UNIVERISTY OF PIRAEUS - DEPARTMENT OF INFORMATICS

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ – ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

MSc «Informatics»

ΠΜΣ «Πληροφορική»

MSc Thesis

Μεταπτυχιακή Διατριβή

Thesis Title: Τίτλος Διατριβής:	**Creating a player-animal interaction farm simulator using smart Animal AI Agents that utilize the Behaviour Tree architecture in Unity.** Δημιουργία προσομοιωτή φάρμας αλληλεπίδρασης παίκτη-ζώου χρησιμοποιώντας έξυπνους πράκτορες ΤΝ που χρησιμοποιούν την αρχιτεκτονική του Δέντρου Συμπεριφοράς στη μηχανή Unity.
Student's name-surname: Ονοματεπώνυμο φοιτητή:	**Maria Traga** Μαρία Τράγα
Father's name: Πατρώνυμο:	**Nikolaos** Νικόλαος
Student's ID No: Αριθμός Μητρώου:	ΜΠΠΛ19058
Supervisor: Επιβλέπων:	**Themistoklis Panagiotopoulos, Professor** Θεμιστοκλής Παναγιωτόπουλος, Καθηγητής

July 2022 / Ιούλιος 2022

3-Member Examination Committee

Τριμελής Εξεταστική Επιτροπή

Themistoklis Panagiotopoulos	**Dionisios Sotiropoulos**	**Ioannis Tasoulas**
Professor	**Assistant Professor**	**Assistant Professor**
Θεμιστοκλής Παναγιωτόπουλος	Διονύσιος Σωτηρόπουλος	Ιωάννης Τασούλας
Καθηγητής	Επίκουρος Καθηγητής	Επίκουρος Καθηγητής

1. Acknowledgments/ Ευχαριστίες

Θα ήθελα να δώσω ένα μεγάλο ευχαριστώ στους γονείς μου που με στήριξαν σε κάθε μέρος της ζωής μου και μου έδωσαν όλη τη βοήθεια που χρειαζόμουν για να πραγματοποιήσω τα όνειρά μου. Επίσης, θα ήθελα να ευχαριστήσω τον Δρ. Θεμιστοκλή Παναγιωτόπουλο για τη σημαντική και χρήσιμη υποστήριξη που μου παρείχε και με καθοδήγησε καθ' όλη τη διάρκεια των ακαδημαϊκών μου σπουδών. Τέλος, θα ήθελα να ευχαριστήσω θερμά τον Δημήτρη Μάλλιαρη για τη βοήθειά του και την υποστήριξή του σε αυτή τη διατριβή και για την ενθάρρυνση του στη δημιουργία αυτού του έργου.

I would like to give a big thank you to my parents for supporting me in every part of my life and giving me all the help I needed to achieve my dreams. Furthermore, I would like to thank Dr. Themistoklis Panagiotopoulos for providing me with important and helpful support and for guiding me throughout my academic studies. Finally, I would like to give a heartfelt thank you to Dimitris Malliaris for his assistance and support on this thesis and for giving me hope and encouragement in creating this project.

2. Abstract

The master thesis project, implements smart A.I. animal agents by using the Behaviour Tree architecture. This implementation is handled by the Unity Engine in a game-like environment, where the user can interact with various parts of the game scene. Such interactions include the ability to plant crops which grow with the passage of time, the use of various tools on objects as well as the interaction with non-playable characters which are either the smart animal agents or stationary objects like the shop. Furthermore, the project features the passage of time from day to night and vice versa as well as an inventory system and a shop where items can be sold. Finally, the smart A.I. agents have specific needs that they need to satisfy as is defined by the structure of the Behaviour Tree. Specifically, their basic needs have been implemented, which are thirst, hunger, tiredness, affection towards the user and overall happiness which is expressed when the player character is nearby.

Keywords: Artificial Intelligence, Smart Animal Agents, Behaviour Tree Architecture, Unity engine

2. Περίληψη

Η ακόλουθη μεταπτυχιακή διατριβή, υλοποιεί έξυπνα ζώα με τεχνητή νοημοσύνη χρησιμοποιώντας την αρχιτεκτονική του Δέντρου Συμπεριφοράς. Αυτή η υλοποίηση έγινε μέσω της μηχανής Unity σε ένα περιβάλλον που προσομοιάζει παιχνίδι, όπου ο χρήστης μπορεί να αλληλεπιδράσει με διάφορα μέρη του κόσμου. Τέτοιες αλληλεπιδράσεις περιλαμβάνουν την ικανότητα να φυτεύονται καλλιέργειες που αναπτύσσονται με το πέρασμα του χρόνου, τη χρήση διαφόρων εργαλείων σε ορισμένα αντικείμενα καθώς και την αλληλεπίδραση με χαρακτήρες που είναι είτε οι έξυπνοι πράκτορες είτε ακίνητα αντικείμενα όπως το κατάστημα. Επιπλέον, το έργο περιλαμβάνει το πέρασμα του χρόνου από τη μέρα στη νύχτα και αντίστροφα καθώς και ένα σύστημα αποθήκευσης αντικειμένων αλλά και ένα κατάστημα για αγοραπωλησίες αυτών. Τέλος, οι έξυπνοι πράκτορες έχουν συγκεκριμένες ανάγκες που πρέπει να ικανοποιήσουν όπως ορίζεται από τη δομή του Δέντρου Συμπεριφοράς. Συγκεκριμένα, έχουν υλοποιηθεί οι βασικές τους ανάγκες που είναι η δίψα, η πείνα, η κούραση, η στοργή προς τον χρήστη και η συνολική τους ευτυχία.

Λέξεις-κλειδιά: Τεχνητή Νοημοσύνη, Έξυπνα Ζώα Πράκτορες, Αρχιτεκτονική του Δέντρου Συμπεριφοράς, Μηχανή Unity

3. Table of Contents

4. List of Figures

5. List of Abbreviations

2D	Two-dimensional
3D	Three-dimensional
AI	Artificial intelligence
API	Application programming interface
BT	Behaviour tree
CPU	Central processing unit
FSM	Finite state machine
GOAP	Goal oriented action planning
HDRP	High Definition Render Pipeline
MCTS	Monte Carlo tree search
ML	Machine learning
NN	Neural network
NPC	Non-player character
STRIPS	Stanford Research Institute Problem Solver
UI	User interface
URP	Universal Render Pipeline

6. Introduction

In the following sections, the most important aspects of the project will be explained. Most importantly, Artificial Intelligence and its multiple versions and implementations as well as the Maslow's Hierarchy of Needs theorem, in order to explain the behaviour implemented on the smart animal agents.

7. Artificial Intelligence

7.1. Intelligent Agents & Artificial Intelligence

In artificial intelligence, an intelligent agent (I.A.) is anything which perceives its environment, takes actions autonomously in order to achieve goals, and may improve its performance with learning or may use knowledge. They may be simple or complex — a thermostat is considered an example of an intelligent agent, as is a human being, as is any system that meets the definition, such as a firm, a state, or a biome (Russell & Norvig, 2003). Intelligent agents is an area of interest that attracts researchers from different Artificial Intelligence fields, such as distributed artificial intelligence, AI Planning and robotics, as well as classical computer science fields, such as information systems, databases, and human-computer interaction. The adjective "intelligent" is used to denote the involvement of AI research in agent technology. AI has been considered as the main contributor to the field of intelligent agents (Marinagi, Panayiotopoulos & Spyropoulos, 2005).

Leading AI textbooks define "artificial intelligence" as the "study and design of intelligent agents", a definition that considers goal-directed behavior to be the essence of intelligence. These agents are, as mentioned before, computational entities that perceive environmental conditions, act to affect conditions and reason about conditions and actions (Marinagi, Panayiotopoulos & Spyropoulos, 2005).

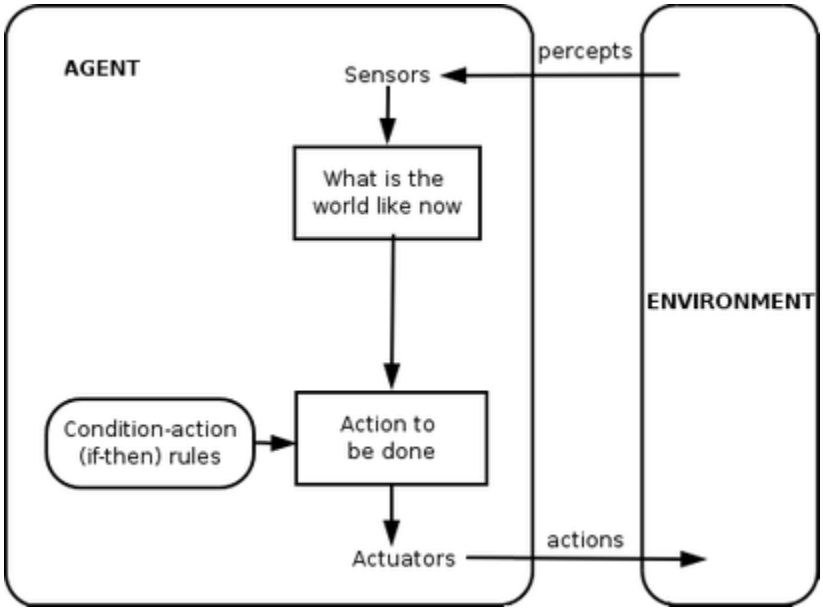


Figure 1 - Simple reflex agent diagram

The reactive agent architecture was introduced in order to allow robust performance in dynamic environments, where the deliberative agents fail to perform. We divide reactive agent architectures into three subcategories that signify the different underlying ideas on which they are based. These are: pure reactive agents, simple reactive planning agents, and sophisticated reactive planning agents. Pure reactive agents act without planning and do not include a symbolic model of the world. The reactive planning approach adds an AI point of view to agents. They include a symbolic model of the world and apply reactive reasoning to choose between alternative plans at run-time. Such reactive planners are considered as simple, while sophisticated reactive planners include more complicated constructs in order to handle execution failures or environmental changes (Marinagi, Panayiotopoulos & Spyropoulos, 2005).

According to Russell & Norvig (2003), agents can be classified based on their degree of perceived intelligence and capability as follows:

- Simple reflex agents
- Model-based reflex agents
- Goal-based agents
- Utility-based agents
- Learning agents

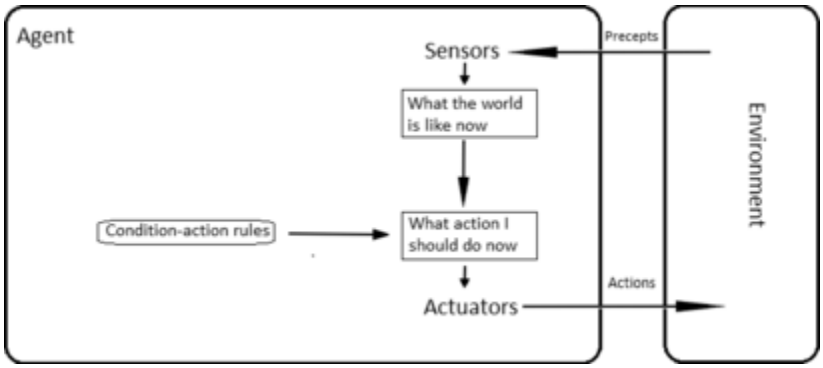


Figure 2 - Simple reflex agent

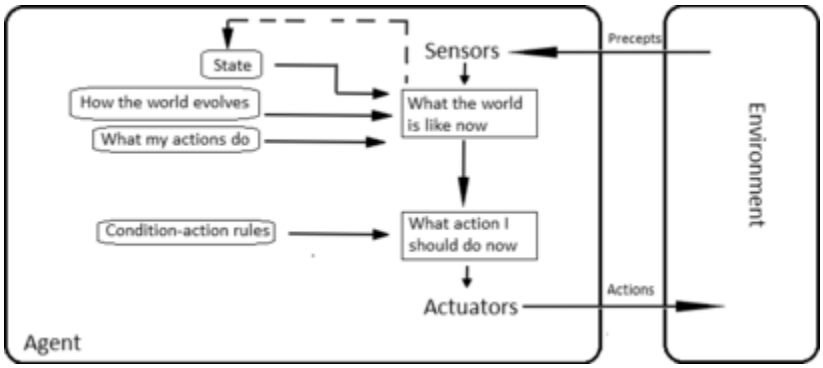


Figure 3 - Model-based reflex agent

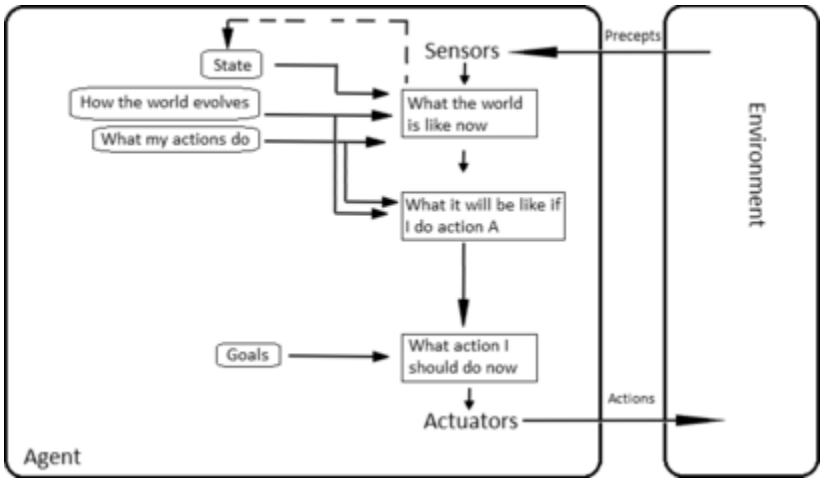


Figure 4 - Model-based, goal-based agent

Artificial intelligence is based on the principle that human intelligence can be defined in a way that a machine can easily mimic it and execute tasks, from the most simple to those that are even more complex. The goals of artificial intelligence include mimicking human cognitive activity. Researchers and developers in the field are making surprisingly rapid strides in mimicking activities such as learning, reasoning, and perception, to the extent that these can be concretely defined. Some believe that innovators may soon be able to develop systems that exceed the capacity of humans to learn or reason out any subject. But others remain skeptical because all cognitive activity is laced with value judgments that are subject to human experience.

