Project 400

Behaviour Trees
VS
GOAP

Author: Bartlomiej Bialowas - S00189532

Supervisor: Neil Gannon

BSc (Honours) in Computing in Software Development (Add-On)

Project Repo: https://github.com/s00189532/Project400-Unity

Note The project was created using Unity version 2021.1.16f1 and coded using Visual Studio 2019

Table of Contents

Table of Figures

1. Overview

This report discusses both the research and creation of videogame A.I. (artificial intelligence) NPCs (Non-Player characters) through the use of two different A.I creation methods, the former being a BT (Behaviour Tree) and the latter being GOAP (Goal-Oriented Action Planning).

The report will go over the research done on GOAP and Behaviour Trees. It will also discuss how the Behaviour Tree and GOAP systems were implemented into the Unity game Engine. Then it will show how those systems were utilised alongside tools built into the Unity game engine and custom scripts to create the NPCs, their logic and everything else in the scenarios the NPCs need to complete their logic.

The goal of this project is to use a game engine in conjunction with these two A.I. creation methods to create two identical scenarios which contain NPCs performing specific actions. These scenarios can then be used to get a better comparison between the two A.I. creation methods by looking at how the NPCs behave and by tracking and documenting the creation process of these NPCs.

2. Literature Review

2.1. Introduction

The aim of this literature review is to examine BTs and GOAP more thoroughly and how they are used in the creation of videogame NPCs. It will discuss how they work, look at their strengths and weakness and compare the two against each other.

2.2. Behaviour Tree (BT) System

2.2.1. Introduction to Behaviour Trees

A BT is a plan of execution of tasks that was made to shape the behaviour of A.I. NPCs in video games. While it was initially developed for the video game industry it has also seen use in other fields such as robotics and computer science. It has been used in many big games such as Halo and Bioshock. One of the highest profile implementations of BTs currently has been built into Unreal Engine (UE) which is one of the biggest game development engines in the world and this is the implementation that will be referred to a lot throughout this review. A BT is comprised of two components. The first is a graphical representation of a hierarchy of different nodes. These nodes store the logic of the behaviour tree and are connected in a tree like structure, hence the name Behaviour Tree. The second component is a data structure called a Blackboard which is used to store important information the NPC might need to use in its behaviour tree such as player location, health, etc. The tree executes its logic going through the nodes from left-to-right and from top-to-bottom.

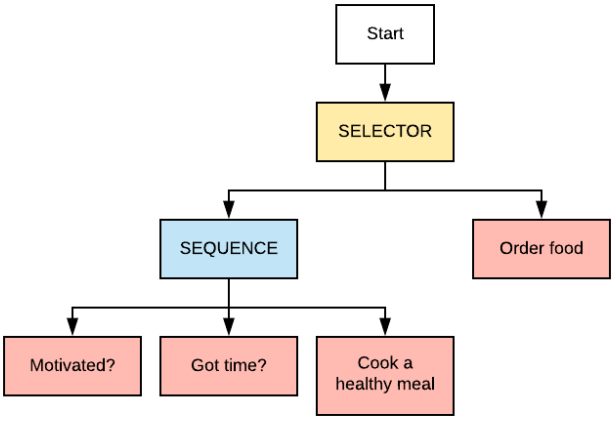

Figure 1: Behaviour Tree Example (Haytam, 2020)

BTs are very modular which allow for easy change in the design of AI behaviour in turn allowing the developers to make the NPCs work and compliment the designers works. It allows NPCs to be manually tweaked to fit an area specifically to get the best result out of that area and to make use of the weapons or environment to give the best experience to the user.

2.2.2. Nodes

The nodes are split in to three main types, the root node, control flow nodes and execution nodes.

Root Node - The root node also sometimes referred to as the start node is a unique node with only one being able to exist in a behaviour tree. It is used as the starting point of a behaviour tree and it can only have one node connected to it.

Control Flow Nodes - These nodes control the child nodes under them that they are connected to. There is no limit to how many of these nodes there can be and they can be connected to other control flow nodes. They can decide whether and how to run the child nodes under them based on different conditions such as does the NPC have a weapon and they return a success or failure based on the result of those nodes. There are four main types of control flow nodes but as different behaviour tree implementations are being made and behaviour trees evolve more are being made and the existing ones can change based on which implementation you might look at.

Sequence Control Flow Node - Can have an unlimited amount of child nodes and returns success if all its child nodes return success otherwise returns failure if any child node returns failure.

Fallback/Selector Control Flow Node - Fallback nodes run their child nodes until they find a child node that returns success then they return success otherwise if all child nodes fail the fallback nodes return failure.

Parallel Control Flow Node – This node can only have two child nodes and will run them both constantly at the same time and returns success if one or more of its child nodes returns success.

Decorator Control Flow Nodes - Can only have one child node. Works as a conditional node deciding whether its child nodes should be run or whether the tree should continue. It has multiple versions that control then how it runs the nodes and how it handles the return value. Some newer versions exist of decorators such as UEs version which turns the decorator node from a node to a condition that you can add to other control flow nodes to control when they should run.

Execution Nodes - Execution nodes also known as task nodes or action nodes carry the main logic of the behaviour tree. They cannot have any child nodes connected to them and they run custom code to execute the specific actions the NPC needs to perform.

2.2.3. Blackboard

While blackboards are not necessary most implementations of BTs have them as BTs nowadays and the AI made through them tend to be very complex and blackboards allow for easy storing and sharing of data between the nodes in the tree and each NPC using that BT. At its simplest form the blackboard is a key-value storage system which allows you to store values and assign a name to them such as storing an integer called health which is then used to call them in the BT when required. These values can be easily used and edited by the nodes of the BT and each NPC using that BT.

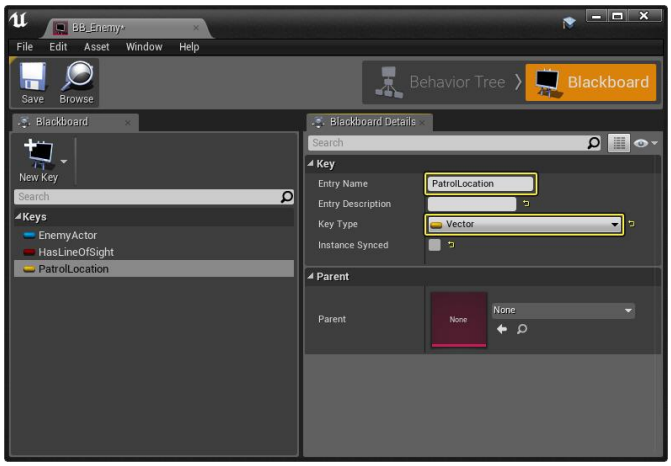

Figure 2: Unreal Engine Blackboard example (Unreal Engine, 2020)

2.3. Goal-Oriented Action Planning (GOAP)

GOAP is a decision-making architecture that takes the next step, and allows characters to decide not only what to do, but how to do it (Orkin, 2002). It is a STRIPS (Stanford Research Institute Problem Solver) architecture used to design the behaviour of NPCs in video games originally created by Jeff Orkin which was first implemented in the game F.E.A.R. (First Encounter Assault Recon) which was published on October 17, 2005.

GOAP allows the NPC to make plans for itself allowing it to adapt to any situation and area it is put in without its behaviour needing to be modified. This makes the NPCs very unpredictable and thus making the game less repetitive and more enjoyable for longer and subsequent playthroughs of it. This also reduces the potential workload on larger games where the NPC would have to be modified to fit each area correctly.

At its base GOAP works like a BT transitioning from one node to another. However, while the BT has all the nodes in a hierarchical structure and defined plan of execution for the tree GOAP instead has a scattering of unconnected nodes containing logic for the actions the NPC can perform and has a cost assigned to them. In comparison the initial implementation of GOAP in F.E.A.R. has a Finite-State Machine (FSM) (which is essentially a BT but with no structure allowing all nodes to be connected to each other) with only three nodes, a Go to, an Animate and Use Smart Object nodes as a base for the NPC used to control the state of the NPC based on the node being run by GOAP with the Use Smart Object node being used to play specific animations essentially working like the Animate node. Jeff Orkin in his paper on F.E.A.R. talks about this stating “As much as we like to pat

ourselves on the back, and talk about how smart our A.I. are, the reality is that all A.I. ever do is move around and play animations! Think about it. An A.I. going for cover is just moving to some position, and then playing a duck or lean animation. An A.I. attacking just loops a firing animation". He says all NPCs do is move and animate which means that you do not need a complex FSM and you just need to pass it the correct animations and values for it to work.

