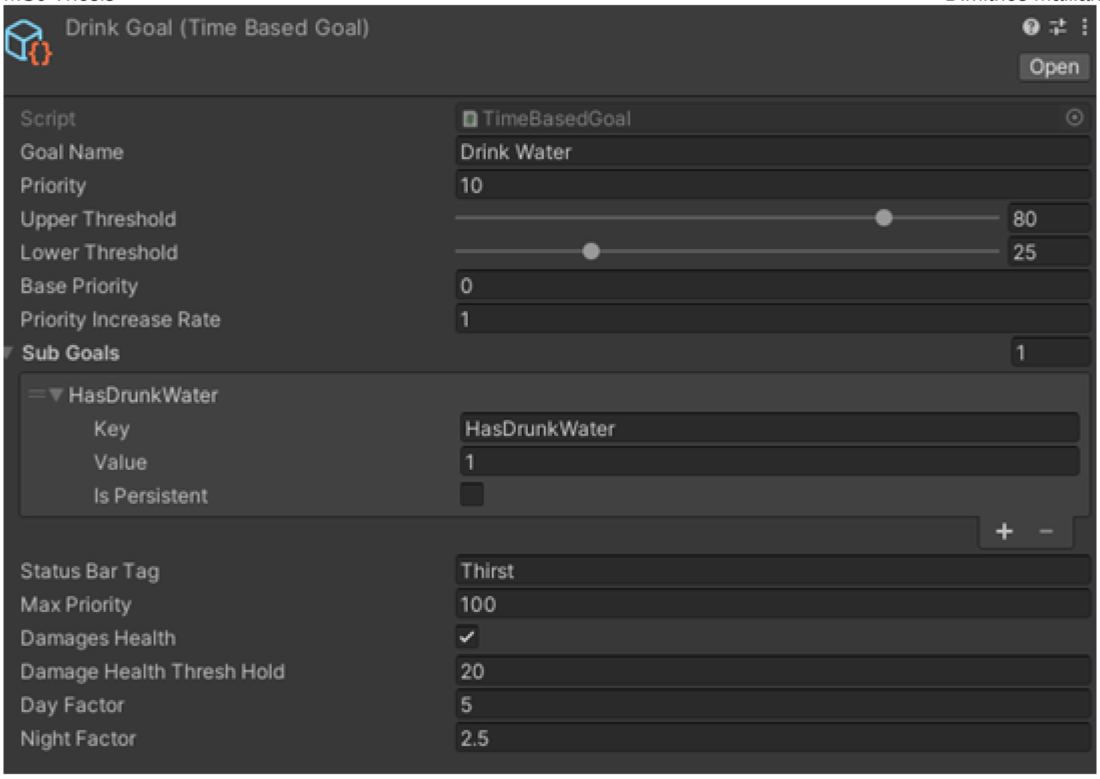
The needs demonstrated in this example are thirst, hunger, sleep and recreation. The need for recreation is just used to describe the general mental health of the Agent just for the sake of this example. The needs are shown in the figures below, represented as Goal objects. The class used to create these is TimeBasedGoal which is an extension to the GoalBase that implements a time related priority increase.



Eat Goal (Time Based Goal)

Script	TimeBasedGoal
Goal Name	Hunger
Priority	9
Upper Threshold	80
Lower Threshold	25
Base Priority	0
Priority Increase Rate	1
Sub Goals	1
HasEaten	
Key	HasEaten
Value	1
Is Persistent	☐
Status Bar Tag	Hunger
Max Priority	100
Damages Health	✓
Damage Health Thresh Hold	20
Day Factor	4
Night Factor	2