As technology advances, previous benchmarks that defined artificial intelligence become outdated. For example, machines that calculate basic functions or recognize text through optical character recognition are no longer considered to embody artificial intelligence, since this function is now taken for granted as an inherent computer function.

AI is continuously evolving to benefit many different industries. Machines are wired using a cross-disciplinary approach based on mathematics, computer science, linguistics, psychology, and more.

7.2. AI in video games

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.

While AI in some form has long appeared in video games, it is considered a booming new frontier in how games are both developed and played. AI games increasingly shift the control of the game experience toward the player, whose behavior helps produce the game experience.

Artificial intelligence has been an integral part of video games since their inception in the 1950s. 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. (Grant, Eugene F.; Lardner, Rex, 1952).

While the vast majority of AI academics (including the author) would claim that games are fully scripted and still use 30-year old AI technology — such as A* and finite state machines — the game industry had been making small, yet important, steps towards integrating nouvelle (or modern) AI (A.J. Champandard, 2004) in their games (S. Woodcock, 2001) during the early days of game AI. A non-inclusive list of games that advanced the game AI state-of-practice in industry (S.Rabin, 2002) includes the advanced sensory system of guards in Thief (EIDOS, 1989); the advanced opponent tactics in Half-Life (Valve, 1998); the fusion of machine learning techniques such as perceptrons, decision trees and reinforcement learning coupled with the belief-desire intention cognitive model in Black and White (EA, 2000); the dynamic difficulty adjustment (DDA) features in the Halo series (MS Game Studios); the imitation learning Drivatar system of Forza Motorsport (MS Game Studios, 2005); the AI director of Left 4 Dead (Valve, 2008) 2 and the neuroevolutionary training of platoons in Supreme Commander 2 (Square Enix, 2010). (Georgios N. Yannakakis, 2012, Game AI revisited). All in all, the most common AI architectures used in video games are:

- Finite State Machines (FSM)

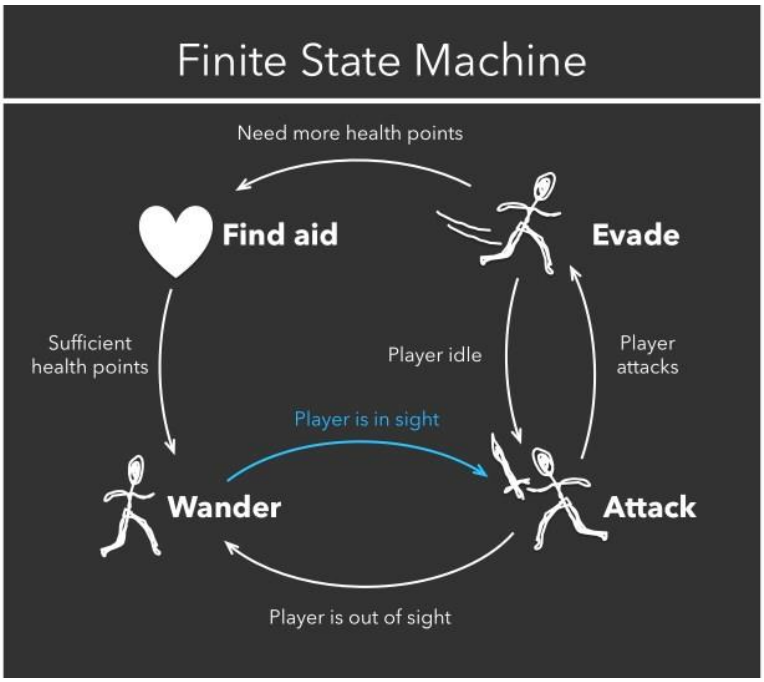


Figure 5 - Finite State Machines

A finite state machine is an abstract machine that can exist in one of several different and predefined states. A finite state machine also can define a set of conditions that determine when the state should change. The actual state determines how the state machine behaves. Only a single state can be

active at the same time, so the machine must transition from one state to another in order to perform different actions.

Finite state machines date back to the earliest days of computer game programming. For example, the ghosts in Pac Man are finite state machines. They can roam freely, chase the player, or evade the player. In each state they behave differently, and their transitions are determined by the player's actions. For example, if the player eats a power pill, the ghosts' state might change from chasing to evading. We'll come back to this example in the next section.

Although finite state machines have been around for a long time, they are still quite common and useful in modern games. The fact that they are relatively easy to understand, implement, and debug contributes to their frequent use in game development. In this chapter, we discuss the fundamentals of finite state machines and show you how to implement them.

- Monte Carlo Search Tree (MCST)

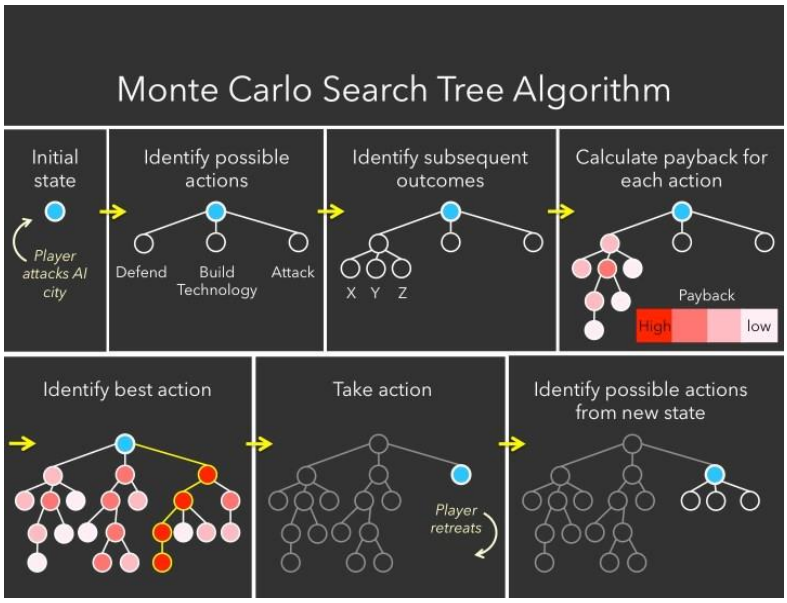


Figure 6 - Monte Carlo Search Tree (MCST)

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.

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. 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:

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.

MCTS is a simple algorithm to implement. Moreover, Monte Carlo Tree Search is a heuristic algorithm and can operate effectively without any knowledge in the particular domain, apart from the rules and end conditions, and can find its own moves and learn from them by playing random playouts.

However, as the tree growth becomes rapid after a few iterations, it requires a huge amount of memory. Also, there is a bit of a reliability issue with Monte Carlo Tree Search. In certain scenarios, there might be a single branch or path, that might lead to loss against the opposition when implemented for those turn-based games. This is mainly due to the vast number of combinations and each of the nodes might not be visited enough number of times to understand its result or outcome in the long run. Finally, MCTS algorithm needs a huge number of iterations to be able to effectively decide the most efficient path, thus, there exists a speed issue.

- Behavioral Decision Trees (BT)

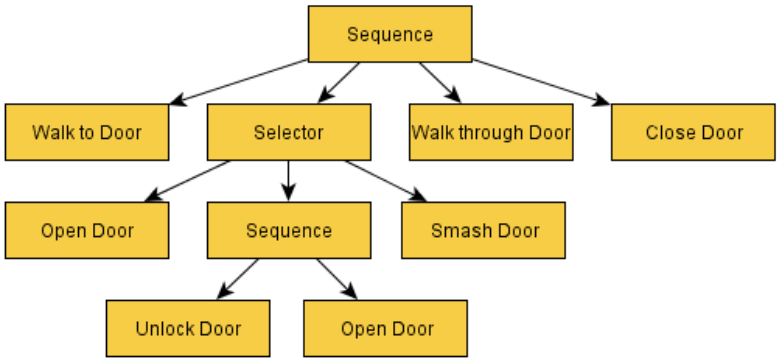


Figure 7 - Behavioral Decision Trees (BT)

A behavior tree is a mathematical model of plan execution used in computer science, robotics, control systems and video games. They describe switchings between a finite set of tasks in a modular fashion. Their strength comes from their ability to create very complex tasks composed of simple tasks, without worrying how the simple tasks are implemented. Behavior trees present some similarities to hierarchical state machines with the key difference that the main building block of a behavior is a task rather than a state. Its ease of human understanding make behavior trees less error prone and very popular in the game developer community. Behavior trees have been shown to generalize several other control architectures.

Behavior trees originate from the computer game industry as a powerful tool to model the behavior of non-player characters (NPCs). They have been extensively used in high-profile video games such as Halo, Bioshock, and Spore

Unlike a Finite State Machine, or other systems used for AI programming, a behaviour tree is a tree of hierarchical nodes that control the flow of decision making of an AI entity. At the extents of the tree, the leaves, are the actual commands that control the AI entity, and forming the branches are various types of utility nodes that control the AI's walk down the trees to reach the sequences of commands best suited to the situation.

The trees can be extremely deep, with nodes calling sub-trees which perform particular functions, allowing for the developer to create libraries of behaviours that can be chained together to provide very convincing AI behaviour. Development is highly iterable, where one can start by forming a basic behaviour, then create new branches to deal with alternate methods of achieving goals, with branches ordered by their desirability, allowing for the AI to have fallback tactics should a particular behaviour fail.

A behaviour tree is made up of several types of nodes, however some core functionality is common to any type of node in a behaviour tree. This is that they can return one of three statuses, which are Success, Failure or Running.

The first two, as their names suggest, inform their parent that their operation was a success or a failure. The third means that success or failure is not yet determined, and the node is still running. The node will be ticked again next time the tree is ticked, at which point it will again have the opportunity to succeed, fail or continue running. There are three main archetypes of a behaviour tree node: Composite, Decorator, Leaf/Action.

Composite

A composite node is a node that can have one or more children. They will process one or more of these children in either a first to last sequence or random order depending on the particular composite node in question, and at some stage will consider their processing complete and pass either success or failure to their parent, often determined by the success or failure of the child nodes. During the time they are processing children, they will continue to return Running to the parent.

Decorator

A decorator node, like a composite node, can have a child node. Unlike a composite node, they can specifically only have a single child. Their function is either to transform the result they receive from their child node's status, to terminate the child, or repeat processing of the child, depending on the type of decorator node.

Leaf/Action

These are the lowest level node type, and are incapable of having any children. Leafs are however the most powerful of node types, as these will be defined and implemented by the game to do the game specific or character specific tests or actions required to make the tree do useful stuff.

- Goal Oriented Action Planning (GOAP)

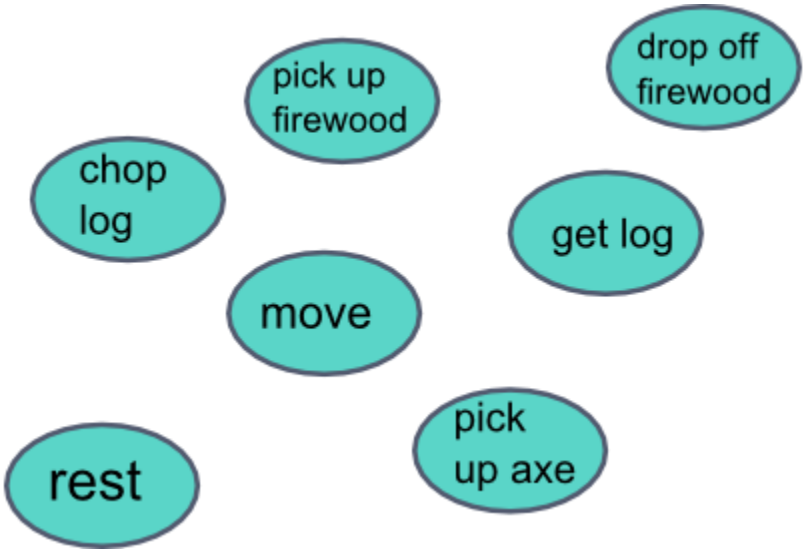


Figure 8 - Goal Oriented Action Planning (GOAP)

Goal oriented action planning is an artificial intelligence system for agents that allows them to plan a sequence of actions to satisfy a particular goal. The particular sequence of actions depends not only on the goal but also on the current state of the world and the agent. This means that if the same goal is supplied for different agents or world states, you can get a completely different sequence of actions., which makes the AI more dynamic and realistic.

GOAP refers to a simplified STRIPS-like planning architecture specifically designed for real-time control of autonomous character behavior in games. The specified AI helps decouple the actions from each other, and allows for focusing on each action individually.

- Neural Networks / Machine Learning (NN/ML)

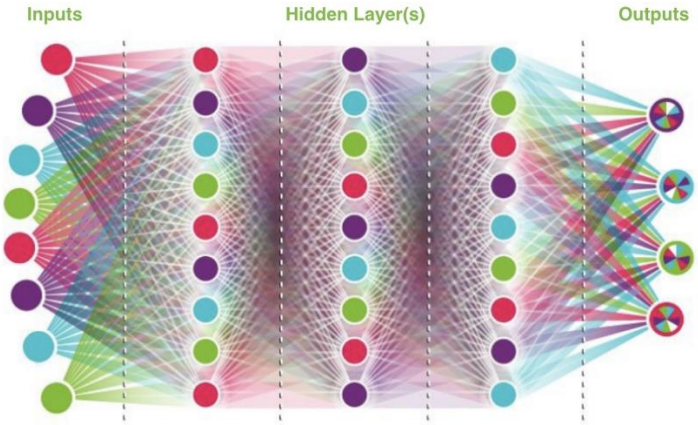


Figure 9 - Neural Networks / Machine Learning (NN/ML)

Machine learning is a subset of artificial intelligence that focuses on using algorithms and statistical models to make machines act without specific programming. This is in sharp contrast to traditional methods of artificial intelligence such as search trees and expert systems.