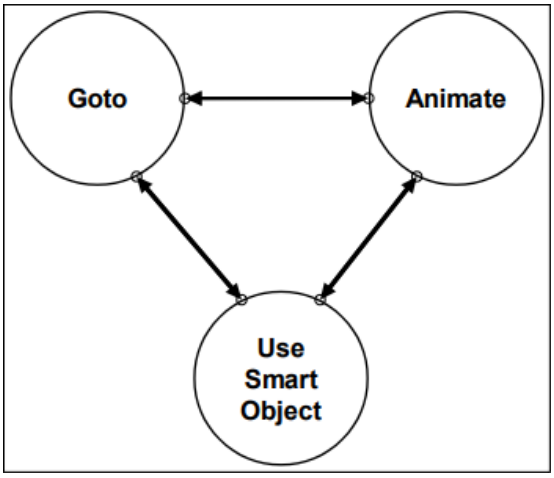

Figure 3: F.E.A.R. GOAP FSM (Orkin, Three States and a Plan: The A.I. of F.E.A.R., 2006)

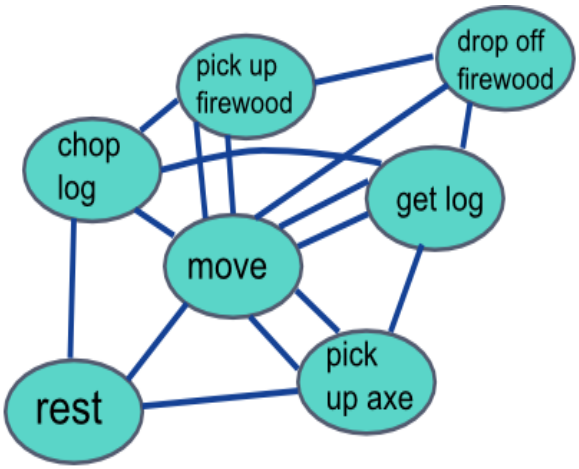

Figure 4: regular FSM example (Owens, 2014)

This is where GOAP differs from BTs. While BTs add more nodes, transitions, and conditions for said transitions GOAP instead make use of a planner, a set of goals, a set of actions and the A* search algorithm. When BTs get more nodes and the tree grows you have to ensure all nodes can transition to each other without issues and wont impact other nodes which makes the tree increasingly more complex to manage as it grows, while with GOAP you can get rid of the transitions between these nodes altogether. GOAP instead just needs a set of nodes for the actions an NPC needs to perform with costs assigned to them and a set of nodes for the goals an NPC can fulfil. NPCs in the game can then be assigned specific sets of actions and goals that they can do such as walk and shoot. The NPC can make use of the actions in a specific chain to complete the goals but with the goals and actions being unconnected they need to be told which to do. For this the planner and search algorithm are used. First an NPC needs a goal to work towards. Goals usually have a priority assigned to them in the form of a value like an integer or float which get edited based on the state of the world such as a shoot player goal rising if a hostile NPC spots a player. The goal with the highest priority will be then assigned to the NPC. For example, in F.E.A.R. the patrol goal will have a higher priority than the

attack enemy goal if the player is not spotted but the if an NPC sees a player then the priority of attack enemy will start rising until it is above patrol at which point it will switch goal to the attack enemy goal and the patrol goal priority will drop significantly.

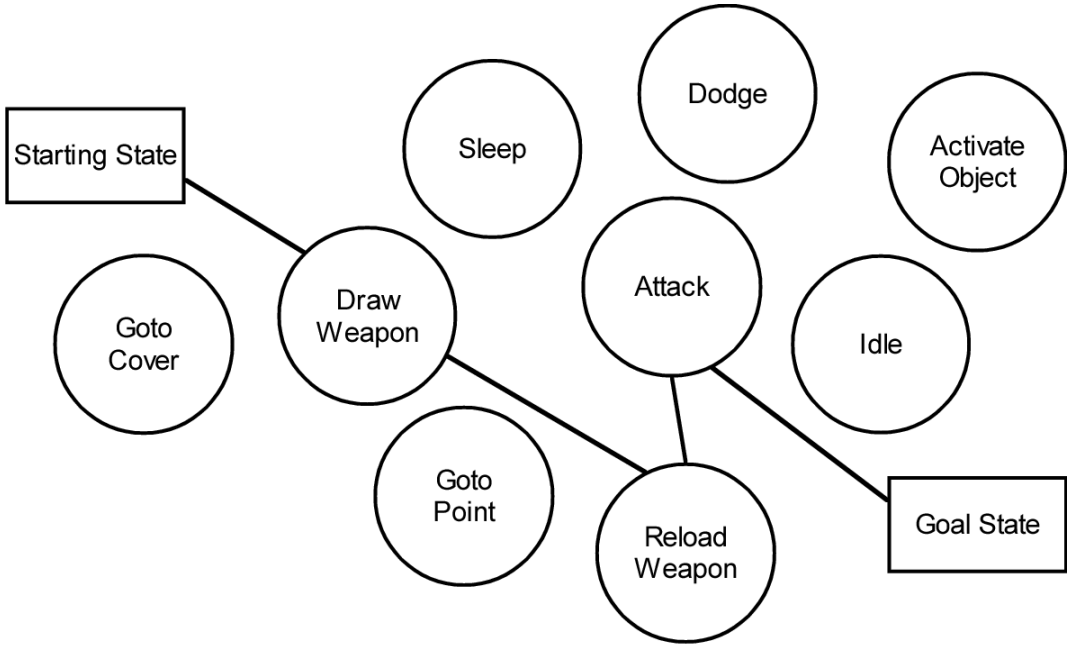

Figure 5: GOAP planning process visualized (Orkin, Applying Goal-Oriented Action Planning to Games, 2002)

Once an NPC has a goal to work towards it needs to formulate a plan to complete the goal. This is where the planner will come in to play. The planner will look at the list of actions that the NPC can perform such as shoot, move, throw grenade, etc, and it will use their cost values with the A* search algorithm to create a chain of actions to perform the goal with the lowest cost possible. So, for example if an NPC gets assigned a goal to move to a location. The planner will gather the actions that NPC can make along with their costs. It will check if the location and path to the location is clear then it will plan to move there but say there is a door in the way than it will need to plan to move to the door than open the door and then move to the location again but what if again now there was another path without a door blocking. This is where the scoring system comes in and the A* search algorithm. The path will be scored based on the actions which in this case will be distance to travel and opening the door. Then the algorithm will go through all choices and pick the least costly one to perform. Then once all actions are picked they get assigned in the correct order working from the end of the goal to the start giving the NPC the plan they need to fulfil their goal.

Once the plan is made each action is gone through one by one through the FSM allowing the NPC to move or play an animation such as opening a door and interacting with that object.

GOAP just like BTs is very modular with how it is made. If the NPCs in the game ever get new content allowing them to perform new actions all that needs to be added for the NPC to work is new actions and goals to their set along with the animation that action performs and cost for the actions for the A* algorithm to use. Even then this is only required for unique actions as if you were to just add new weapons to the game than all that needs to change is the animation assigned to the shooting action when the NPC has that weapon.

GOAP can also be used very easily between all NPCs in the game as the actions are unconnected they allow all the NPC to just use the same planner with their set of actions. For example, in F.E.A.R.

there is mice that use the same planner as regular combat enemies except the set of actions and goals they use only require them to move.

2.4. Comparison

2.4.1. Advantages of BTs over GOAP

- **Low Complexity –** Easy to start up and use especially with big game engines like Unreal Engine having high-quality built-in implementations that are free to use. Many Blueprints implementations also use visual nodes making them easy to look at and understand with Unreal Engines version being able to be done with their visual scripting so you do not have to even touch code. GOAP on the other hand is more complex to set up and only really benefits the game when the NPC start getting more complex making management of a BT more difficult and if you want the NPC to be more unpredictable making it difficult to set up and not worth using for most smaller games.
- **More Control Over Design of NPCs –** GOAP AI makes the NPCs be able to plan out their own moves and be very unpredictable which while it has its advantages it also makes it so that the NPC may not make full use of the design and environment of the game making those parts feel pointless while with BTs you have full control over what the NPC does and they can be setup to compliment the design of the game and allow the designers to craft the most enjoyable experience possible.
- **Easier Debugging –** GOAP unlike BTs is a lot more difficult to debug because of how unpredictable it is and because of everything being unconnected while BT debugging is a lot easier because of its lower complexity and control the developer has.
- **Smaller Computational Overhead –** GOAP is a lot more computationally expensive as while in BTs you have a defined set of transitions to check and follow, GOAP on the other hand must make plans for every NPC constantly checking over all the possible actions for each NPC and calculating the best one and whenever there is a change to the goal they all must be recalculated again. Nowadays however this is a minor issue with how powerful computers and consoles are nowadays.