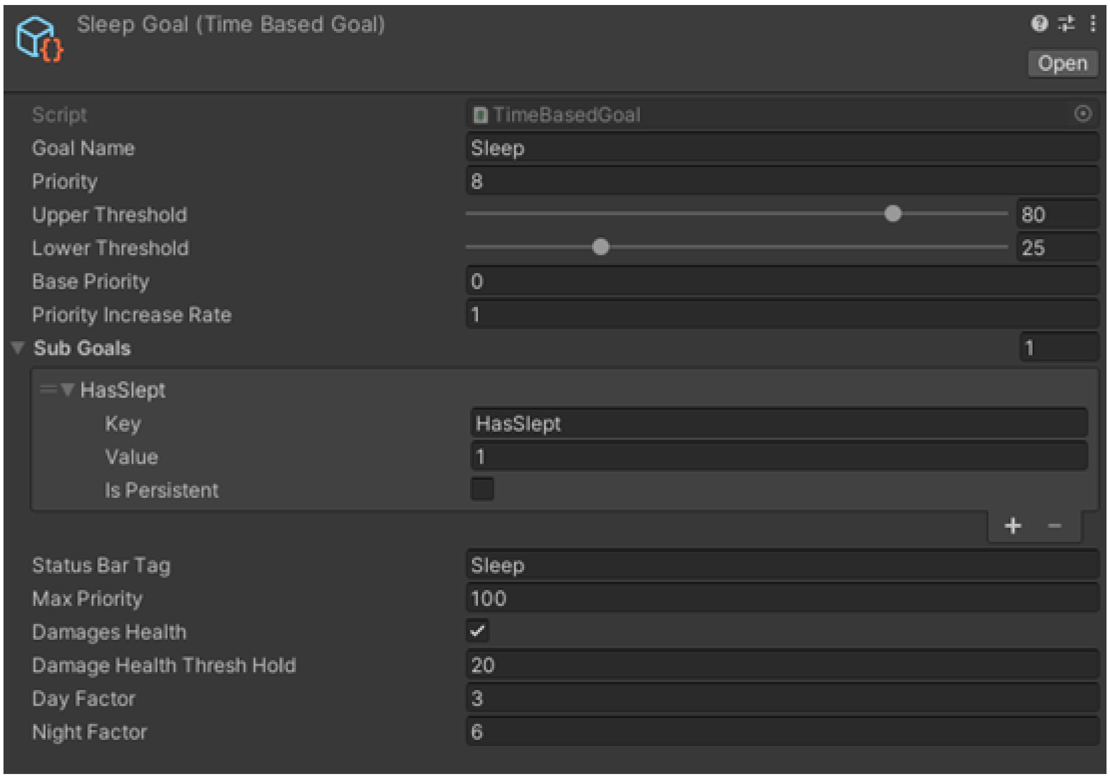
Figure 27 GOAP - Hunger - Goals' ScriptableObject


Drink Goal (Time Based Goal)

Open

Script	TimeBasedGoal
Goal Name	Drink Water
Priority	10
Upper Threshold	80
Lower Threshold	25
Base Priority	0
Priority Increase Rate	1
Sub Goals	1
HasDrunkWater	
Key	HasDrunkWater
Value	1
Is Persistent	
Status Bar Tag	Thirst
Max Priority	100
Damages Health	✓
Damage Health Thresh Hold	20
Day Factor	5
Night Factor	2.5

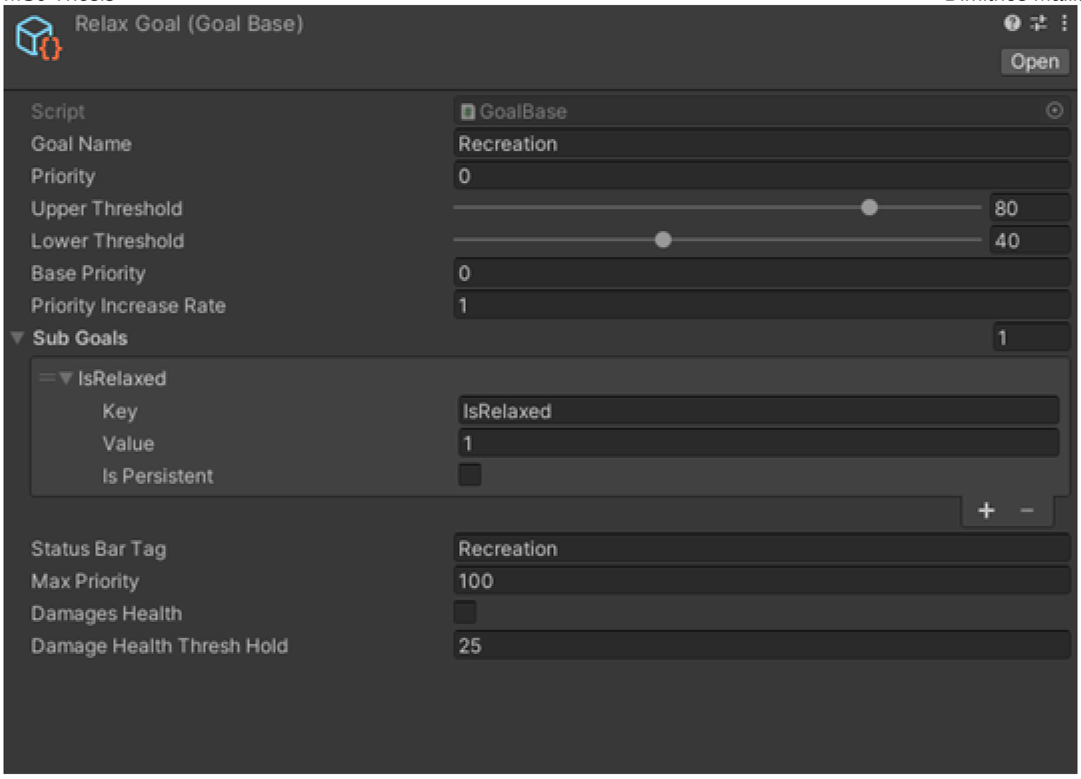
Figure 28 GOAP - Thirst - Goals' ScriptableObject