Information on machine learning techniques in the field of games is mostly known to public through research projects as most gaming companies choose not to publish specific information about their intellectual property. The most publicly known application of machine learning in games is likely the use of deep learning agents that compete with professional human players in complex strategy games. There has been a significant application of machine learning on games such as Atari/ALE, Doom, Minecraft, StarCraft, and car racing. Other games that did not originally exists as video games, such as chess and Go have also been affected by the machine learning.

7.3. Animal Behaviour

An essential element in intelligent virtual agents is the concept of believability, a notion that refers to creating the illusion of interaction with living characters, as mentioned in (Riedl and Young, 2005). This sense of believability is in turn of major importance in order to enhance the sense of immersion and presence, elements that according to Zeltzer (1992), are among the sine qua non for an engaging virtual reality experience.

In this project, in order to give ‘life’ to the animal NPCs the Behavioural Decision Tree architecture was used. The implementation of the produced Behaviour Tree objects was done via the Unity Engine and the C# programming language. Before proceeding to the actual implementation of these said Trees, first the AI agents’ motivations must be understood. This is of great importance and is vital in order to simulate a believable animal-behaving agent.

7.4. Animal needs

All living organisms have a plethora of needs in order to live a healthy life. Basic needs such as air, water, food and protection from environmental dangers are necessary for an organism to live. 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.

Maslow's hierarchy of needs is an idea in psychology proposed by American Abraham Maslow in his 1943 paper "A Theory of Human Motivation" in the journal Psychological Review (Maslow, A.H. 1943).

Figure 10 - Maslow's Motivation Model

Maslow's hierarchy consists of:

- Physiological
- Safety
- Love/Belonging
- Esteem and
- Self-actualization needs.

Although this hierarchy's main goal is to describe what drives the human species, it can also be applied to a certain extent to other animals. Below, follows an implementation of the above hierarchy modified to meet dogs' needs and how it affects their training and relationship with their human companion.

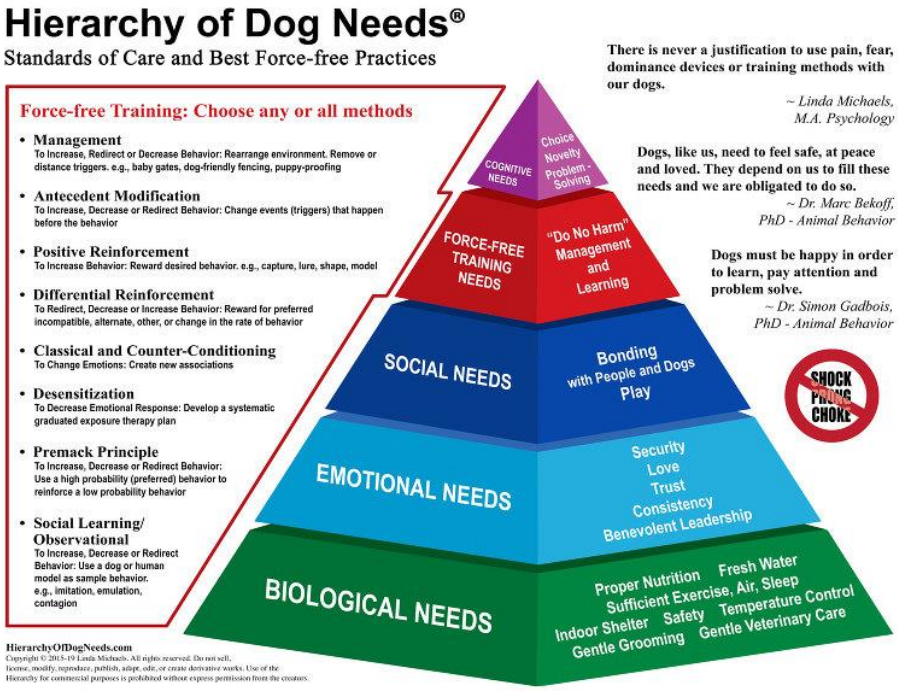


Figure 11 - Hierarchy of Dog Needs

The above hierarchy was designed by Linda Michaels, M.A., Psychology and it can be understood that it is not that different from that of a human's. Moreover, another implementation that concerns farm animals can be seen in the next figure.

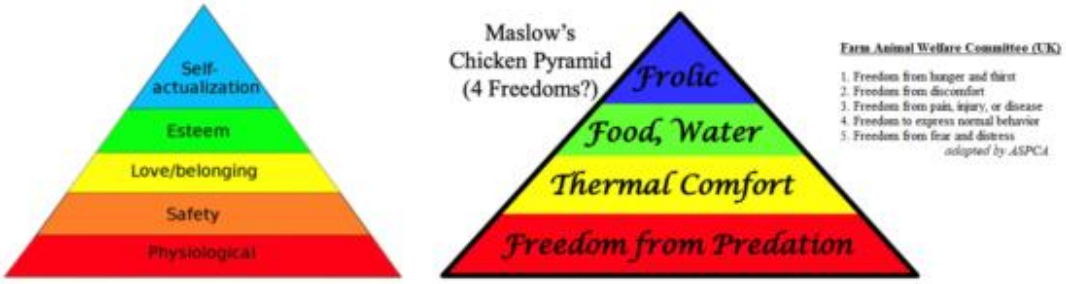
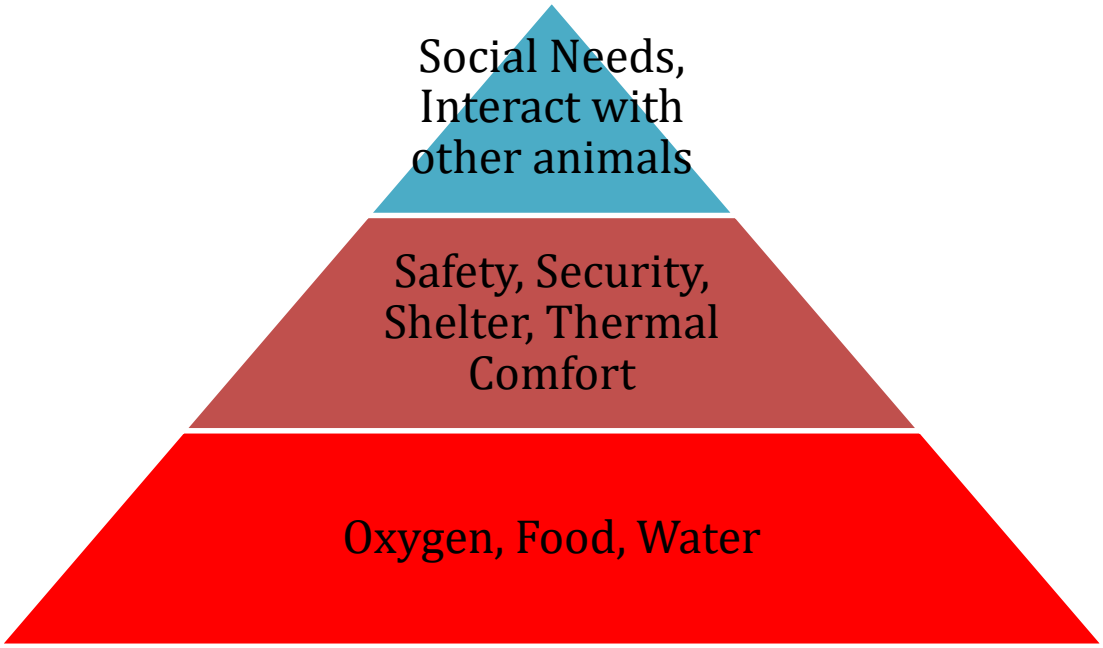


Figure 12 - Hierarchy of Chicken Needs

In conclusion a general simplified figure of farm animal's hierarchy of needs could be the figure below which is the one that is going to be used in this project.


Social Needs, Interact with other animals
Safety, Security, Shelter, Thermal Comfort
Oxygen, Food, Water

7.5. Unity Engine

Unity is a cross-platform game engine developed by Unity Technologies, first announced and released in June 2005 at Apple Worldwide Developers Conference as a Mac OS X game engine. The engine has since been gradually extended to support a variety of desktop, mobile, console and virtual reality platforms. It is particularly popular for iOS and Android mobile game development and is considered easy to use for beginner developers and is popular for indie game development.

Figure 13. 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.

Unity gives users the ability to create games and experiences in both 2D and 3D, and the engine offers a primary scripting API in C# using Mono, for both the Unity editor in the form of plugins, and games themselves, as well as drag and drop functionality. Prior to C# being the primary programming language used for the engine, it previously supported Boo, which was removed with the release of Unity 5, and a Boo-based implementation of JavaScript called UnityScript, which was deprecated in August 2017, after the release of Unity 2017.1, in favor of C#.

Within 2D games, Unity allows importation of sprites and an advanced 2D world renderer. For 3D games, Unity allows specification of texture compression, mipmaps, and resolution settings for each platform that the game engine supports, and provides support for bump mapping, reflection mapping, parallax mapping, screen space ambient occlusion (SSAO), dynamic shadows using shadow maps, render-to-texture and full-screen post-processing effects.

Two separate render pipelines are available, High Definition Render Pipeline (HDRP) and Universal Render Pipeline (URP), in addition to the legacy built-in pipeline. All three render pipelines are incompatible with each other. Unity offers a tool to upgrade shaders using the legacy renderer to URP or HDRP.

Running a Unity script executes a number of event functions in a predetermined order. The diagram below summarizes how Unity orders and repeats event functions over a script's lifetime.

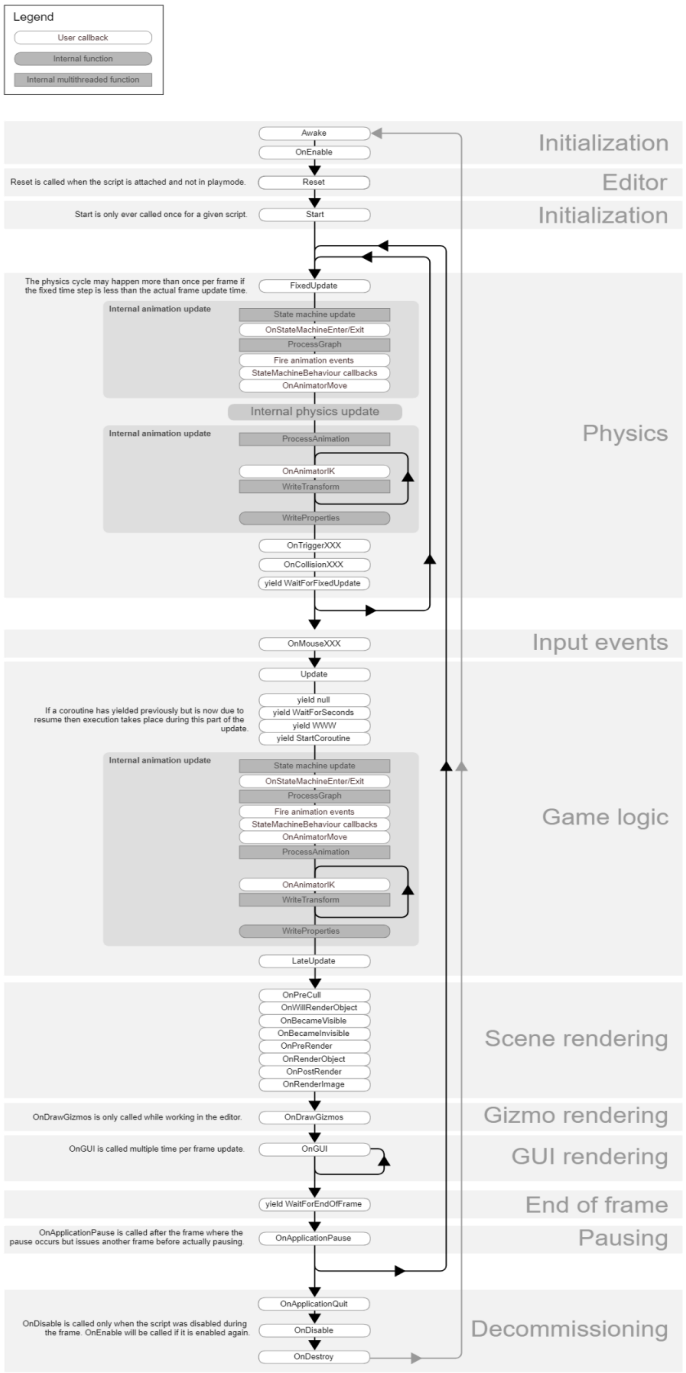


Figure 14 – Unity Life Cycle

8. Project Application

In order for the game to be created, the Unity Engine 2020.3.2f1 was used and the programming language for the scripts was C#. The game features, which will be further analyzed in the following sections, are the following:

- Animals that behave according to their defined Behaviour Tree AI.
- Likeness of the animals towards certain actions of the player.
- Crops that grow and produce various items.
- Inventory system that allows the player to store items on the internal inventory as well as external storage containers.
- Day and night cycle that cycles through the time of each day.
- Interactable objects such as a shop and NPCs.

8.1. Game Features

The core feature of the game are as follows:

8.1.1. AI Animal Agents

When referring to AI animal agents, I mean the agents that will implement the Behaviour Tree architecture. These animals will act upon their defined AI and, in accordance with the respective data that each animal holds, will select which action to take (e.g. drink water, eat, etc.). Moreover, the animals hold data regarding their likeness towards the player which is calculated based on the interactions between them.