2.4.2. Advantages of GOAP over BTs

- **Easy Scaling –** While GOAP is more difficult to implement and start up for bigger games it can be used to scale the game easier as when big games are made and have more content added BTs need a lot of reworking for all the transitions to work with each other correctly while with GOAP you would need to just create the appropriate goals and actions with their costs and priorities assigned and the planner will handle it making GOAP work with new content a lot easier because of its adaptability.
- **Adaptability –** GOAP allows NPCs made with it to adapt to most situations. The AI can plan for most changes made and any new areas added to the game that do not make use of new mechanics and can make effective use of the area without needing any changes while BTs will require a lot of extra work for the AI to make use of the new area or mechanics properly. This also allows for user generated content to work with the game better as the user will not have to edit the AI for it to work with what they make and allow them to make more creative content with less restraint.
- **Replay Ability –** The unpredictability of the NPCs as they make their plan and player interaction changing that plan in an infinite number of ways leads to the game being infinitely more replay able than as it would be mathematically impossible to get the same encounter twice while NPCs made with BTs can only have so many ways they can act especially when designed to take the map into account. This make games developed with

GOAP able to be played many years and playthroughs down the line and still lead to interesting encounters and can still challenge the player.

2.5. Literature Review Conclusion

In conclusion both methods have their strengths however BTs are more common and have become the industry standard because of their modular design, ease of use and low complexity GOAP is a lot more complex, more difficult to use and start up making it a worse choice for small games but very good for giant games where the NPC gets very complex and well-done implementations can make a game endlessly replay able. So, for most games BTs are usually the best option and allow for very creative and designer friendly creation of NPCs but bigger games that can make proper use of GOAP can make for timeless and endlessly replay able games.

Since each method has its own strengths, weaknesses and there is no clear comparison between the methods as the games made using these methods are drastically different and therefore cannot be used to give a clear comparison between, the two methods making the answer as to which is better not clear which is the reason for this project. The project aims to get more clear information by creating a small game and implementing two identical versions of AI NPCs into the game with the use of BTS and GOAP and tracking the process to try to get a better first-hand comparison of the two AI creation methods.

3. Design

3.1. Scenario Design

This project will contain two scenarios which as mentioned before will be identical with the exception that the NPCs will be made with two different A.I. creation methods but will still ultimately have the same goals to complete. The scenarios themselves will be pretty simple. Each scenario will be a Scene object in the Unity game engine which is essentially a sandbox to put your work in. The scenarios themselves will be simulating a game where there is going to be three NPC workers that will be gathering/storing resources and working based on the type of worker they are and the tool they have.

The initial scenes will be very simplistic in terms of design with all the assets being made in the engine with simple shapes and materials and being static with no animations and no audio. The intention is to replace them with proper assets and animations acquired online once the project is done if there is enough time.

3.1.1 Workers

As mentioned there are three workers. These workers are the Miner, the Lumberjack, and the Blacksmith. Each one has a tool with limited durability except the blacksmith which has unlimited durability. Each one will perform their job on an endless loop based on their logic made by the BTs and GOAP. The NPCs themself initially will start out as just a capsule with simple 3D tools attached to their side and a cube protruding from the higher half of the capsule to show which side of the NPC is the front.

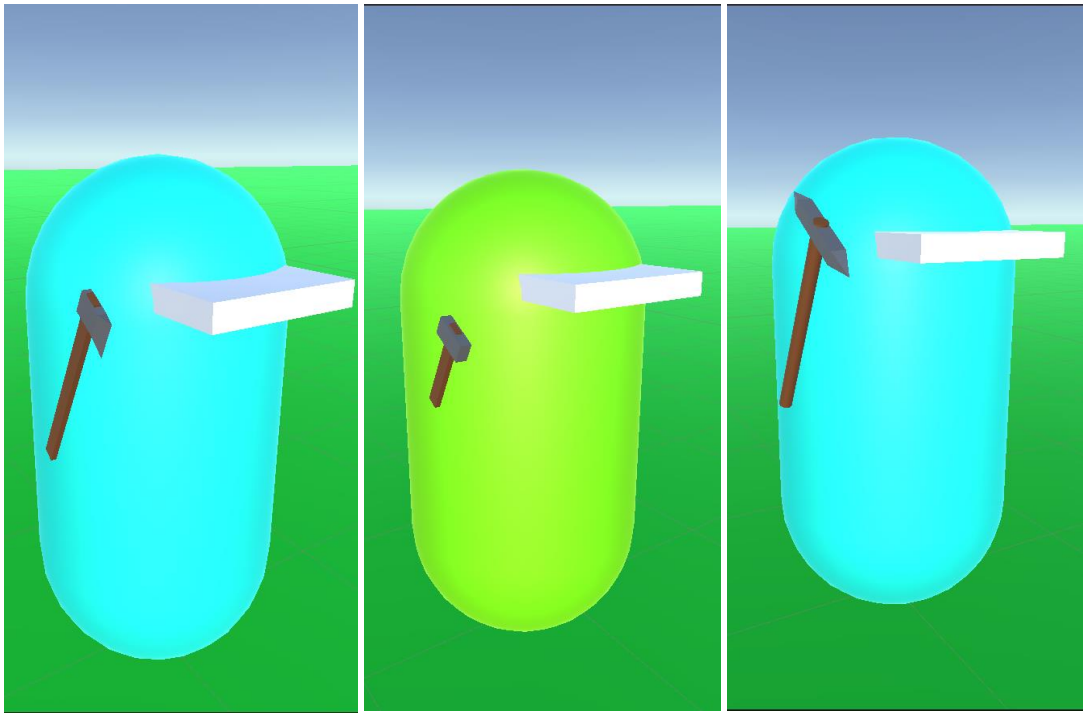

Figure 6: Lumberjack, Blacksmith, and Miner NPCs in that order

Figure 7: The tools used by the NPCs in the project

Miner/Lumberjack - The Miner and the Lumberjack are going to work very similarly. They will gather iron and wood until their tool runs out of durability at which point they will go to store the resources they gathered at a storage point. At that point they will go to their corresponding tool station and either take a new tool from the station if there is one available or wait at the station until the blacksmith creates a new tool and deposits it at the station for them.

Blacksmith – The blacksmith differs from the other two workers. Unlike the other workers the blacksmiths tool has no durability and he does not harvest any resources. Instead, the blacksmith job is to keep track of the tool stations and make sure there is a tool on them at all times for the other workers to pick up when theirs break. The blacksmith will first go to the storage if there are any resources in there and collect the resources he needs to craft a new tool. Then he will return to his anvil and wait until one of the stations is missing a tool at which point he will use up the resources he gathered at the anvil to make a new tool and go to deposit it at the station.

3.1.2 Scene Assets

The scenes themselves will contain multiple simple structures that the NPCs will make use of to complete their jobs. In the end the scenes consist of a floor, a resource storage structure, tool stations, iron deposits, trees, and an anvil.

Floor - The floor is currently a flat plane that is coloured green to represent grass with the intent to switch it to actual 3D animated grass and dirt patches.

Figure 8: Floor plane in the Unity project scene

Resource Storage - The resource storage contains and iron and a wood resource deposit with each being surrounded by small walls which is used by the Lumberjack and Miner to store resources in and by the Blacksmith to take resources from to make new tools. It is currently made of simple cubes with the intent once again being to upgrade to proper assets if possible.

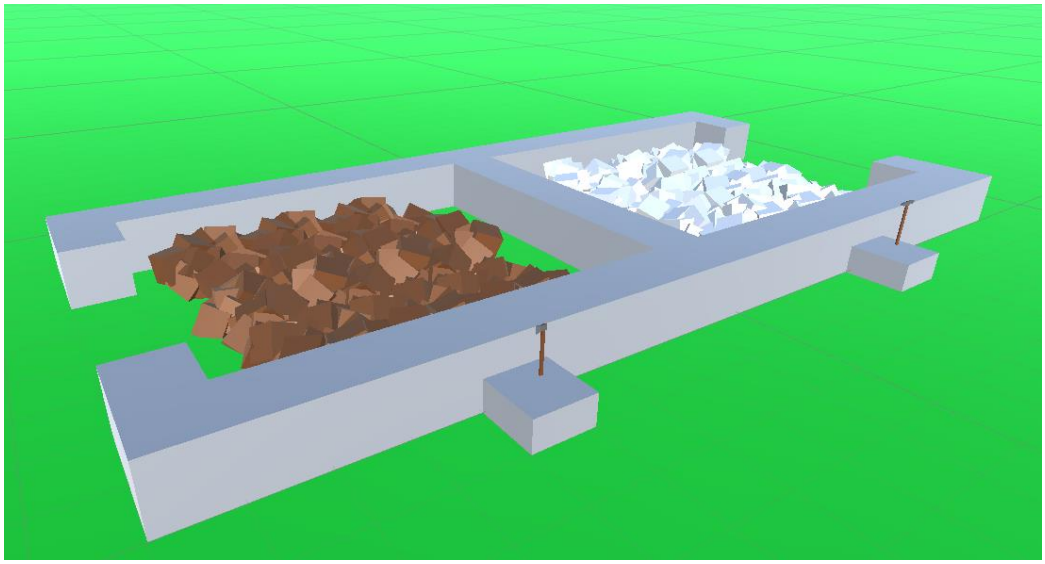

Figure 9: Storage used by the NPCs in the project with a wood deposit on the left and iron deposit on the right

Tool Stations - Each resource storage will have two tool stations attached to it, one by each resource deposit which will hold a corresponding tool based on the deposit for the Lumberjack and the Miner to pick up when their tool breaks. Another tool can then be placed on it by a Blacksmith. Currently it is a half slab with the tool floating above it.