Sleep Goal (Time Based Goal)

Open

Script	TimeBasedGoal
Goal Name	Sleep
Priority	8
Upper Threshold	80
Lower Threshold	25
Base Priority	0
Priority Increase Rate	1
Sub Goals	1
HasSlept	
Key	HasSlept
Value	1
Is Persistent	
Status Bar Tag	Sleep
Max Priority	100
Damages Health	✓
Damage Health Thresh Hold	20
Day Factor	3
Night Factor	6

Figure 29 GOAP - Sleep - Goals' ScriptableObject



Relax Goal (Goal Base)

Script	GoalBase
Goal Name	Recreation
Priority	0
Upper Threshold	80
Lower Threshold	40
Base Priority	0
Priority Increase Rate	1
Sub Goals	1
IsRelaxed	
Key	IsRelaxed
Value	1
Is Persistent	☐
Status Bar Tag	Recreation
Max Priority	100
Damages Health	☐
Damage Health Thresh Hold	25

Figure 30 GOAP - Recreation - Goals’ ScriptableObject

Running the Simulation

In order to run the simulation, a scene was created. It was then filled with GameObjects like trees, rocks, a patch of water, a house, a campfire and some bushes with fruits. The white figure that the camera focuses on is the Agent's GameObject. In the UI we can examine the Agent's needs, its current action and its current health factor which is a factor whose value determines if the Agent should lose or gain health.

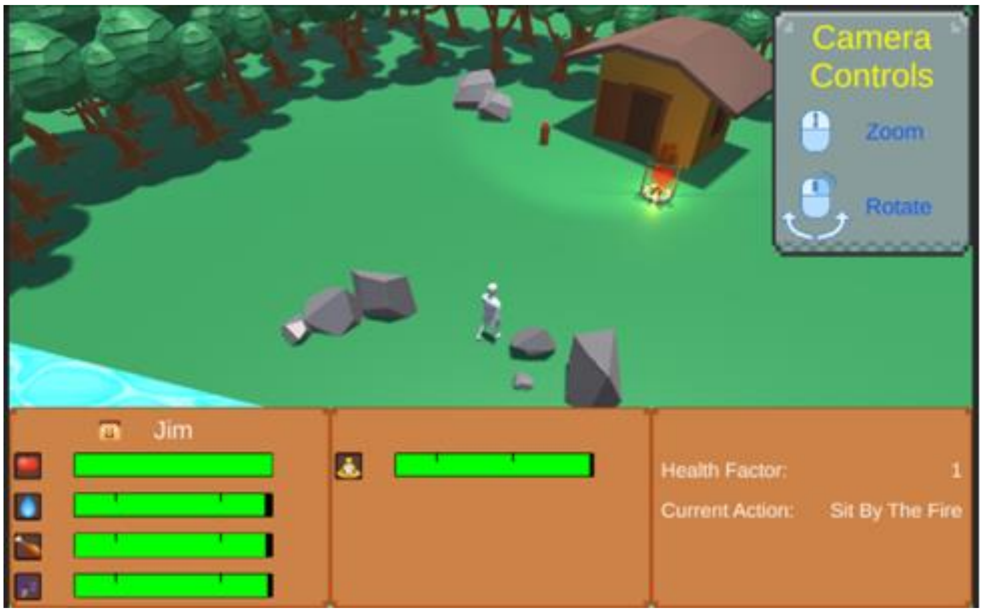


Figure 31 GOAP - Simulation at runtime example.