When the animals are content based on the necessary calculations, they will express their feeling when the player is nearby and will produce certain items depending on whether their needs are satisfied or not. The animals and their Behaviour Tree will be further explained in the following sections.

8.1.2. Crop System

A feature that is present in the game is the ability to purchase seeds and grow crops that will produce various items. These items can then be sold for profit.

Figure 15 - Crop System Shop

As seen above, the player has 5 coins available and can purchase certain seeds that are in stock from the shop. After purchasing the seeds, the player can seed them in fertile soil, as presented on the image below.

Figure 16 - Crop System Planting

The crops will start growing with varying speed based on the plant and when they get fully grown, they can be harvested with an empty hand. Each crop is stored in a scriptable object that defines the item that will spawn after harvesting, as well as the amount, the time it takes to grow, etc.

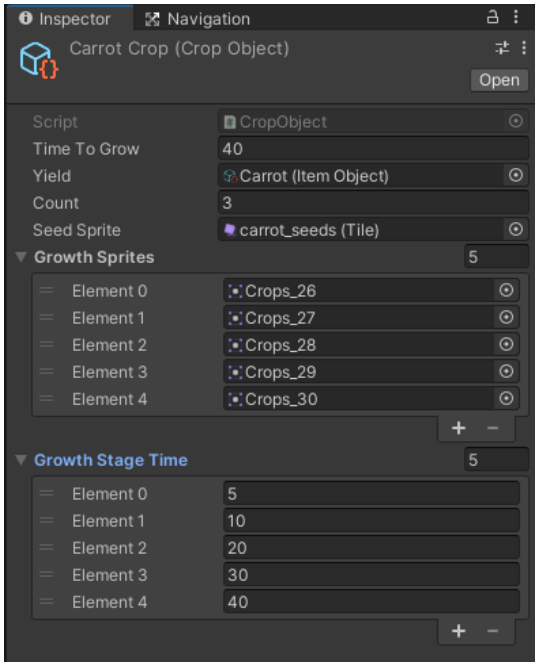


Figure 17 - Crop Object

Moreover, in order for the crops to be displayed on the map, the location of each crop on the tilemap must be stored. To achieve that, I created a scriptable object to hold all the positions available for growing crops. As such, when a crop gets planted its necessary data will be stored as presented below.

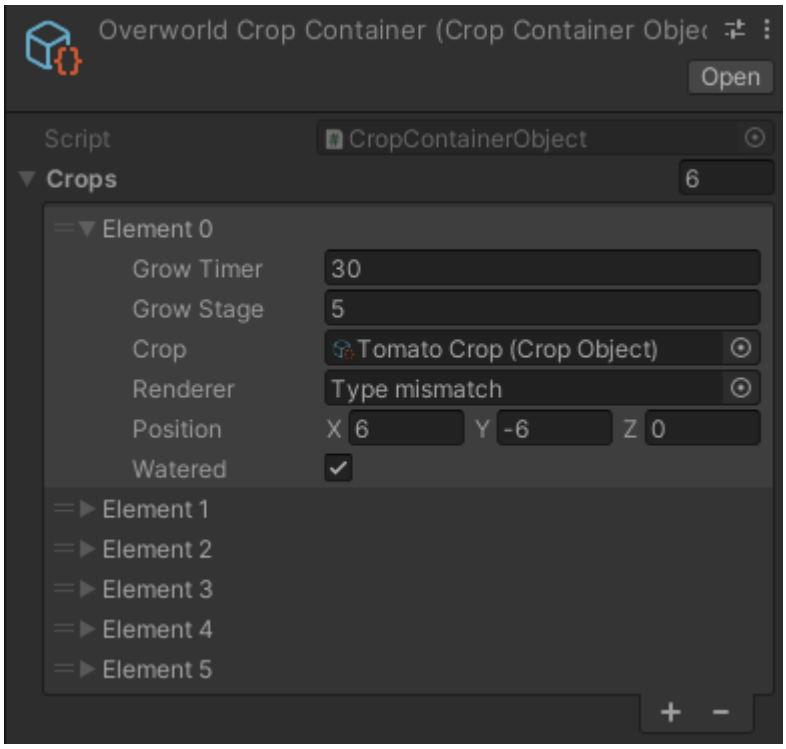


Figure 18 - Crop Container Object

The item is also stored in a scriptable object, which has the following data.

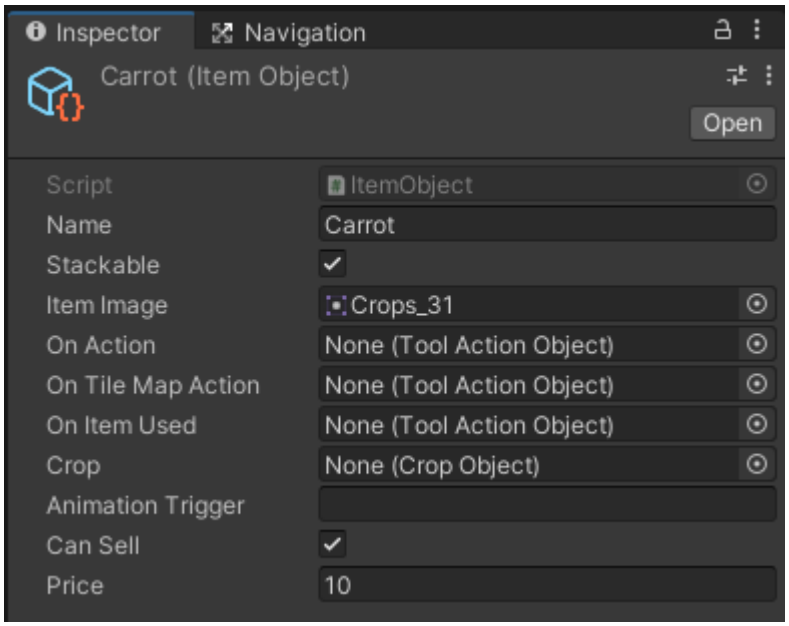


Figure 19 - Item Object

Each item can either be stackable or not. This means that it will either be a tool which can only exist once for the player (e.g. an axe) or it will be an object that the player will want to have more than one in order to sell or store (e.g. carrots).

8.1.3. Inventory System

In order for the player to have the items that have been described above, an inventory system must be implemented. When the player wants to access the inventory, the “I” button must be pressed, which displays the screen below.

Figure 20 - Inventory System Player Inventory

Furthermore, aside from the player inventory there are multiple storages available, such as chests, where the player can deposit or withdraw items.

Figure 21 - Inventory System Chest Container

The scriptable object architecture was used in the inventory system, in order for the items to have their position and quantity display correctly when the player interacts with any storage.

Each inventory/storage is a scriptable object, which holds the slots of the container (e.g. 30 for player inventory). Consequently, each slot holds the item object and the amount of the item, as shown below.

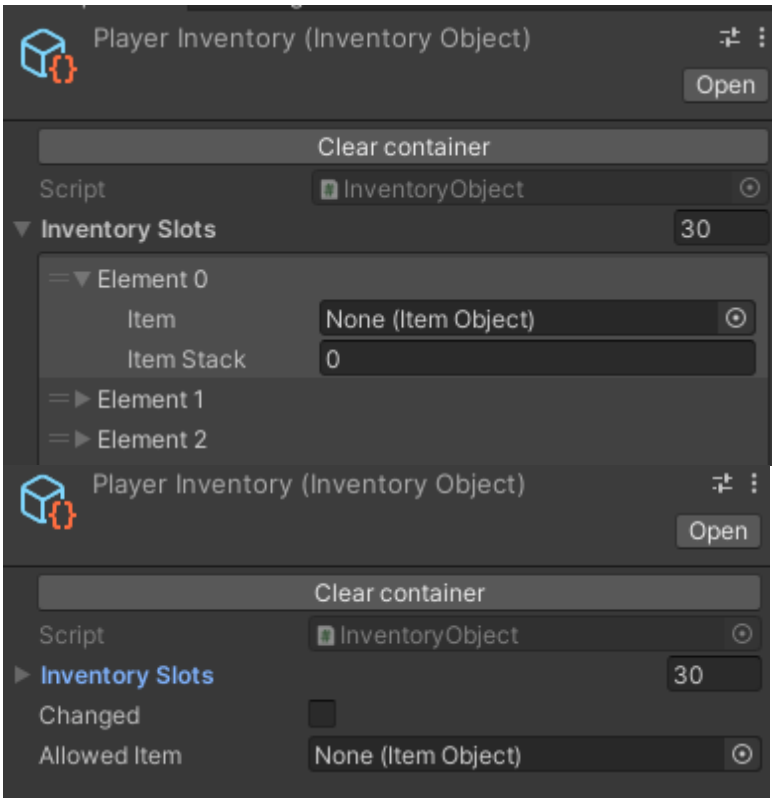


Figure 22 - Inventory Object

In certain cases, some storage containers need to accept only one specific item (e.g. animal feeder accepts only hay), and the player will only be able to deposit the specified item.

8.1.4. Day and Night System

Another system that has been implemented is the change between day and night as well as the passing of each day. Moreover, a time agent has been created which depends on the day/night system, in order to invoke methods on certain intervals (e.g. 15 in game seconds).

When the in game time reaches a certain point, the appearance of the map changes with the help of the universal render pipeline (URP) to simulate the perspective of night in a 2D environment.

Figure 23 - Day and Night System

The global volume of the URP changes by evaluating a defined animation curve.

Finally, when the day changes into night, all the lights on the map get activated to give the illusion of lighting on a 2D environment. The lights implement the point light 2D system of the Unity Engine.

8.2. Project Setup

In order for the application to function as intended, certain scripts with relative game logic needed to be implemented. The gameobjects that hold the majority of the most important scripts are:

- Game Manager
- Player

8.2.1. Game Manager Scripts

The Game Manager gameobject holds some of the most necessary scripts for the game. Such scripts will be presented in the following paragraphs.

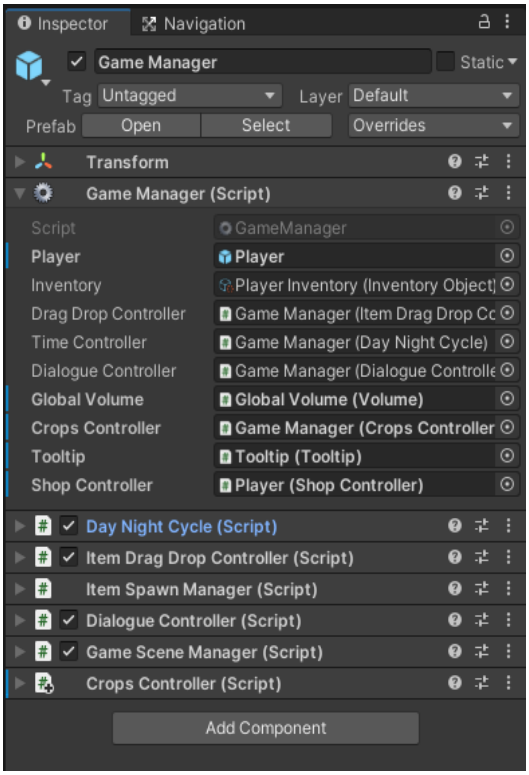


Figure 24 - Game Manager Gameobject

Game Manager Singleton

```
Unity Script (1 asset reference) | 43 references
public class GameManager : MonoBehaviour
{
    private static GameManager _instance; //singleton

    41 references
    public static GameManager Instance { get {return _instance;} }

    Unity Message | 0 references
    private void Awake()
    {
        if(_instance!=null && _instance != this)
        {
            Destroy(this.gameObject);
        }
        else
        {
            _instance = this;
        }
    }

    public GameObject player;
    public InventoryObject inventory;
    public ItemDragDropController dragDropController;
    public DayNightCycle timeController;
    public DialogueController dialogueController;
    public Volume globalVolume;
    public CropsController cropsController;
    public Tooltip tooltip;
    public ShopController shopController;
}
```

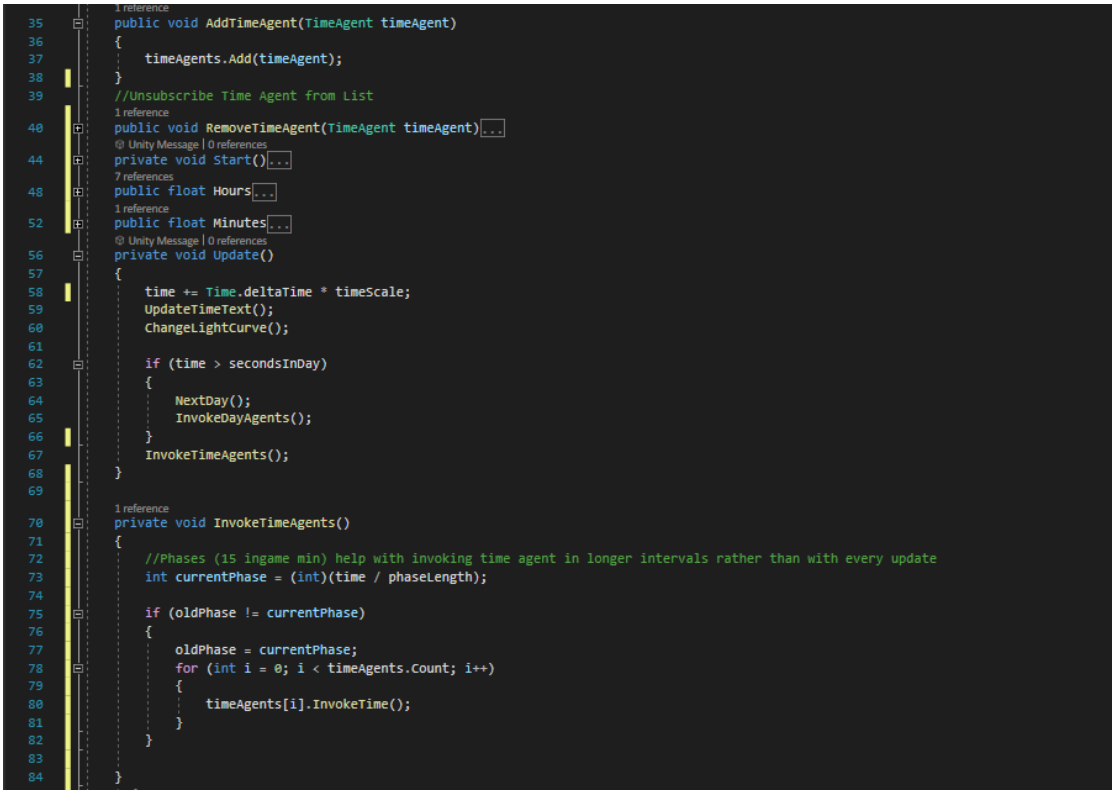
Figure 25 - Game Manager Singleton Script

The GameManager script is a singleton script which ensures that when the instance gets created it will always remain the same and will not be destroyed when, for example, another scene is loaded.