Figure 10: Tool stations with tools placed on them

Iron Deposit - The level will contain iron deposits for the Miner NPC to able to gather iron from it if they have a functioning pickaxe on them. The iron deposit is currently depicted by a set of cubes rotated in different positions.

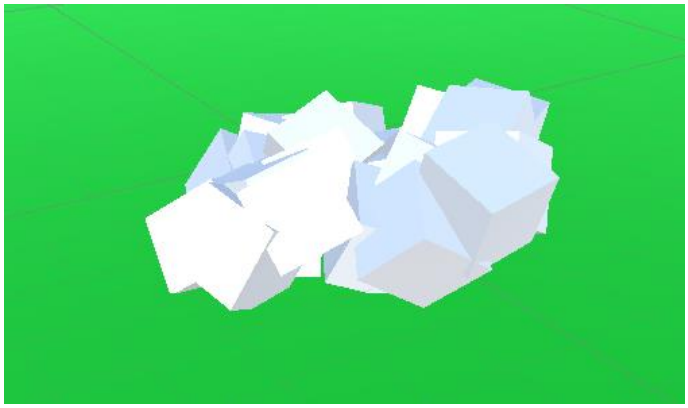

Figure 11: Iron Deposit used by Miner NPC to harvest iron

Trees - There are going to be trees located in the level that the Lumberjack NPC can go to and gather wood from if they have an axe. The tree is currently comprised of a cylinder mesh as the wood base and a set of cones used as the leaves.

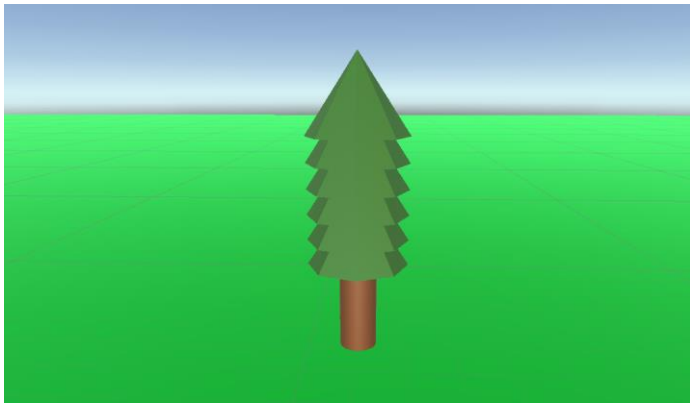

Figure 12: Tree used by Lumberjack NPC to harvest wood

Anvil - There is an anvil in the world that the Blacksmith NPC can use to make the tools that it deposits at the tool stations for the other worker NPCs. It is currently represented by a black rectangle.

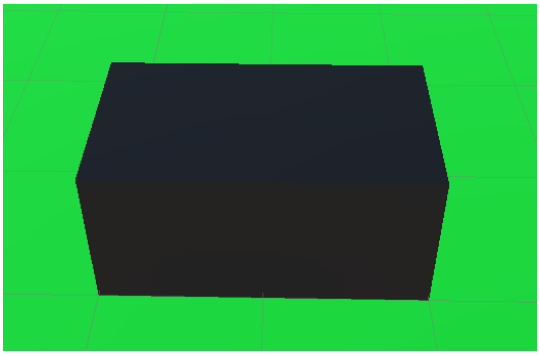

Figure 13: Anvil used by the Blacksmith NPC to craft tools

4. Behaviour Tree Scenario Implementation

4.1. Behaviour Tree System Setup

Unlike Unreal Engine, the Unity game engine does have not a BT system built into it. Because of a custom BT system had to be built for this project since it was done in Unity. Being that BTs are so popular and that the Unity Engine has a giant community, has a lot of documentation, and uses C# as its coding language which is a lot easier to use than Unreal Engines C++ made it quite easy to research and find helpful documentation and tutorials on how to make a BT system in the Unity Engine.

In the end after going through many tutorials and a lot of documentation the BT system for the project was made through the use of Unity documentation and following two YouTube tutorials on the channel TheKiwiCoder and editing that system afterwards a bit to fit the needs of the project better. TheKiwiCoder has two YouTube tutorials showing how to set up a great BT system but what set his tutorials apart from the others was that he made use of a pretty new Unity Engine tool called UI Builder which allows the user to create custom UI in the editor of the engine. This made it possible to make a good, easy to use and customizable visual representation of the BTs and connect it with the code of the nodes which made it a lot easier to connect the nodes and make the BTs. Also, unlike other tutorials he made blackboard that works with the behaviour trees which was used throughout the project to store the position the NPCs had to move to. The two videos and the channel are named below along with the YouTube link to them.

- Unity | Create Behaviour Trees using UI Builder, GraphView, and Scriptable Objects https://youtu.be/nKpM98I7PeM
- Behaviour Tree Editor with UI Builder – Part 2 https://youtu.be/jhB_GFgS6S0
- TheKiwiCoder YouTube channel https://www.youtube.com/channel/UCjszZMwnOW4fO5VIDU_Wh1Q

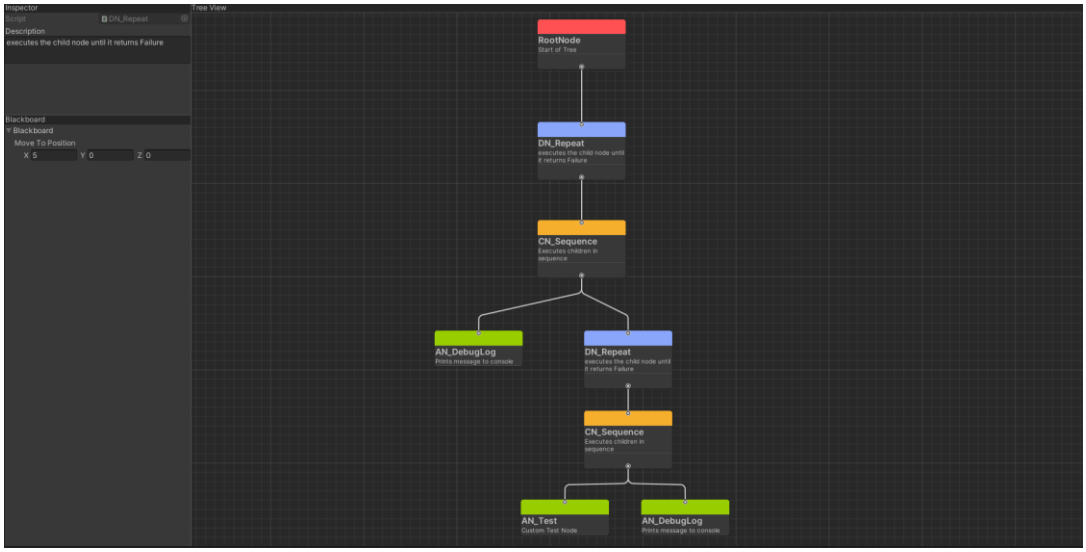

Figure 14: Example Behaviour Tree made inside the custom-built Behaviour Tree Editor

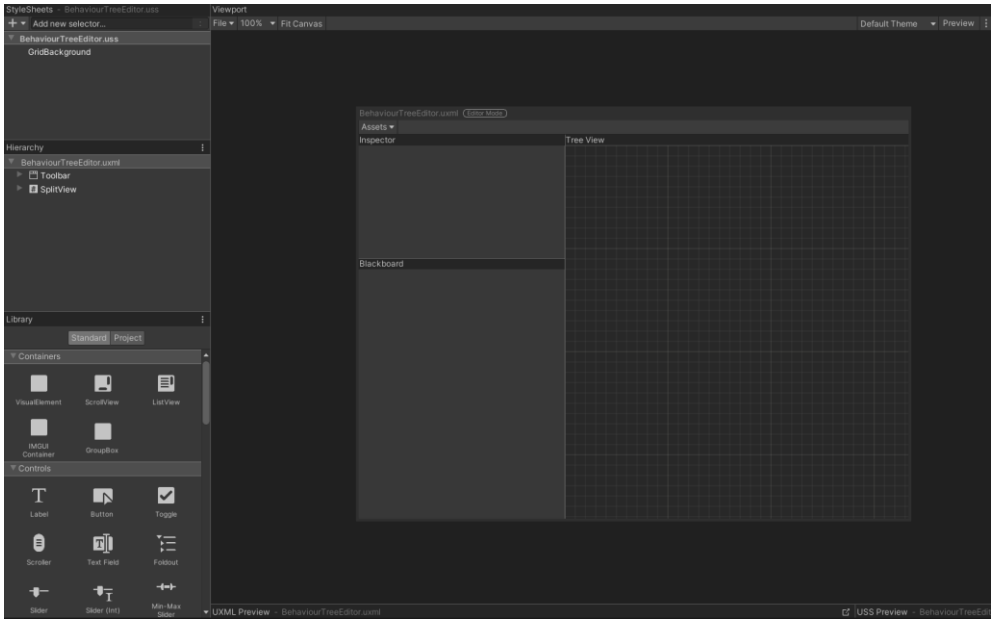

Figure 15: Behaviour Tree Editor UI within UI Builder

The Behaviour Tree Editor itself consist of five parts. The tree view, the nodes, the inspector, the blackboard, and the logic that allows the user to create, view, connect and edit the Behaviour Tree and nodes within it.