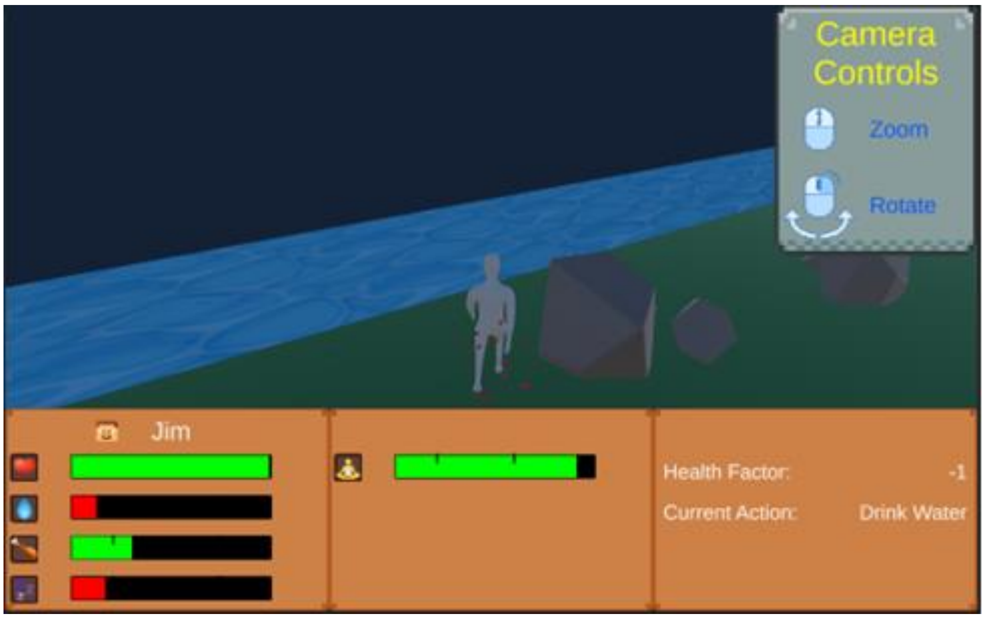
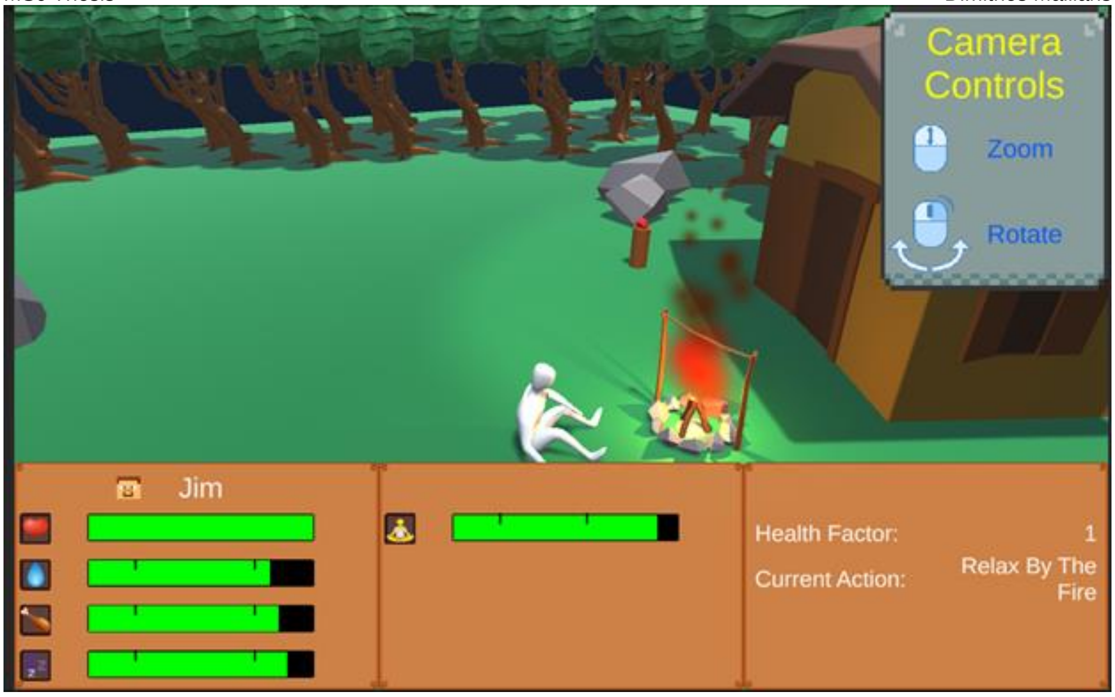


Figure 32 GOAP - Simulation at runtime example 2.