Generally, a singleton in Unity is a global accessible class that exists in the scene, but can only exist once. Any other script can access the singleton, allowing for easy connection between unrelated objects and global systems such as the audio manager, by using the Instance variable.

The singleton script above holds the references for certain gameobjects that will also get the abilities of a singleton, for easier use throughout the project.

Day Night Cycle Script



```
1 reference
public void AddTimeAgent(TimeAgent timeAgent)
{
    timeAgents.Add(timeAgent);
}
//Unsubscribe Time Agent from List
1 reference
public void RemoveTimeAgent(TimeAgent timeAgent)...
Unity Message | 0 references
private void Start()...
7 references
public float Hours...
1 reference
public float Minutes...
Unity Message | 0 references
private void Update()
{
    time += Time.deltaTime * timeScale;
    UpdateTimeText();
    ChangeLightCurve();

    if (time > secondsInDay)
    {
        NextDay();
        InvokeDayAgents();
    }
    InvokeTimeAgents();
}

1 reference
private void InvokeTimeAgents()
{
    //Phases (15 ingame min) help with invoking time agent in longer intervals rather than with every update
    int currentPhase = (int)(time / phaseLength);

    if (oldPhase != currentPhase)
    {
        oldPhase = currentPhase;
        for (int i = 0; i < timeAgents.Count; i++)
        {
            timeAgents[i].InvokeTime();
        }
    }

}
```

Figure 26 - Day Night Cycle Script

The script above handles the change from day to night, holds the amount of days that passed from the beginning of the game and invokes certain time agents on specific time periods (e.g. once per tick, once per day).

The time agents utilize the events architecture of the Unity Engine. Specifically, it is possible to create modular connections between scripts and objects by using events and delegates. These help trigger game logic as it happens, without relying on tight connections between scripts.

In order to handle certain states of a script it is best to use the event system, considering that if everything is checked every update, meaning every frame, then it could be highly constraining and burdening for the game. For example, the time agent gets invoked every 15 ingame minutes, which is significantly less time than handling a certain check every frame.

A method that helps utilize events is the Observer Pattern. This pattern allows for creating modular logic that can be executed when an event is triggered. It typically works by allowing observers, in this case, other scripts, to subscribe one or more of their own functions to a subject's event. Then, when the event is

triggered by the subject, the observers' functions are called in response. For example, the player's health script could declare an On Player Death event, that's called when the player runs out of health.

This means that connecting different pieces of game logic with the actual events of a game can be done easily without needing to manage specific script to script connections.

Finally, in order to create an observer-style system in Unity, a common method is to use delegates, which are essentially function containers. They allow for storing and calling a function as if it were a variable.

Item Drag Drop Controller Script

```
2 references
public void OnClick(InventorySlot inventorySlot)
{
    //if the pointer copy of the inventory slot is empty (nothing has been coppied/picked up)
    //then create a copy and clear the inventory slot in the button
    if (this.inventorySlot.Item == null)
    {
        this.inventorySlot.Copy(inventorySlot);
        inventorySlot.Clear();
    }
    //if a copy already exists in the pointer
    else
    {
        //Create an item object and the stack count for the coppied item in the pointer
        ItemObject item = this.inventorySlot.Item;
        int count = this.inventorySlot.ItemStack;

        //If the pointer holds the same item as the inventory slot and it is stackable then add them together
        if (this.inventorySlot.Item == inventorySlot.Item && this.inventorySlot.Item.Stackable)
        {
            inventorySlot.Set(item, inventorySlot.ItemStack + count);
            this.inventorySlot.Clear();
            dragDropImage.SetActive(false);
        }
        else
        {
            //Create a copy in the pointer (overwriting the previous one)
            //from the inventory slot still in the inventory (from the button)
            this.inventorySlot.Copy(inventorySlot);
            //Set the variables saved above in the button inventory slot
            inventorySlot.Set(item, count);

        }

        //Note: If the button inventory slot is empty then the coppied item will also be empty
    }

    UpdateDragDropImage();
}


//Enable/Disable the image shown on pointer
1 reference
private void UpdateDragDropImage()...

1 reference
private void SetDragDropImagePosition()...
```

Figure 27 - Item Drag Drop Controller Script

The player can drag and drop an item from the inventory to any storage, as well as, drop the selected item on the world. The script shown above, handles the respective feature by creating a copy of the item clicked in any storage on the mouse. When the item is clicked, an image is shown to the player, following the mouse movement, with the selected item and an instance of the item data (e.g. item object, stack amount) is stored temporarily in the memory. After the player clicks for a second time, either on the overworld or in any slot in a storage, the held item will be added either as a gameobject instance or will be added in the inventory slot of the active inventory scriptable object, respectively. Following that, the stored data will be cleared from the memory.

Game Scene Manager Script

```
1 reference
public void InitSwitchScene(string sceneName, Vector3 targetPosition, VolumeProfile volumeProfile, AudioClip walkSound)
{
    StartCoroutine(HandleTransition(sceneName, targetPosition, volumeProfile, walkSound));
}

1 reference
IEnumerator HandleTransition(string sceneName, Vector3 targetPosition, VolumeProfile volumeProfile, AudioClip walkSound)
{
    HandleMovementOnTransition(false);
    transitionScreenTint.Tint(true);

    yield return new WaitForSeconds(1f / transitionScreenTint.tintSpeed + 0.1f);

    SwitchScene(sceneName,targetPosition,volumeProfile, walkSound);

    while (load != null & unload != null)
    {
        if (load.isDone)
            load = null;
        if (unload.isDone)
            unload = null;
        yield return new WaitForEndOfFrame();
    }

    SceneManager.SetActiveScene(SceneManager.GetSceneByName(currentScene));

    cameraConfinerController.UpdateBounds();
    transitionScreenTint.Tint(false);
    HandleMovementOnTransition(true);
}

1 reference
public void SwitchScene(string sceneName, Vector3 targetPosition, VolumeProfile volumeProfile, AudioClip walkSound)
{
    load = SceneManager.LoadSceneAsync(sceneName, LoadSceneMode.Additive);
    unload = SceneManager.UnloadSceneAsync(currentScene);
    currentScene = sceneName;

    Transform playerTransform = GameManager.Instance.player.transform;

    Cinemachine.CinemachineBrain currentCamera = Camera.main.GetComponent<Cinemachine.CinemachineBrain>();
    currentCamera.ActiveVirtualCamera.OnTargetObjectWarped(playerTransform, targetPosition - playerTransform.position);

    GameManager.Instance.player.transform.position = targetPosition;
    GameManager.Instance.globalVolume.profile = volumeProfile;
    GameManager.Instance.player.GetComponent<CharacterController2D>().walkSound = walkSound;
}
```

Figure 28 - Game Scene Manager Script

When the player reaches certain spots on the map, a transition gets called that can either be moving, on the same scene, in a specific location or switching between different scenes.

The GameSceneManager script, handles the case of switching between scenes. In order for the transition to occur smoothly, the coroutine functionality of Unity is used. When the transition happens, for seconds, the screen turns darker gradually, and then the first scene gets unloaded, while the second scene gets the respective references and gets loaded. After everything is set the screen shows the changed scene.

A coroutine allows the spread of tasks across several frames. In most situations, when a method is called, it runs to completion and then returns control to the calling method, plus any optional return values. This means that any action that takes place within a method must happen within a single frame update. In Unity, a coroutine is a method that can pause execution and return control to Unity but then continue where

it left off on the following frame. However, coroutines are not threads and any code that gets executed within them will run on the main thread.

Crops Controller Script

```
//Stores current state of the crop tile
[Serializable]
13 references
public class CropTile
{
    public int growTimer;
    public int growStage = 0;

    public CropObject crop;
    public SpriteRenderer renderer;
    public Vector3Int position;

    public bool watered = false;

    2 references
    public bool Completed...

    1 reference
    public void Harvested()...
}

Unity Script (1 asset reference) | 1 reference
public class CropsController : MonoBehaviour
{
    public CropsTilemapController cropsTilemapController;

    1 reference
    public void PickUpTile(Vector3Int position)...

    1 reference
    public bool CheckIfPlantedTile(Vector3Int position)...

    1 reference
    public void PlantSeed(Vector3Int position, CropObject cropToSeed)...

    1 reference
    public void WaterTile(Vector3Int position)...

}
```

```
1 reference
public void Tick()
{
    if(container.crops == null)
    {
        return;
    }
    foreach (CropTile cropTile in container.crops)
    {
        if (!IsTilemapAvailable()) { return; }

        //If crop tile is empty/null then move to the next iteration
        if (cropTile.crop == null) { continue; }
        //If the crop tile that holds a seed (/not grown crop) has not been watered
        //then move to the next iteration
        if (!cropTile.watered ) { continue; }
        //If the crop tile has finished growing then move to the next iteration
        if (cropTile.Completed) { continue; }

        //With each Tick (which hapens every 900 sec / 15 ingame min and is handled by DayNightCycle script)
        //the grow timer increments by 1
        cropTile.growTimer += 1;

        MoveGrowthToNextStage(cropTile);
    }
}
//Check availability of tilemaps in case of indoors areas/scenes
5 references
private bool IsTilemapAvailable()...
1 reference
private static void MoveGrowthToNextStage(CropTile cropTile)...
3 references
public void VisualizeTile(CropTile cropTile)...
1 reference
public void PlantSeed(Vector3Int position, CropObject cropToSeed)...
3 references
public bool CheckIfPlantedTile(Vector3Int position)...
1 reference
public void WaterTile(Vector3Int position)...
1 reference
public void PickUpTile(Vector3Int position)...
1 reference
public void ResetTileState(Vector3Int position)...
}
```

Figure 29 - Crops Controller Script

The CropsController script gets the reference of the CropsTilemapController, which handles the functionality of the crop system and the growth of the crops after a certain amount of time (per tick).

The scripts above handle and interact with the tilemap system of Unity. Specifically, the tilemap component is mostly used in 2D game development and allows for storing tile assets, in order to create 2D levels. It transfers the required information from the tiles placed on it to other related components such as the Tilemap Renderer and the Tilemap Collider 2D.

In order for the crop system to function, the related scripts handle the information from the tilemap and identify whether the soil in front of the player is available for planting seeds. Moreover, they allow for storing the location on the tilemap as well as the data of the seeded tile. After that, through these scripts, the player can water the seeded tiles that are selected by the cursor and then pick them up once they are fully grown. In order for the player to pick up the items, a spawn item system was introduced, which instantiates the item object in the spot of the plant. Lastly, the scripts above also handle the growth of each plant by cooperating with the time system, as explained previously.

8.2.2. Player Scripts

The Player gameobject refers to the character that the user sees on the screen. The gameobject holds some necessary scripts for the player movement and controls. Such scripts will be presented in the following paragraphs.