4.1.1. Tree View

The Tree View uses special Unity UI code called GraphView to create the graphed section of the Behaviour Tree editor as seen in the figures above and below where the nodes can be created, edited, moved around, and connected together alongside with custom code tweaking how it works. This code within the Tree View allows the user to create new nodes for a Behaviour Tree when right clicking within the Tree View, select nodes highlighting them, undo actions, redo actions, scroll in and out of the view to zoom in on the Tree View and it also handles the positional code of the nodes to make sure the nodes get executed in order from left to right. The UI Builder tool was then used to add the Tree View to the Behaviour Tree Editor and to handle its positioning and scaling within the Behaviour Tree Editor.

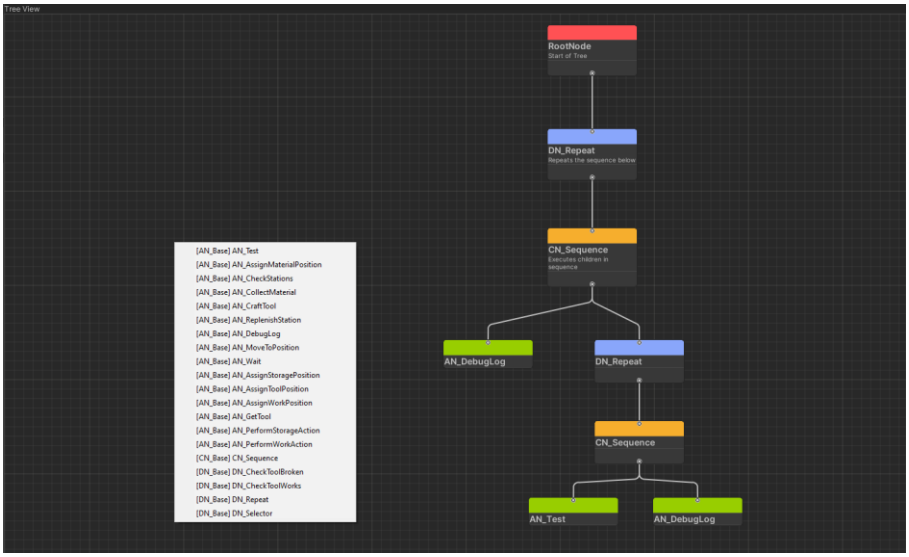

Figure 16: Tree View with a test Behaviour Tree inside and popup for a new node to be created

4.1.2. Nodes

4.1.2.1. Behaviour Tree Editor Display

The UI Builder tool was used to create special node UI which gets made, scaled, and edited based on the type of node it is when a user creates a node within the Tree View. Each node has the name of its script in the middle section of it. When a node is created the top part of its colour gets changed based on whether it is a root (red), action (green), decorator (blue) or composite (yellow) node. Based on the node type the connection spots gets altered as well based on whether the node type can have a parent and how many children they can have. All nodes have a connection at the bottom and then based on how many children it can have that connection can have one or multiple children attached parenting them to the node. All nodes also have a top connection for their parent nodes except the root node which being the start node can not have any parent nodes attached to it.

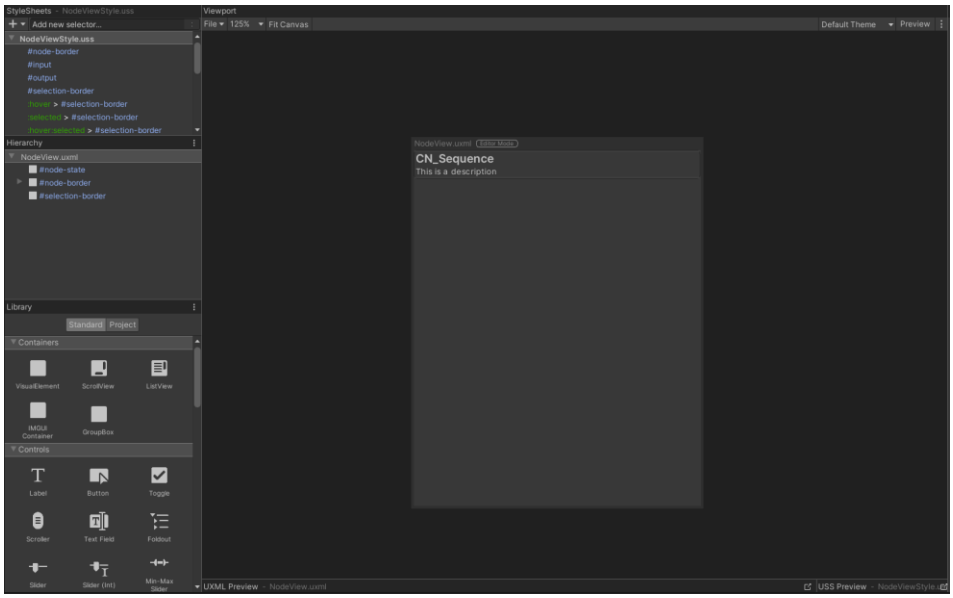

Figure 17: Node UI within UI Builder

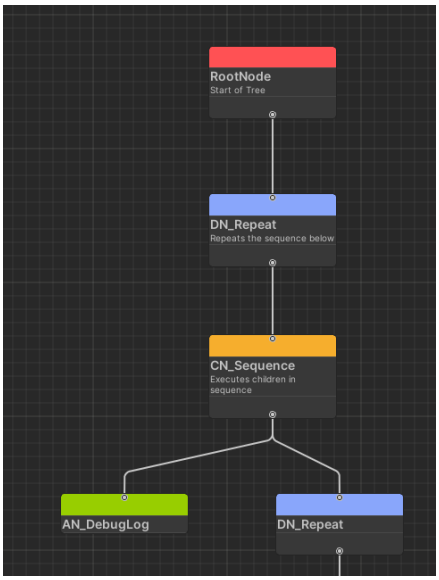

Figure 18: Connection display of different nodes

4.1.2.2. Logic

Each node type contains code inside to perform the logic it needs to function in the Behaviour Tree alongside inherited code from a base node. From there when a node needs to be created in code it can inherit from one of the node types to get its custom execution code along with the base node code and then can have custom code added to it for NPC specific requirements. For example, a sequence node which is used to execute child nodes from left to right inherits its code from a composite node which inherits code from the base node.

Base Node – The base node contains any code that all the other nodes require to run such as their state. This allows for less duplication of code and makes sure that all nodes work using that code preventing errors that could be made with duplication of code. The bade node has also code that can be overridden by anything that inherits it allowing for editing of code in certain cases that require it.

Composite Node – The composite node type has little extra code outside of what it inherits only storing the fact that it can have multiple child nodes and overriding that need to take that into account.

Decorator Node – The decorator node inherits the base node code and has code for having only one child node and overrides base node code that requires editing to make use of that knowledge.

Action Node – The action node unlike the others can have no child nodes and therefore does not need any execution code between the custom nodes and the base node so no base action node exists and instead all custom action nodes inherit from the base node and have no intermediary code class to inherit from.

Root Node – The root node can only have one child, no parents, only one can exist in a Behaviour Tree and it starts the behaviour tree so its code its more custom than the other types. Just like the action node it has no intermediary making it a custom node and it just inherits from the base node code and just like the decorator node it has code and overrides other code to ensure it can only have one child node. Unlike the other nodes however the root node is one of a kind so no other version of it exist. It also contains custom code that execute the one child it has to allow the tree to start.

4.1.3. Inspector

A custom inspector was made for the Behaviour Tree Editor which works with the Tree View to display information about selected nodes and allow the user to edit them. The Tree View has code which communicates with the inspector passing the selected node to it at which point the inspector displays specific information that the node in its code allows the inspector to show and edit such as all nodes having a description which can be edited in the inspector and then the nodes will update showing that description under their name in the Tree View.