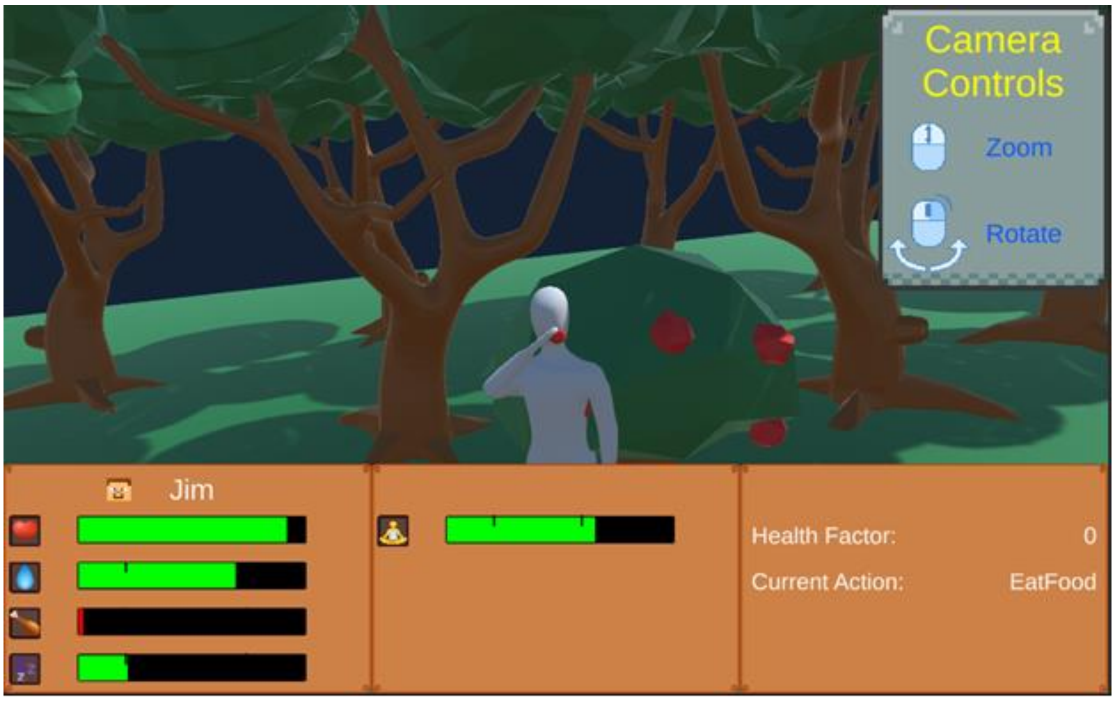
When there is a deficiency in more than two physiological needs the Agent takes damage until his health reaches zero and then perishes. The bars representing the Agent's needs display a lower and upper threshold. The upper threshold serves no practical purpose in this example. The lower threshold represents an Agent's need's critical level and when reached, the Agent will prioritize doing the corresponding actions to satisfy the need. In other words, the Agent will set the satisfaction of that need as his active Goal.



Health Factor: 1
Current Action: Relax By The Fire

Figure 33 GOAP - Simulation at runtime example 3.

When there is no other need that requires the Agent’s attention, the Agent is set to by default “relax” by the campfire. This can be replaced by any action and especially if all of Maslow’s needs are implemented, the Agent will most likely be busy most of the time doing something.



Health Factor: 0
Current Action: EatFood

Figure 34 GOAP - Simulation at runtime example 4.