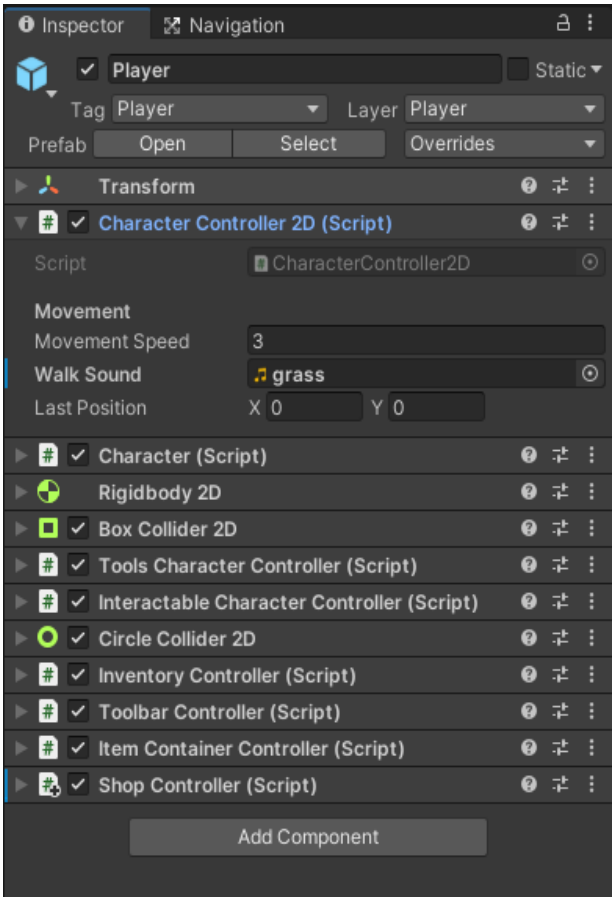
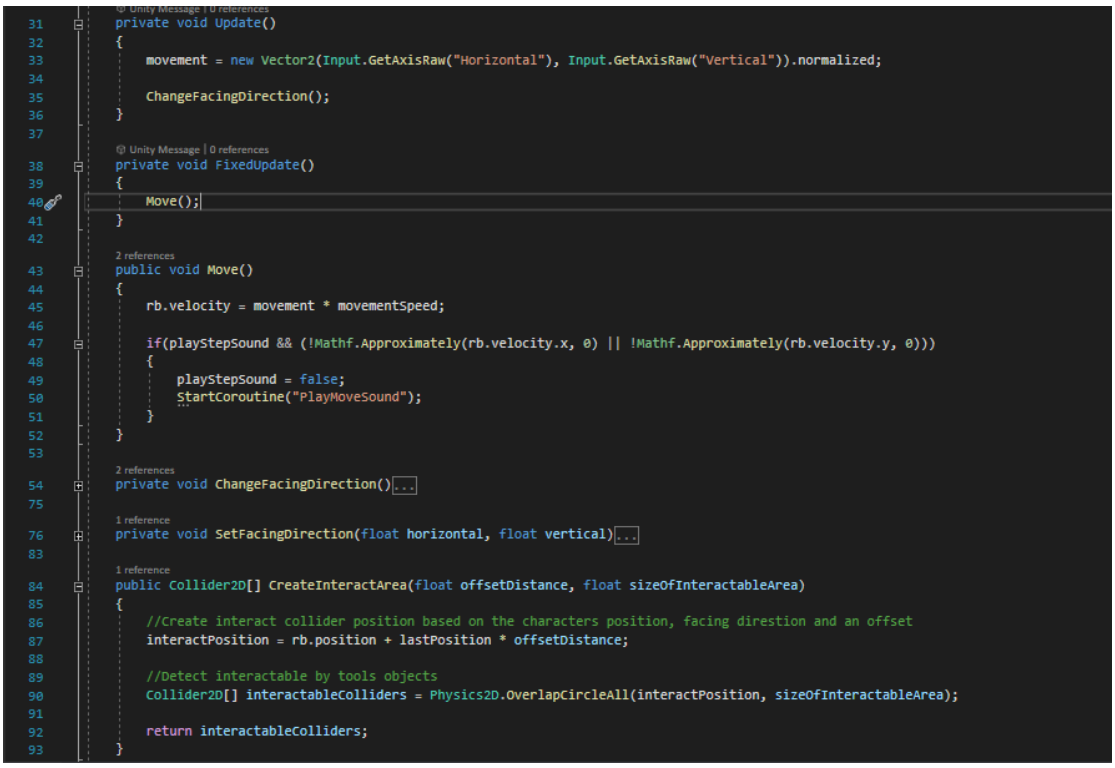


Figure 30 - Player Gameobject

Character Controller 2D Script



```
Unity Message | 0 references
private void Update()
{
    movement = new Vector2(Input.GetAxisRaw("Horizontal"), Input.GetAxisRaw("Vertical")).normalized;

    ChangeFacingDirection();
}

Unity Message | 0 references
private void FixedUpdate()
{
    Move();
}

2 references
public void Move()
{
    rb.velocity = movement * movementSpeed;

    if(playStepSound && (!Mathf.Approximately(rb.velocity.x, 0) || !Mathf.Approximately(rb.velocity.y, 0)))
    {
        playStepSound = false;
        StartCoroutine("PlayMoveSound");
    }
}

2 references
private void ChangeFacingDirection()...

1 reference
private void SetFacingDirection(float horizontal, float vertical)...

1 reference
public Collider2D[] CreateInteractArea(float offsetDistance, float sizeOfInteractableArea)
{
    //Create interact collider position based on the characters position, facing direstion and an offset
    interactPosition = rb.position + lastPosition * offsetDistance;

    //Detect interactable by tools objects
    Collider2D[] interactableColliders = Physics2D.OverlapCircleAll(interactPosition, sizeOfInteractableArea);

    return interactableColliders;
}
```

Figure 31 - Character Controller 2D Script

The script presented above handles the movement of the player character. Specifically, when the user presses the arrow keys, the movement variable registers the input axis. Consequently, the velocity of the rigidbody component on the player gameobject, updates and moves the character towards the axis as indicated by the user's input.

Furthermore, an interact area can be created through the CreateInteractArea function, which creates colliders in the facing direction of the player.

Tools Character Controller Script

```
59   private void GetTileMapReader()...
66
     1 reference
67   private void CanSelectTile()
68   {
69       Vector2 playerPos = transform.position;
70       Vector2 cameraPos = Camera.main.ScreenToWorldPoint(Input.mousePosition);
71       selectableTile = Vector2.Distance(playerPos, cameraPos) < maxDistanceFromPlayer;
72       markerManager.ShowMarker(selectableTile);
73   }
74
     1 reference
75   private void SelectTile()
76   {
77       selectedTilePos = tilemapReadController.GetGridPosition(Input.mousePosition, true);
78   }
79
     1 reference
80   private void PlaceMarker()...
84
     1 reference
85   private bool UseToolWorld()
86   {
87       Vector2 position = rb.position + characterController.lastPosition * offsetDistance;
88
89       ItemObject item = toolbarController.GetItem;
90       //if no tool is selected then try to use empty hand to pick up something
91       if (item == null)
92           return false;
93       if(item.onAction == null)
94           return false;
95
96       animator.SetTrigger(item.AnimationTrigger);
97       bool isCompleted = item.onAction.OnApply(position);
98
99       if (isCompleted == true)
100      {
101          if (item.onItemUsed != null)
102          {
103              item.onItemUsed.OnItemUsed(item, GameManager.Instance.inventory);
104          }
105      }
106
107      return isCompleted;
108  }
109
     1 reference
110  private void UseToolGrid()...
138
```

Figure 32 - Tools Character Controller Script

In order for the player to be able to use tools (e.g. axe, watering can) the respective functionality should be implemented. The ToolsCharacterController script handles the required logic by connecting the item - tool selected by the player from the toolbar with the input required, and calls the onAction function of each tool.

A tool can be used in two different situations, on the tilemap and on the world. In the case of using a tool to interact with the tilemap (e.g. with the crops) then the position of the selected tile is stored and the respective function gets initiated, as explained in detail in the previous section.

However, when a tool needs to be used on the world (e.g. for cutting trees), then the player needs to be able to interact with a gameobject instance, as opposed to tilemap data. In order to be able to translate the position of the pointer to the position of the interactable gameobject on the world the ScreenToWorldPoint function of Unity needs to be used. This function returns the worldspace point created by converting the screen space point at the provided distance z from the camera plane.

Shop Controller Script

```
1 reference
public void Buy(ItemObject item)
{
    if(item.Price > playerData.Money)
    {
        Debug.Log("Not enough money.");
        // play not enough money sound
        AudioManager.Instance.Play(onNotEnoughMoneyAudioClip);
    }
    else
    {
        GameManager.Instance.inventory.AddToInventory(item);
        playerData.Money -= item.Price;
        // play buy sound
        AudioManager.Instance.Play(onBuyAudioClip);
    }
}
1 reference
public void Sell(ItemObject item)
{
    if (!item.CanSell)
    {
        Debug.Log($"Cannot sell {item.Name}.");
    }
    else
    {
        GameManager.Instance.inventory.RemoveFromInventory(item);
        playerData.Money += Mathf.Max(item.Price / 2, 1);
        // play sell sound
        AudioManager.Instance.Play(onSellAudioClip);
    }
}
}
```

Figure 33 - Shop Controller Script

The ShopController script handles the player input regarding the interaction with the ingame shop. When the player interacts with the shop, the UI canvas stops being inactive. Consequently, when the UI is active then, by clicking the respective interaction key or by moving away from the shop, it will become inactive. The same functionality applies to the item containers and the player inventory.

Moreover, when the player clicks on an item from the shop item selection then the respective item gets added to the player's inventory object and money get subtracted from the player's data. The data is a scriptable object related to the player. Additionally, when the player clicks on an item in the inventory, then similar logic applies.

Finally, the sound manager plays the correct sound regarding the player's activities (e.g. buy, sell).

8.3. Behaviour Tree Implementation

The application features the implementation of the behaviour tree architecture for the A.I. functionality of the animal agents.

Such implementation requires certain scripts to be created, namely, a script to execute the behaviour tree, scriptable objects that will hold the parameters of each A.I. agent and their respective controller. Finally, the structure for the behaviour tree needed to be implemented.

Specifically, the required behaviour tree structure has certain nodes which are as follows:

- Composite Nodes
- Decorator Nodes
- Selector Nodes

The visual representation of the A.I. agents' behaviour tree is presented in the appendices below.

8.3.1. A.I. Agent Scripts

In order for the behaviour tree to function, the following scripts needed to be implemented to each A.I. agent.

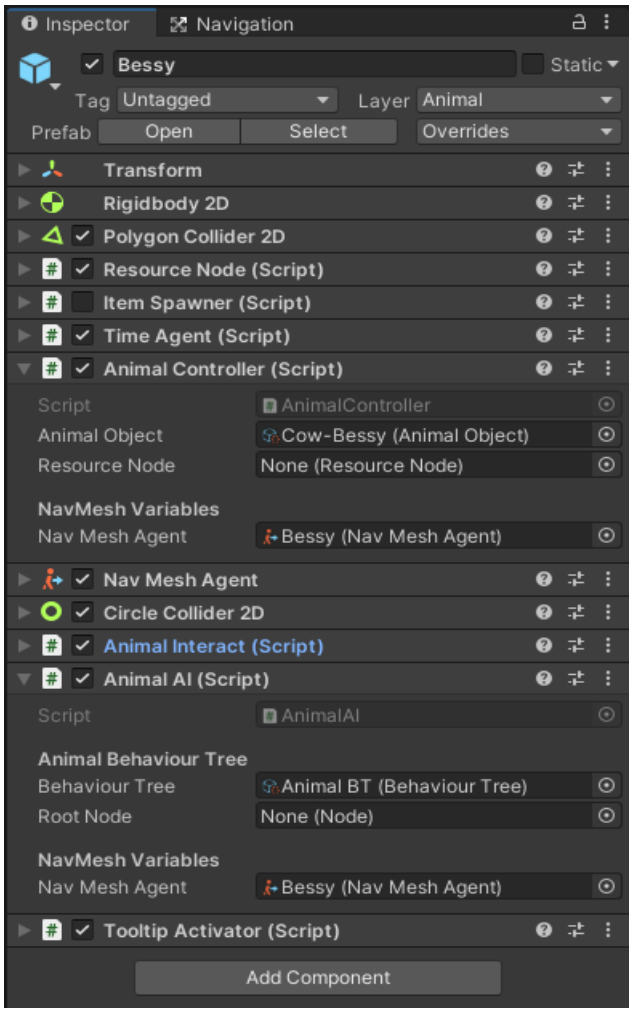
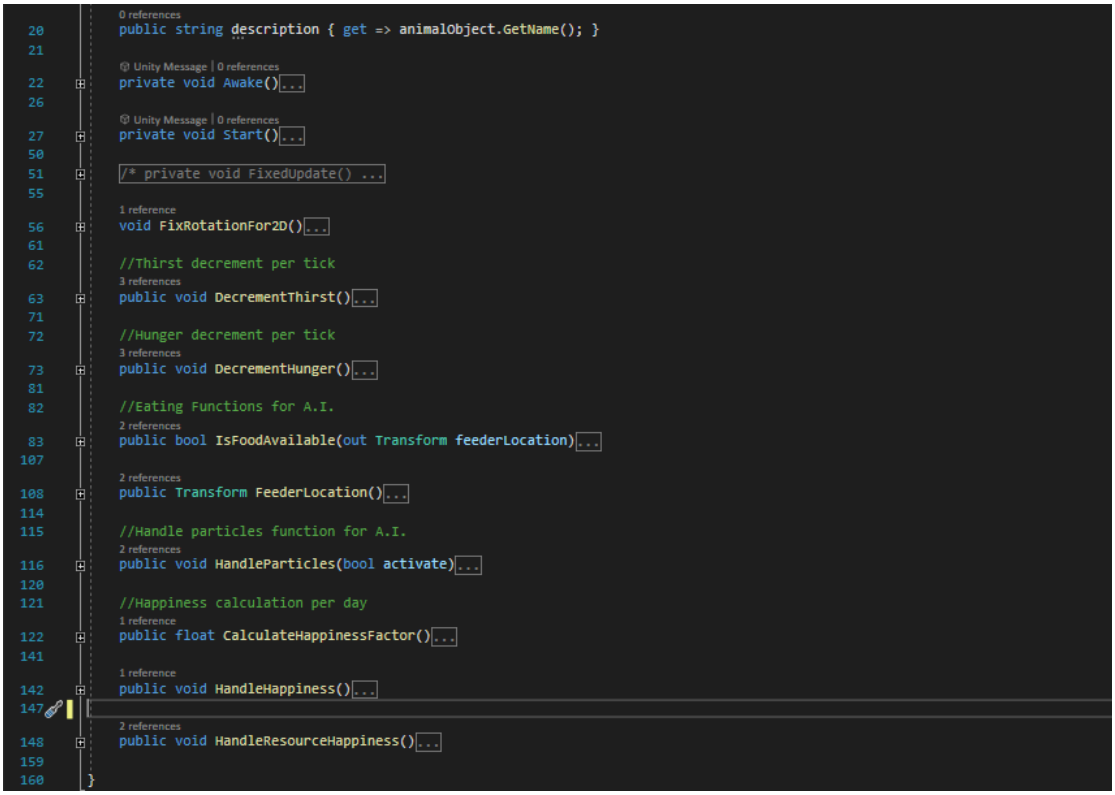