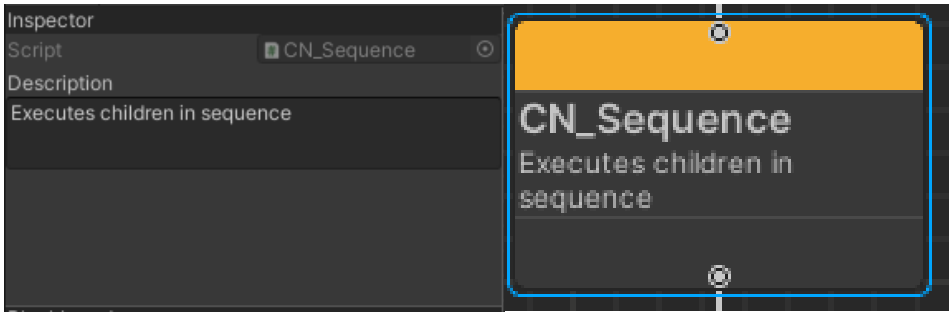

Figure 19: Images showing a selected sequence node and the Behaviour Tree Editor Inspector with a custom edited description

4.1.4. Blackboard

A custom blackboard was made for the Behaviour Tree Editor which similarly to the inspector displays information and allows it to be edited. Unlike the inspector though it is not node specific. It stores custom information from its code that can be used and edited by any node in a Behaviour Tree allowing for better sharing of information and code between the nodes, reducing duplication of code and improving performance by reducing the number of calls in code that need to do to done searching for objects. For example, in this project the blackboard stores a position value the NPCs use to move to a position. This reduces custom code/nodes needed telling NPCs where to move when instead this value can be edited at the end of existing nodes and then a custom universal node called MoveToPosition uses that value calling the code to tell the NPC to move to that value.

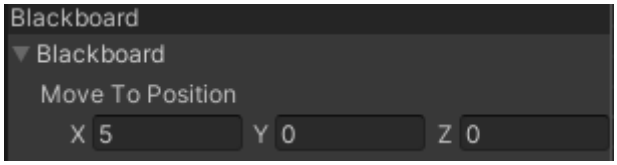

Figure 20: Blackboard with a Vector3 MoveToPosition value

4.1.5. Behaviour Tree

The actual Behaviour Tree logic is stored within Unity Scriptable Objects. The user can create one of these in the Engine and pick a code class that has specific code allowing it to be turned into a scriptable object. A custom Behaviour Tree code class exists for this which defines it as a scriptable object and it stores all the information displayed on the Behaviour Tree Editor. The Behaviour Tree Editor really just visualises this Scriptable object and makes it easier to edit through the custom actions and tools explained above. The code itself for the Behaviour Tree then is what runs and controls all the nodes and logic within the nodes for NPCs to function. It also by default when a Behaviour Tree is made creates a root node as it needs one to run.

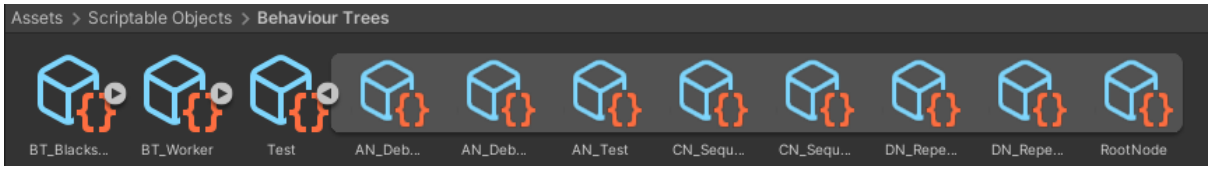

Figure 21: Unity folder showing Behaviour Tree Scriptable Objects and the nodes inside the Test Behaviour Tree

4.1.6. Behaviour Tree Controller

After the Behaviour Tree is fully complete the NPC must somehow make use of it which is where a Behaviour Tree Controller comes into play. This custom code class can be put on an NPC and allows the user to assign a Scriptable Object Behaviour Tree to it. Then when the Scene is started this controller boots up the Behaviour Tree assigned to it and the Behaviour Tree will do the rest controlling the NPC.

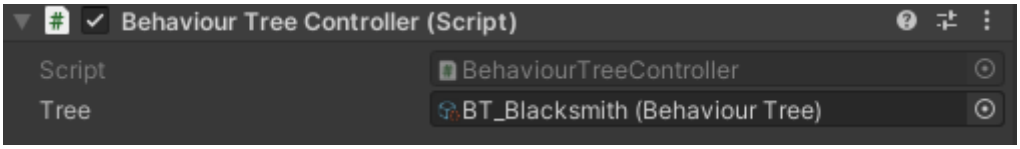

Figure 22: Behaviour Tree Controller script with an assigned Behaviour Tree

4.1.7. A.I. Agent

While everything shown before is enough to get the NPCs to work there is one more script that was implemented to improve performance, reduce code duplication, and clean up the code a lot. This is the A.I. Agent script. It stores all the specific information that are stored in NPC specific scripts such as how many resources the worker NPC is carrying and the movement system responsible for allowing the NPC to have the ability to move. This A.I. Agent script compiles all this information so it does not need to be searched for each time a node runs and all nodes have access to this information as it is built into the base node script and is therefore as mentioned previously inherited by all the other nodes.

4.2 NPC Logic

4.2.1. Introduction

This part of the report will discuss the custom Behaviour Trees and nodes made for the NPCs in the Behaviour Tree Scenario to work. There was two Behaviour Trees made for this project. One is for the Blacksmith NPC and the other is shared by the Miner and Lumberjack NPCs. There were also nodes made that were used in both Behaviour Trees.

4.2.2. Universal Nodes

Some actions do not require very specific code requiring very custom nodes such as moving. This can allow for creation of universal nodes that can be reused in multiple Behaviour Trees and work for those NPCs even if they work very differently to another. Three universal action nodes were made for this project. A MoveToPosition node, a DebugLog node and a Wait node.

MoveToPosition Node – The MoveToPosition nodes makes use of the MoveToPosition value from the blackboard shown before and the A.I. Agent also shown before. It uses those in tandem to get the NPC to move by getting the NavMeshAgent component on the NPC which is in control of the NPC's movement through the A.I. Agent and telling it to use the MoveToPosition value to get the NPC to move to it.

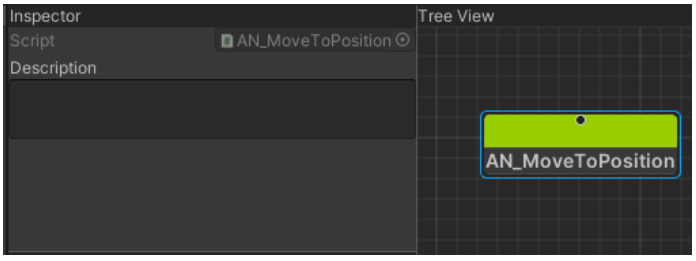

Figure 23: MoveToPosition node in the Behaviour Tree Editor alongside its inspector values

DebugLog Node – The DebugLog node as the name would suggest is purely for debugging purposes. It simply can take text through the inspector when selected and when that node plays it prints that text in Unity's Console. This helps to check whether certain parts of the Behaviour Tree are being hit.

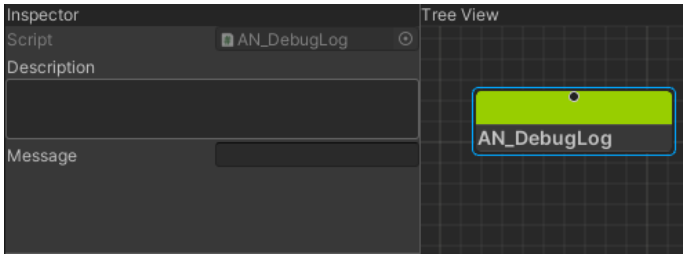

Figure 24: DebugLog node in the Behaviour Tree Editor alongside its inspector values

Wait Node – The Wait node is just a simple timer that does exactly as it says. It makes the NPC wait for a fixed duration. This duration can be edited within the inspector when the node is selected in the Behaviour Tree Editor.

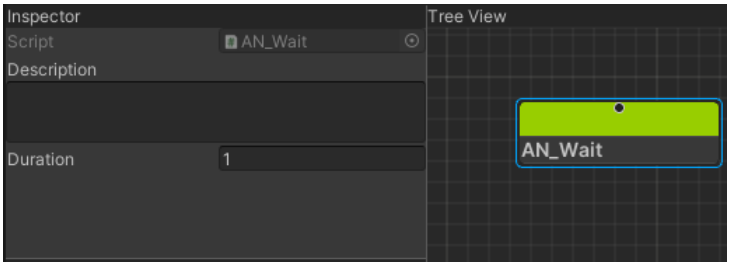

Figure 25: Wait node in the Behaviour Tree Editor alongside its inspector values

4.2.3. Miner/Lumberjack Behaviour Tree

The Miner and Lumberjack NPCs share the same Behaviour Tree as they function nearly identically with the difference being them moving to various positions and using a different resource which can be easily taken into account within the nodes of the Behaviour Tree. This reduces the workload of the project and once again reduces duplication of code.

The Behaviour Tree for the NPCs itself is relatively simple looping over on itself giving the NPCs positions to move to based on their role, tool durability and resources collected, making them wait to signify them performing an action and updating values such as the resources collected and tool durability.