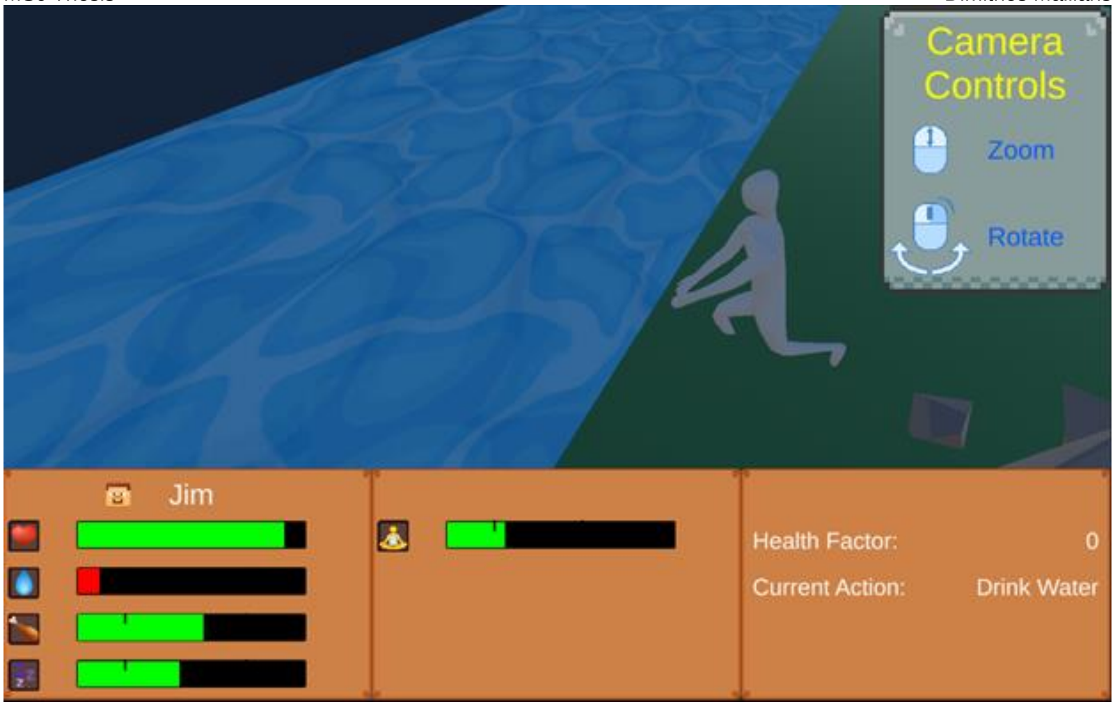


Figure 35 GOAP - Simulation at runtime example 5.

Conclusion & Future Work

Through the course of this project, there are a plethora of conclusions that become apparent. To begin with, Goal Oriented Action Planning is a very effective tool when creating dynamic AI Agents. There are multiple possibilities as to how this project could be further expanded. For example, the idea of implementing more needs from Maslow's hierarchy, along with a much wider variety of available Actions for the Agent to choose from, should be explored as it could provide better human-like behavior and some interesting conclusions might emerge. Also, an emotion system that would simulate human feelings and consequently affect the Agents' behavior could be implemented. Furthermore, the Agents could be equipped with an awareness system that would allow them to simulate the human senses. Lastly, a major addition would be to have more than one Agents in the same simulation, that would interact with one another in many possible ways (e.g. social, combat). In summary, this work's goal is to provide a base for later projects to build upon and expand, as there is a lot of work involved into creating it and could prove useful for later endeavors on the subject.

A Word from the Author

As a personal experience, in the process of developing this project I was able to acquire valuable knowledge concerning the C# programming language, Unity engine, the process of creating an intelligent Agent and Artificial Intelligence in general. I believe that this experience will prove most valuable in my later career in the field of Information Technology and I hope that my work will be able to inspire and help other students as they progress their studies. Thank you for taking the time to read my work.