Figure 34 - A.I. Agent Gameobject

Animal Controller Script



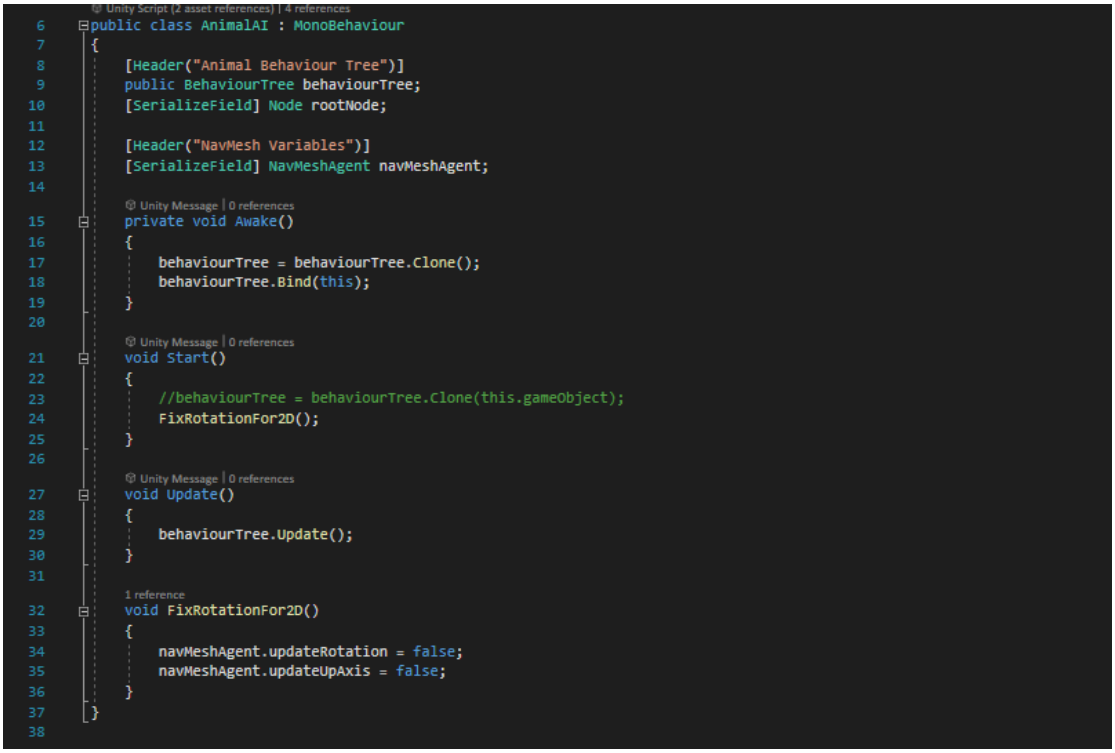
```
0 references
public string description { get => animalObject.GetName(); }

Unity Message | 0 references
private void Awake()...

Unity Message | 0 references
private void Start()...

/* private void FixedUpdate() ...

1 reference
void FixRotationFor2D()...

//Thirst decrement per tick
3 references
public void DecrementThirst()...

//Hunger decrement per tick
3 references
public void DecrementHunger()...

//Eating Functions for A.I.
2 references
public bool IsFoodAvailable(out Transform feederLocation)...

2 references
public Transform FeederLocation()...

//Handle particles function for A.I.
2 references
public void HandleParticles(bool activate)...

//Happiness calculation per day
1 reference
public float CalculateHappinessFactor()...

1 reference
public void HandleHappiness()...

2 references
public void HandleResourceHappiness()...
}
```

Figure 35 - Animal Controller Script

The AnimalController script handles certain functionality regarding the data of each animal. Specifically, each separate A.I. agent has data defined by using the scriptable object functionality, as explained earlier. This data needs to be able to change based on certain criteria, like the passage of ingame time or the interaction with the player (e.g. hunger increments when time passes).

Moreover, certain functions that are needed in the behaviour tree of each agent, are defined in the script above. The navigation nodes mostly need some information from the AnimalController, particularly the availability of food on the map as well as its location.

Animal AI Script



```
Unity Script (2 asset references) | 4 references
public class AnimalAI : MonoBehaviour
{
    [Header("Animal Behaviour Tree")]
    public BehaviourTree behaviourTree;
    [SerializeField] Node rootNode;

    [Header("NavMesh Variables")]
    [SerializeField] NavMeshAgent navMeshAgent;

    Unity Message | 0 references
    private void Awake()
    {
        behaviourTree = behaviourTree.Clone();
        behaviourTree.Bind(this);
    }

    Unity Message | 0 references
    void Start()
    {
        //behaviourTree = behaviourTree.Clone(this.gameObject);
        FixRotationFor2D();
    }

    Unity Message | 0 references
    void Update()
    {
        behaviourTree.Update();
    }

    1 reference
    void FixRotationFor2D()
    {
        navMeshAgent.updateRotation = false;
        navMeshAgent.updateUpAxis = false;
    }
}
```

Figure 36 - Animal AI Script

The script above connects the behaviour tree scriptable object with the agent gameobject (i.e. the animal). In order for the behaviour tree to function correctly, firstly a clone of the original scriptable object is made and then the owner of the tree is passed to the tree by using the function Bind(), which will be explained further in the next section.

8.3.2. Behaviour Tree Scripts

Behaviour Tree Object Script

```
20
21      //Mirrors the update function of the root node
        2 references
22      public Node.NodeState Update()
23      {
24          //When the root node returns something other than RUNNING then it will stop being updated
25          if (rootNode.state == Node.NodeState.RUNNING)
26          {
27              treeState = rootNode.Update();
28          }
29          return treeState;
30      }
31
32  #if UNITY_EDITOR
        2 references
33      public Node CreateNode(System.Type type)...
44
        1 reference
45      public void DeleteNode(Node node)...
51
        1 reference
52      public void AddChild(Node parent, Node child)...
72
        1 reference
73      public void RemoveChild(Node parent, Node child)...
93
94  #endif
95
        2 references
96      public List<Node> GetChildren(Node parent)...
119
120
        3 references
121     public void Traverse(Node node, System.Action<Node> visiter)...
130
131     //Create a clone of the behaviour tree, in order to prevent overlapping trees and permanent SUCCESS states
        2 references
132     public BehaviourTree Clone()...
144
        1 reference
145     public void Bind(AnimalAI owner)
146     {
147         Traverse(rootNode, node =>
148         {
149             node.owner = owner;
150             node.blackboard = blackboard;
151         });
152     }
153 }
```

Figure 37 - Behaviour Tree Object Script

As explained previously, each node of the behaviour tree can return one of three available states, RUNNING in case it is still executing without an end result, SUCCESS in case the node succeeds in doing a certain task or FAILURE in case it fails.

For the behaviour tree scriptable object, the above functions needed to be implemented. The Update() function of the tree calls the Update() function of the root node (i.e. the first node) in the case that it returns RUNNING state.

Moreover, the function Clone() creates a copy of the object in order to prevent duplicate trees and permanent SUCCESS state. Since the BT is a scriptable object, if the same tree tries to get executed twice then the first will run correctly and return a specific state (e.g. SUCCESS) but the second time it will not get executed considering that the state will still be the result of the first. Specifically, scriptable objects maintain the data and then can access it by reference. This means that there is one copy of the data in memory.

Finally, with the Bind() function, each node, as well as the behaviour tree can have access to the gameobject that holds the tree, thus creating a bridge between the A.I. and all the necessary data of each agent.

8.3.3. Composite Node Scripts

Composite Node Base Class

```
Unity Script | 15 references
public abstract class CompositeNode : Node
{
    public List<Node> childrenNodes = new List<Node>();

    5 references
    public override Node Clone()
    {
        CompositeNode node = Instantiate(this);
        node.childrenNodes = childrenNodes.ConvertAll(c => c.Clone());
        return node;
    }
}
```

Figure 38 - Composite Node Base Class Script

A composite node is a node that can have one or more children. Each composite node will process these children nodes, which in turn will either return SUCCESS or FAILURE. After that the composite node will, in most cases, return that state to its parent. During the time the composite node is processing children, it will continue to return RUNNING to the parent. Usually, the most used composite nodes are the sequence node and the selector node.

Sequence Node

```
Unity Script | 0 references
public class SequencerNode : CompositeNode
{
    int currentChild;

    2 references
    protected override void OnStart()
    {
        currentChild = 0;
    }

    2 references
    protected override void OnStop()
    {

    }

    2 references
    protected override NodeState OnUpdate()
    {
        Node childNode = childrenNodes[currentChild];
        switch (childNode.Update())
        {
            case NodeState.RUNNING:
                return NodeState.RUNNING;
            case NodeState.SUCCESS:
                //If a child succeeds, move on to the next child
                currentChild++;
                break;
            case NodeState.FAILURE:
                //As soon as a child fails then all the sequence returns FAILURE
                return NodeState.FAILURE;
        }

        //If the current child is the last one then return SUCCESS differently continue RUNNING
        return currentChild == childrenNodes.Count ? NodeState.SUCCESS : NodeState.RUNNING;
    }
}
```

Figure 39 - Sequence Node Script

A sequence node is derived from the composite node base class and processes each child in a sequence. Specifically, when the child returns a SUCCESS state then the node starts processing the next child, from left to right. In the case that any child returns FAILURE then the whole process stops and the composite node returns FAILURE to its parent node. After every child was processed successfully, the node returns SUCCESS to the parent.

Selector Node

```
Unity Script | 0 references
public class SelectorNode : CompositeNode
{
    int currentChild;

    2 references
    protected override void OnStart()
    {
        currentChild = 0;
    }

    2 references
    protected override void OnStop()
    {

    }

    2 references
    protected override NodeState OnUpdate()
    {
        Node childNode = childrenNodes[currentChild];
        switch (childNode.Update())
        {
            case NodeState.RUNNING:
                return NodeState.RUNNING;
            case NodeState.SUCCESS:
                //As soon as a child succeeds then the selector returns SUCCESS
                return NodeState.SUCCESS;
            case NodeState.FAILURE:
                //If a child fails, move on to the next child
                currentChild++;
                break;
        }

        //If the current child is the last one then return FAILURE differently continue RUNNING
        return currentChild == childrenNodes.Count ? NodeState.FAILURE : NodeState.RUNNING;
    }
}
```

Figure 40 - Selector Node Script

A selector node is derived from the composite node base class and processes child nodes until one succeeds. Specifically, when the child returns a SUCCESS state then the node stops processing the next children, and returns that state to its parent. In the case that the child returns FAILURE then the selector moves on to the next child. After every child was processed and none returned SUCCESS, then the node returns FAILURE to the parent.

8.3.4. Decorator Node Scripts

Decorator Node Base Class

```
Unity Script | 14 references
public abstract class DecoratorNode : Node
{
    public Node childNode;

    5 references
    public override Node Clone()
    {
        DecoratorNode node = Instantiate(this);
        node.childNode = childNode.Clone();
        return node;
    }
}
```

Figure 41 - Decorator Node Base Class Script

A decorator node is a node that can only have one child. Their function is either to transform the result they receive from their child node's status, to terminate the child, or repeat processing of the child, depending on the type of decorator node.

Such decorator nodes, that are oftenly used, are the inverter node and the repeater node.

Inverter Node

```
Unity Script | 0 references
public class InverterNode : DecoratorNode
{
    NodeState _state;

    2 references
    protected override void OnStart()
    {

    }

    2 references
    protected override void OnStop()
    {

    }

    2 references
    protected override NodeState OnUpdate()
    {

        switch (childNode.Update())
        {
            case NodeState.RUNNING:
                _state = NodeState.RUNNING;
                break;
            case NodeState.SUCCESS:
                _state = NodeState.FAILURE;
                break;
            case NodeState.FAILURE:
                _state = NodeState.SUCCESS;
                break;
        }
        return _state;
    }
}
```

Figure 42 - Inverter Node Script

The inverter node gets the result of its child and returns to the parent the opposite state. For example, when the child of the inverter returns SUCCESS then the inverter will pass to its parent FAILURE.

Repeater Node

```
Unity Script | 0 references
public class RepeaterNode : DecoratorNode
{
    2 references
    protected override void OnStart()
    {

    }

    2 references
    protected override void OnStop()
    {

    }

    2 references
    protected override NodeState OnUpdate()
    {
        //The child always returns running and thus this creates a loop
        childNode.Update();
        return NodeState.RUNNING;
    }
}
```

Figure 43 - Repeater Node Script

The repeater node allows for repeating the following nodes of the tree infinitely. Specifically, the repeater will always return RUNNING state to its parent regardless of the child nodes result.

8.3.5. Action Node Scripts

Action Node Base Class

```
Unity Script | 20 references
public abstract class ActionNode : Node
{

}
```

Figure 44 - Action Node Base Class Script

The action node is a node that can have no children and is the leaf of a behaviour tree. Usually, most functionality of an A.I. agent is written in an action node.

These mostly include true or false checks on certain criteria (e.g. check proximity of an agent to an object) or application specific actions (e.g. go eat action on an agent). An important action node that has been implemented in the project is the navigation node.

Navigation Node

```
17   protected override void OnStart()...
23
24   protected override void OnStop()...
28
29   protected override NodeState OnUpdate()
30   {
31       float distance = Vector2.Distance(owner.transform.position, target.transform.position);
32       if (distance > navMeshAgent.stoppingDistance)
33       {
34           Move();
35           HandleAnimation();
36           return NodeState.RUNNING;
37       }
38       else
39       {
40           StopMove();
41           HandleAnimation();
42           return NodeState.SUCCESS;
43       }
44   }
45
46   private void Move()
47   {
48       animator.SetBool("isWalking", true);
49       navMeshAgent.isStopped = false;
50       navMeshAgent.SetDestination(target.transform.position);
51       posX = navMeshAgent.velocity.x;
52       posY = navMeshAgent.velocity.y;
53   }
54
55   private void StopMove()
56   {
57       animator.SetBool("isWalking", false);
58       navMeshAgent.isStopped = true;
59   }
60   public void HandleAnimation()...
67
68   }
69
```

Figure 45 - Navigation Node Script

This node allows the agent to handle the movement to a specific location. When the agent reaches the destination then the node returns SUCCESS to its parent or if it still is computing then the node will return RUNNING.