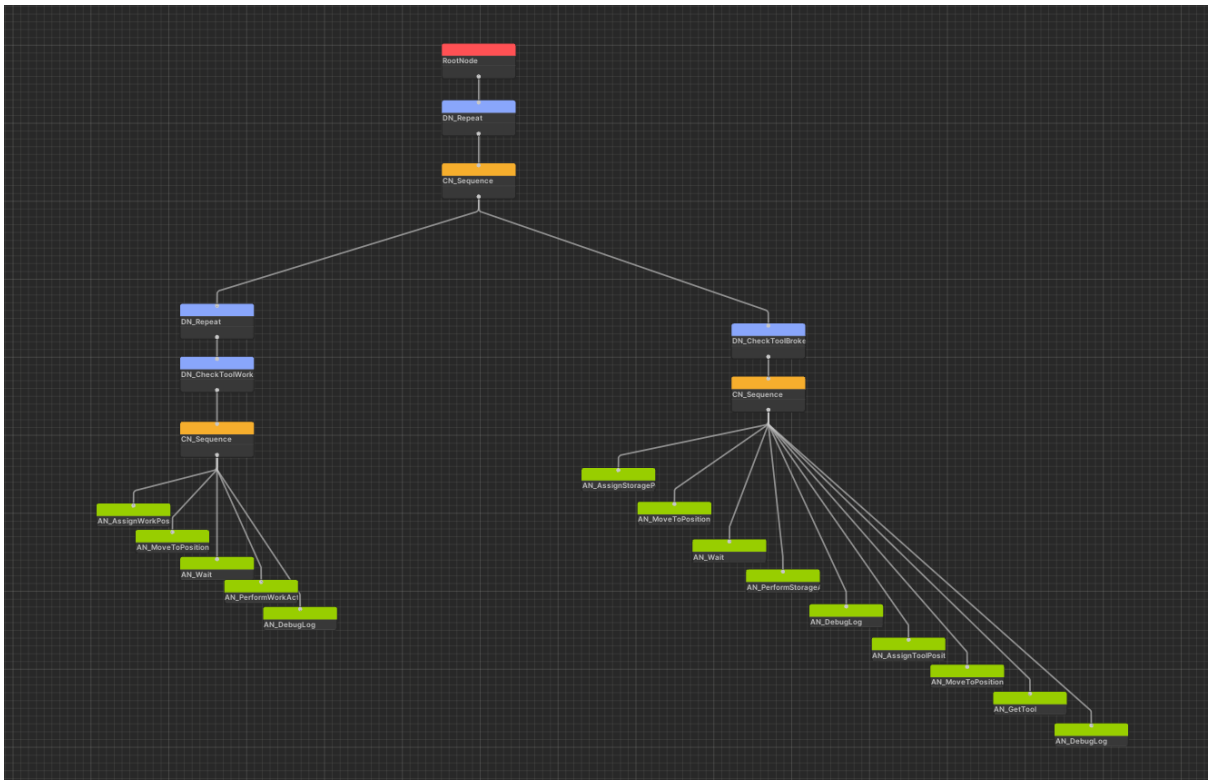

Figure 26: Lumberjack/Miner NPC Behaviour Tree

For this Behaviour Tree eight custom nodes were created to make the logic for these NPCs. Out of these nodes two of them are decorator nodes which decide whether a section of the tree should be able to run. The remaining six nodes are action nodes which provide the custom logic for these NPCs. The rest of the nodes in the Behaviour Tree are universal and base nodes which were described earlier. All these nodes can be seen in the figure above.

CheckToolWorks Decorator Node – This custom decorator node is located in the left section of the tree just below the Repeat decorator node. This node is used to prevent it is children nodes from running unless the NPC has a functioning tool with durability on it.

CheckToolBroken Decorator Node – This decorator node is located on the right section of the tree just above a sequence node. This node work just like CheckToolWorks decorator node but in reverse only allowing the sequence node underneath it to be executed if the NPC's tool is broken as the name of the node suggests.

AssignWorkPosition Action Node – This is the first custom action node within the tree and it is the furthest left located green node in the tree. This node checks whether the NPC it is attached to is a Miner or a Lumberjack and based on that assigns the blackboard MoveToPosition value to be an iron deposit or a tree.

PerformWorkAction Action Node – This node gets executed after the NPC has arrived at their assigned work position following the AssignWorkPosition node and a MoveToPosition node. It simply lowers the durability on the NPC's tool and add to the resources they have collected.

AssignStoragePosition Action Node – This node starts the logic contained in the right section of the Behaviour Tree. It works just like the AssignWorkPosition node however this time it assigns the blackboard MoveToPosition value to be a wood or iron deposit within the storage.

PerformStorageAction Action Node – This node gets triggered when the NPC is at their assigned storage location and takes the resources the NPC has and transfers it to the storage deposit for that resource.

AssignToolPosition Action Node – Just like the previous assign position nodes this node changes the blackboard MoveToPosition value. This one however assigns the value to be a tool station that the NPC needs to go to for a new tool when their tool's durability is gone.

GetTool Action Node – This node gets the tool station the NPC is at and takes the tool from it if there is one available and gives it to the NPC if their tool has broken so the NPC can continue harvesting resources.

4.2.4. Blacksmith Behaviour Tree

The Blacksmith NPC works differently than the other two NPCs and has a separate Behaviour Tree from the other NPCs. However, it still functions similarly by giving the NPC positions to move to and updating values in the code such as what tool the Blacksmith has crafted or whether the Blacksmith has enough resources to craft a tool. The logic for the Blacksmith NPC is simpler than the other two NPCs and has no other NPC sharing its Behaviour Tree making this Behaviour Tree much easier to make only containing the root node, one repeat node, one sequence node, and seven action nodes for its actual logic.

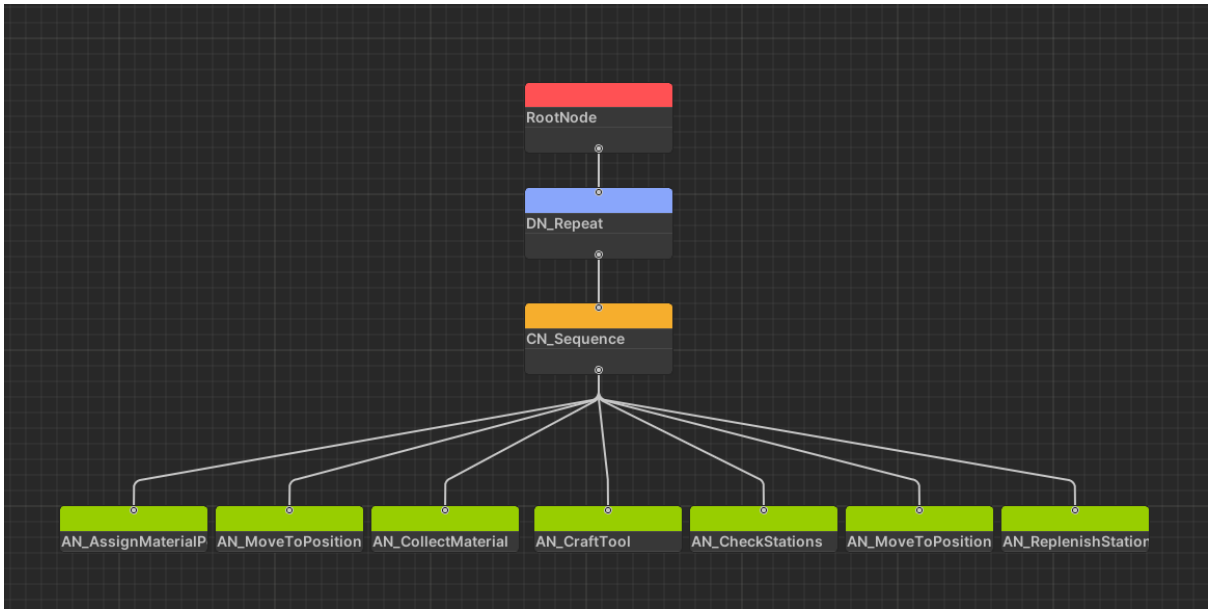

Figure 27: Blacksmith Behaviour Tree

Out of the nodes that are in this Behaviour Tree as seen in the figure above there are five custom nodes made for the logic of the NPC and two universal MoveToPosition nodes. These nodes allow the Blacksmith NPC to go to the positions it needs to go, collect resources, and craft and deposit tools for the other NPCs.

AssignMaterialPosition Action Node – This node returns a position that the Blacksmith uses to go to collect resources.

CollectMaterial Action Node – This node updates values in the Blacksmith code to transfer resources to it from the storage.

CraftTool Action Node – This node allows the Blacksmith to use up their resources at an anvil to craft a tool based on what tool stations are empty.

CheckStations Action Node – This node checks the tool stations and assigns the tool station that has no tool for the Blacksmith to go to.

ReplenishStation Action Node - This node takes the tool that the Blacksmith has crafted and transfers it to the tool station the Blacksmith is at.