Bibliography & References

1. Abd Algfoor, Zeyad; Sunar, Mohd Shahrizal; Kolivand, Hoshang (2015). "A Comprehensive Study on Pathfinding Techniques for Robotics and Video Games". 2015
2. AlphaStar: Mastering the Real-Time Strategy Game StarCraft II. DeepMind. Retrieved 2019-06-04
3. Applying Goal-Oriented Action Planning to Games, Jeff Orkin – Monolith Productions, http://www.jorkin.com
4. Artificial Intelligence A Modern Approach Second Edition Stuart J. Russell and Peter Norvig, 2003
5. Brügmann, Bernd (1993). Monte Carlo Go
6. Finite State Machines – Brilliant Math & Science Wiki. brilliant.org. Retrieved 2018-04-14.
7. G.M.J.B. Chaslot; M.H.M. Winands; J.W.H.M. Uiterwijk; H.J. van den Herik; B. Bouzy (2008). "Progressive Strategies for Monte-Carlo Tree Search"
8. Goodwin, S. D., Menon, S., & Price, R. G. (2006). Pathfinding in open terrain.
9. https://alumni.media.mit.edu/~jorkin/goap.html
10. https://docs.Unity.com
11. https://en.wikipedia.org/
12. https://gamedevelopment.tutsplus.com/tutorials/goal-oriented-action-planning-for-a-smarter-ai--cms-20793
13. https://learn.unity.com/project/goal-driven-behavior

14 Nikos Avradinis*, Themis Panayiotopoulos and George Anastassakis, 2013 "Behavior believability in virtual worlds: agents acting when they need to"

15. Nikos Avradinis*, Themis Panayiotopoulos and George Anastassakis, 2013 "Modelling basic needs as agent motivations"

16 OpenAI, Dota 2 with Large Scale Deep Reinforcement Learning, March 10, 2021

17. Reeve J (2010) Understanding Motivation and Emotion. John Wiley & Sons,Hoboken, NJ
18. Silver, David; Hubert, Thomas; Schrittwieser, Julian; Antonoglou, Ioannis; Lai, Matthew; Guez, Arthur; Lanctot, Marc; Sifre, Laurent; Kumaran, Dharshan (2018-12-06). "A general reinforcement learning algorithm that masters chess, shogi, and Go through self
19. Silver, David; Hubert, Thomas; Schrittwieser, Julian; Antonoglou, Ioannis; Lai, Matthew; Guez, Arthur; Lanctot, Marc; Sifre, Laurent; Kumaran, Dharshan (2018-12-06). "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play"
20. Sturtevant, N. R. (June 2012). "Benchmarks for Grid-Based Pathfinding". IEEE Transactions on Computational Intelligence and AI in Games
21. Sweetser, Penelope; Wyeth, Peta (2005-07-01). "GameFlow". Computers in Entertainment
22. Wang, Jiacun (2019). Formal Methods in Computer Science. CRC Press. p. 34.
23. Yannakakis, G. N. (2012, May). Game AI revisited

24 Yap, Peter. "Grid-based path-finding." In Conference of the Canadian Society for Computational Studies of Intelligence, pp. 44-55. Springer, Berlin, Heidelberg, 2002

Appendix 1 - Keywords

Action	Action in this context, refers to the ScriptableObject that contains all the information needed for the Agent to perform an action.
ActionBase	Refers to the base class that all Action scripts inherit from. It is used to represent the Agent’s Actions.
Agent / AI Agent	Agent in this context refers to the AI Agent, be it a GOAP Agent or an FSM Agent.
Goal	Goal in this context refers to the ScriptableObject that represents a need from Maslow’s pyramid that also holds the end state that the Agent is trying to achieve. The end state may be comprised by many States or just one.
GoalBase	Refers to the base class that all Goal scripts inherit from. It is used to represent the Agent’s needs in the form of Goals.
Planner	Planner in this context refers to the script that calculates the best course of action for an Agent to achieve his Goal.
State	State in this context refers to the Key Value pairs used for describing the states of an Agent (e.g. Key: “HasEaten” Value: 1)
TimeGoalBase	Refers to the class that inherits from the Goal Base class and can adjust how quickly the Goal’s priority increases based on the simulation’s time of day.

Appendix 2 - System Specs

The project was created and executed on a system with the below specs:

OS	Windows 10 Pro
CPU	AMD’s RX5500XT
GPU	Intel’s i5 10400f
RAM	16Gb
PSU	Corsair’s TX550M
MONITOR	Dell S2721HN

Appendix 3 - Further Information

All of the non-referenced material written in this work, comes from either notes and presentations of Dr Panayiotopoulos's (Thesis supervisor) classes' material or the author's ideas.

- Most assets used in the project were created by the author using Blender version 3.
- Credits for the Icons used in the project's UI go to Icons8's Pichon (Microsoft Store App).
- The version of Unity used for the creation of the project was Unity 2020.3.2f1 Personal.