9. Applying the Behaviour Tree

The A.I. agents' behaviour tree, as is implemented in Unity, will be presented in the section below.

The behaviour tree of an animal agent is divided in 5 main sections, the Sleep Sequence, Thirst Sequence, Hunger Sequence, Happiness Sequence and Wander Sequence which are implemented and occur in that order.

In order for the behaviour tree to be presented in action, the Hunger Sequence is selected as an example.

Specifically, when it is not time for the animal to sleep and the animal is not thirsty yet then the hunger sequence begins.

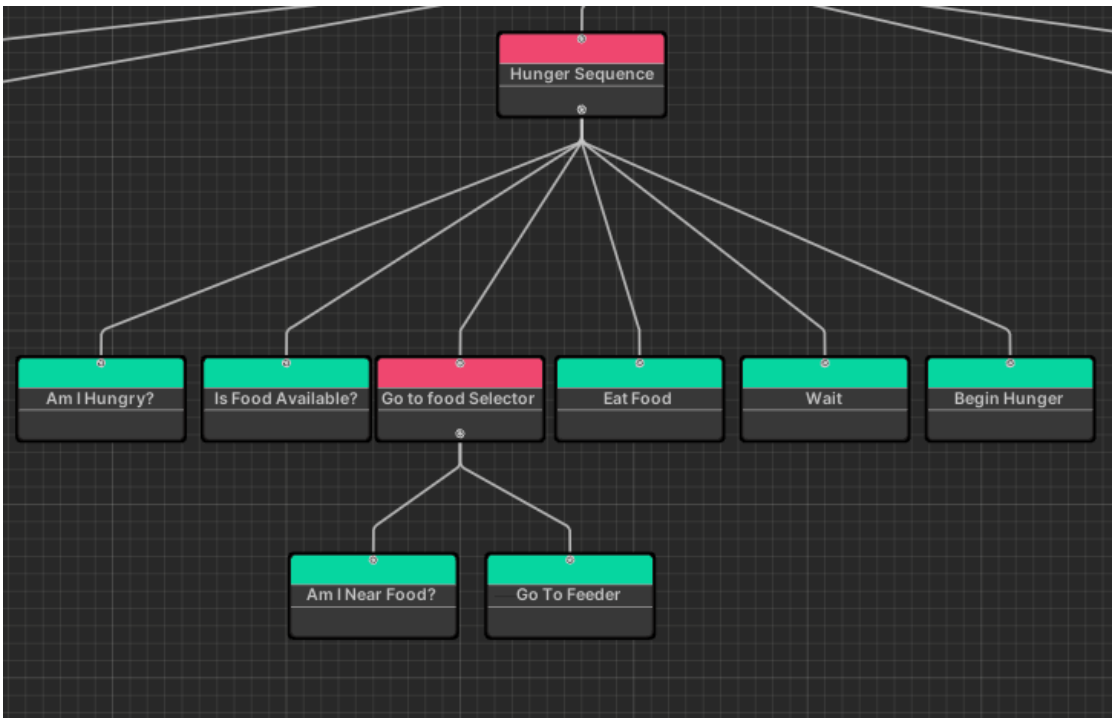


Figure 46 - Behaviour Tree Hunger Sequence

Firstly, the AmIHungry node will be processed which checks whether the animals' hunger variable has fallen below a certain threshold and will return SUCCESS, considering that the parent node is a sequence, this means that the A.I. will move on to the next node which is the IsFoodAvailable. This node checks if food exists on the map, which also returns SUCCESS. Following that, another composite node which is the GoToFood selector node, is implemented. This means that if the first of its children, which is the AmINearFood, returns SUCCESS, then the whole selector will stop executing and pass that state to the hunger sequence. Differently, the GoToFeeder node will also be executed which is a navigation node. After that, the action EatFood is executed which returns SUCCESS after the hunger variable reaches a certain

point of saturation. Finally, the two following nodes, allow for pausing the behaviour tree with the Wait node and after that for initiating the hunger process to start counting down again with the BeginHunger node.

Figure 47 - Smart Animal A.I. Agents

As presented above, from left to right, the first agent being in the Hunger Sequence started eating, the second agent is wandering by reaching the Wander Sequence and the third agent is drinking water through the Thirst Sequence.

10. Conclusion

In conclusion to the project, the most important aspect of this work was the implementation and visualization of the Behaviour Tree architecture. Through this, it is made clear that one of the most efficient ways to handle Artificial Intelligence in a game-like environment is by using the beforementioned architecture. Consequently, by visualizing the A.I. with Behaviour Trees, developers can escape the implementation of multiple if statements and clauses of the FSM architecture which can be reason for mistakes and difficult to fix, errors, if mishandled, and instead focus on structuring correct trees seeing as, after their initial code implementation, are much easier to use, maintain and scale. Another important aspect of this work is the cooperation and interaction of multiple features that were needed in order for the project to have a complete and working system. Such feat was not an easy task, but was made possible through the multiple features that the Unity Engine had to offer.

Finally, the work presented above, can be further developed by implementing more needs of the animal agents, as well as creating new Behaviour Trees for different agents such as possible enemies or friendly characters. Thank you for taking the time to read my Master Thesis.

11. Bibliography & References

1. AI for Game Developers, Glenn Seemann, David M Bourg, 2004
2. AI Game Development. New Riders Publishing, A. J. Champandard, 2004
3. AI Game Programming Wisdom. Charles River Media, Inc., S. Rabin, 2002.
4. AI Planning and Intelligent Agents, Marinagi, Panayiotopoulos & Spyropoulos, 2005.
5. Artificial Intelligence A Modern Approach Second Edition Stuart J. Russell and Peter Norvig, 2003
6. Behavior Trees in Robotics and AI: An Introduction, Michele Colledanchise, Petter Ögren, 2017-08-31
7. Finite State Machines – Brilliant Math & Science Wiki. brilliant.org. Retrieved 2018-04-14.
8. 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"
9. Game AI: The State of the Industry 2000-2001: It's not Just Art, It's Engineering., S. Woodcock, 2001
10. http://www.dogpsychologistoncall.com/hierarchy-of-dog-needs-tm/
11. https://docs.Unity.com
12. https://en.wikipedia.org/
13. https://gamedevbeginner.com/
14. https://poultryhealthtoday.com/maslows-pyramid-self-actualization-for-chickens/
15. https://shubibubi.itch.io/
16. https://towardsdatascience.com/designing-ai-agents-behaviors-with-behavior-trees-b28aa1c3cf8a
17. https://www.gamedeveloper.com/
18. https://www.geeksforgeeks.org/
19. https://www.investopedia.com
20. https://www.oreilly.com/library/view/ai-for-game/0596005555/ch01.html
21. Maslow, A. H., 1943. A theory of human motivation
22. Practical Game AI Programming, Micael DaGraca, 2017
23. Riedl and Young, 2005. An objective character believability evaluation procedure for multi-agent story generation systems
24. The Talk of the Town – It., Grant, Eugene F.; Lardner, Rex, 1952.
25. The World and Mind of Computation and Complexity, Greg Schaffter, 2012
26. Unity Artificial Intelligence Programming: Add powerful, believable, and fun AI entities in your game with the power of Unity, 5th Edition, Dr. Davide Aversa, 03-28-2022
27. Yannakakis, G. N. (2012, May). Game AI revisited
28. Zeltzer, 1992. Autonomy,Interaction,and Presece

Appendix A - A.I. Agent Behaviour Tree Visual Representation

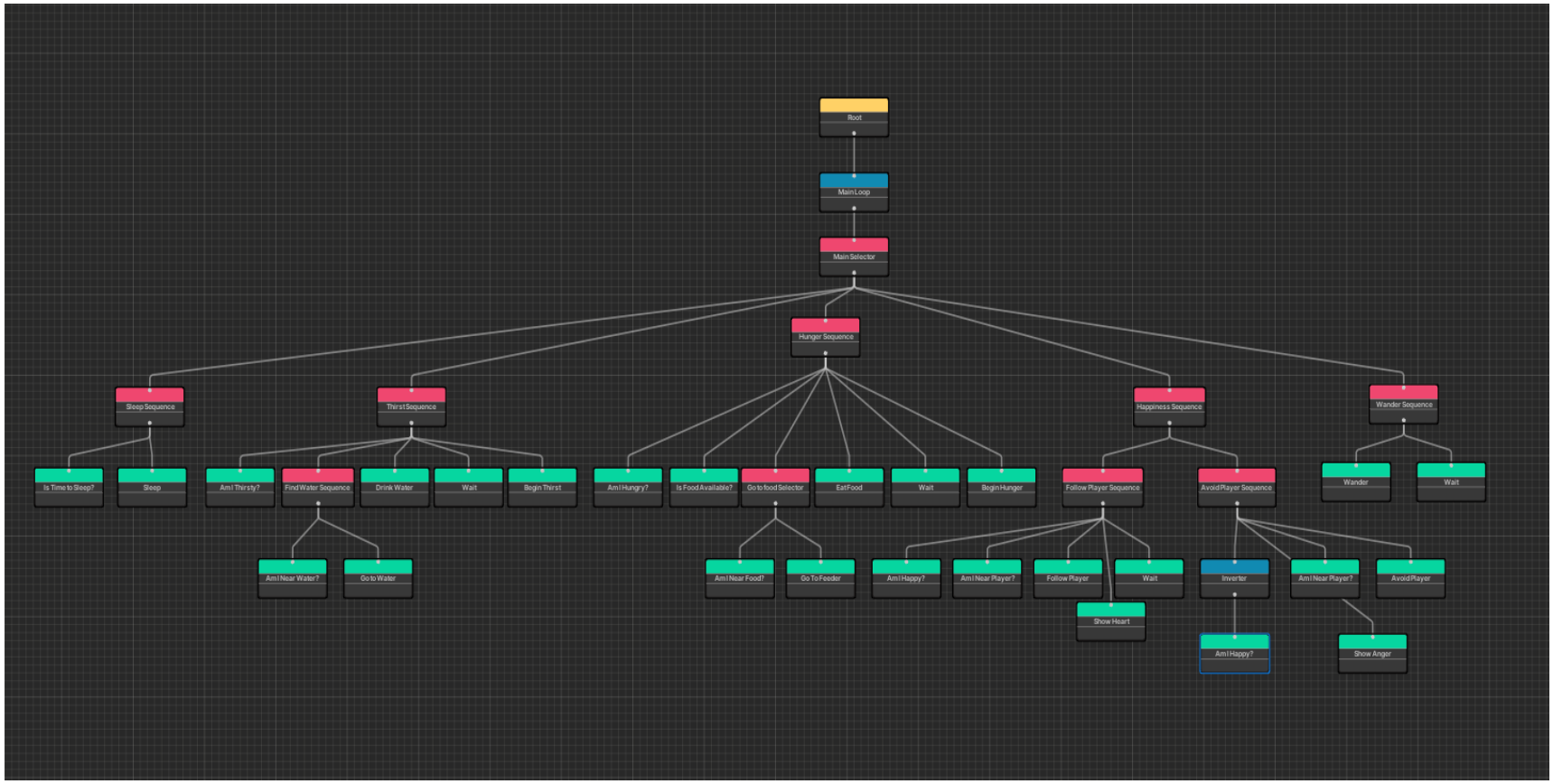


Figure 48 - Animal Behaviour Tree

Δημιουργία προσομοιωτή φάρμας αλληλεπίδρασης
παίκτη-ζώου χρησιμοποιώντας έξυπνους πράκτορες ΤΝ
που χρησιμοποιούν την αρχιτεκτονική του Δέντρου Συμπεριφοράς
στη μηχανή Unity.

Appendix B – Important Information

- All of the non-referenced material written in this work, derive either from the presentations of Dr Panayiotopoulos or the author's ideas.
- Most assets used in the project were created by the author using the Aseprite application.
- Credits for the soundtracks used in the project go to Dimitris Malliaris.
- The version of Unity used for the creation of the project was Unity Engine 2020.3.2f1.

Appendix C – Project Inspiration

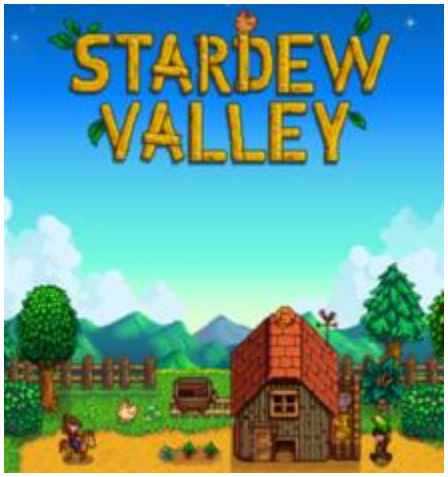


Figure 49 - Stardew Valley Logo

Stardew Valley is a simulation role-playing video game developed by Eric "ConcernedApe" Barone. Players take the role of a character who takes over their deceased grandfather's dilapidated farm in a place known as Stardew Valley. The game was released for Microsoft Windows in February 2016 before being ported to several other computer, console, and mobile platforms.

Stardew Valley is open-ended, allowing players to take on activities such as growing crops, raising livestock, mining and foraging, selling produce, and socializing with the townspeople, including the ability to marry and have children. It also allows up to three other players to play online together.

Barone developed Stardew Valley by himself over four years. He was heavily inspired by the Harvest Moon series, with additions to address some of the shortcomings of those games. He used it as an exercise to improve his own programming and game design skills. British studio Chucklefish approached Barone midway through development with the offer to publish the game, allowing him to focus more on completing it.