5. GOAP Scenario Implementation

5.1. GOAP System Setup

5.1.1. Introduction

The GOAP system for the project after much research was made through the use of various parts of the Unity engine's documentation and a comprehensive YouTube tutorial made by Iain McManus. This tutorial explained GOAP and showed how to implement a monolithic version of GOAP using C# in Unity. The Monolithic version of GOAP works just like the version explained in the literature review which is called Microlithic except where a goal is completed by multiple actions in the Microlithic version instead in the Monolithic a goal can be completed by one consolidated action.

The system itself consists of three main parts. The GOAP Planner which controls the goals and the actions that an NPC is assigned. The other two are the goals and actions which the NPC uses to

perform its logic. Unlike the Behaviour Tree system there is no fancy custom editor and all of GOAP is done through code. The video and channel used in the creation of the system are linked below.

- Unity AI Tutorial: Goal Oriented Action Planning
  https://youtu.be/Q7aHXn_LypI
- Iain McManus YouTube Channel
  https://www.youtube.com/c/Iaintheindie

5.1.2. GOAP Planner

The GOAP Planner is what controls what goal an NPC will work towards and what actions will they use to complete the goal. The planner is just a script that gets attached in the Unity Editor to any NPC that uses GOAP. When the planner boots it first gets all the goals and actions the NPC it is attached to can perform. Then it will give the NPC a goal by going over all the goals the NPC has and then assigning the one that is able to be run based on the circumstances in the game, has the highest priority and whether or not the NPC has actions that allow that goal to be completed. Once a goal is assigned then it will assign the actions the NPC needs to complete the goal. To assign the actions the NPC will use the planner will make use of the A* search algorithm and the cost value of each action to find the lowest cost action or set of actions that are needed to complete the goal and assign them to the NPC. Since the scale of this project is small each goal only has one action that can complete it which means the cost of the actions was not utilised in this project however it is implemented and would be used if more actions were made to satisfy a goal.

5.1.3. GOAP Goals

For the implementation of the goals a base goal script was created from which all the custom goals that are made for the NPC inherit from to get the basic goal functionality. The base goal contains code that can be overridden by the goals made to make editing of code easier. For example, when a custom goal is being made it can overwrite the priority code in the base goal to edit the priority and how it gets calculated or it can overwrite and edit the code that controls when a goal can run. The base goal also contains overrides for when the goal is activated, updated, and deactivated to add custom code that can be triggered at specific times of the goals execution.

5.1.4. GOAP Actions

The actions are implemented similarly to the goals with a base action script that has code that can be overwritten by the custom actions made. The main difference between the two however is the actions use a cost value instead of a priority value and the actions need to have the goals they can complete added to it in the code. Such as a run action having to need connected to the chase goal in the script. All of the code overrides for the execution progress of the action are then used to store the logic of the action.

5.2. NPC Logic

5.2.1. Introduction

The NPCs in this scenario will try to fulfil the same jobs as the ones in the Behaviour Tree scenario however this time they will perform the logic through the use of GOAP. Just like the Behaviour Tree scenario since the Lumberjack and Miner NPCs are extremely similar they will use the same goals and actions while the Blacksmith NPC will use their own. Outside of these there is a universal idle goal and action that just makes the NPC stand still which all the NPCs have as a fallback in case all the other goals or actions were to fail intentionally or not.

5.2.2. Miner/Lumberjack GOAP

The Miner and Lumberjack NPCs had four goals and one action for each goal named identically to the goal it is used to complete. These goals and actions can be seen in the figure below on the NPC along with the custom values that can be edited in the inspector.

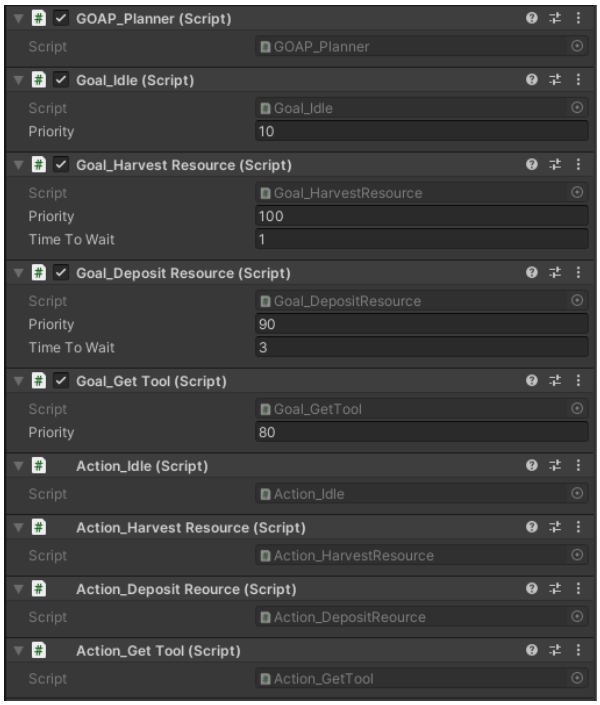

Figure 28: Miner/Lumberjack GOAP planner, goals and actions added to the NPC

5.2.2.1. Goals

The Lumberjack and Miner NPC have four goals as mentioned before. These are the Idle, Harvest Resource, Deposit Resource and Get Tool goals. These goals store their priority and other values that the actions require for the goal to work. These values are accessible to be easily edited in the inspector as seen in the figure above. Each goal also has custom code for when it can run.

5.2.2.2. Actions

As described previously each goal has an identically named action that completes it. These actions use the values from their goals along with custom code to provide the logic the NPCs need to perform their roles. Three of these actions are custom made for this NPC with the exception of the Idle action which is a universal action that can be used on any NPC.

HarvestResource Action – The Harvest Resource action controls to where the NPC has to go to harvest their resource based on their role, the harvesting of the resource and how long they have to wait in between each gathering of the resource to simulate the NPC staying there working for a while. The time to wait can be edited then within the inspector in the Harvest Resource goal.

DepositResource Action – The Deposit Resource action handles where the NPC has to go to store their resources harvested, the actual storing of the resources and how long it takes for the NPC to perform this action. The wait time can once again be edited within the inspector in the Deposit Resource goal.

GetTool Action – the Get Tool action tells the NPC what tool station they can go to get a new tool and it manages the transfer of the tool from the tool station to the NPC.

5.2.3. Blacksmith GOAP

The Blacksmith NPC has four goals and like before each goal has one action named the same way that allows the goal to be completed. The Blacksmith NPC also has a custom fallback idle goal and action. These can all be seen in the image below along with the priority each goal has.

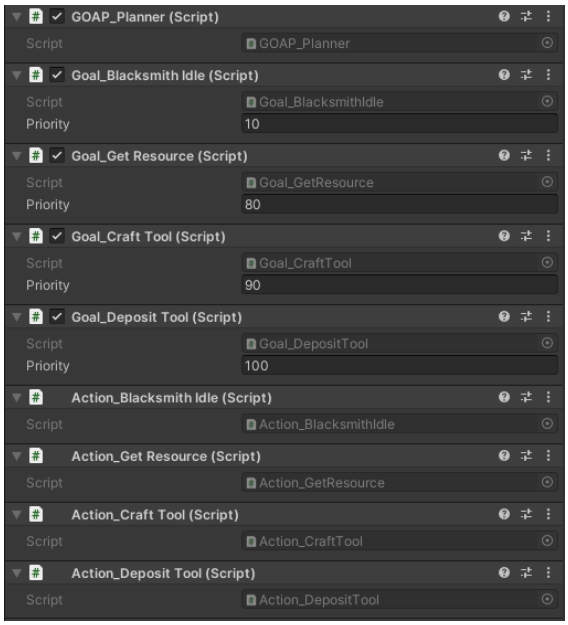

Figure 29: Blacksmith GOAP planner, goals and actions added to the NPC

5.2.3.1. Goals

The goals the Blacksmith uses work similarly to the ones used by the other NPCs. It stores the values its actions need and each of them contains custom code which controls when the goal can be used such as the Deposit Tool goal only being able to be assigned to the Blacksmith if the blacksmith has crafted a tool and a tool station is empty. Each goal also has a custom priority which can be overridden in the inspector in Unity. The Blacksmith also has modified version of the universal Idle goal as a fallback goal.

5.2.3.2. Actions

All the actions used by the Blacksmith are custom made for the NPC since all of the goals are custom made. Just like before the actions take the values that they need from their goals and run their logic for the NPC.

BlacksmithIdle Action – The Blacksmith Idle action works like the idle action except the Blacksmith Idle has the Blacksmith NPC go to their anvil where they can idle.

GetResource Action – This action has the NPC go to the storage where they can pick up the resources they need to craft a tool.

CraftTool Action – This action gets the Blacksmith NPC to go to their anvil if they are not at it already and craft a tool if they have the resources to craft one and a tool station is empty.

DepositTool Action – The Deposit Tool action allows the Blacksmith to go to and empty tool station if they have crafted a tool and transfer it to that tool station.

6. Conclusion

In conclusion the systems work and through their use two identical versions of each NPC were able to be made using those systems for each scenario. Because of the scale of the project there is no visible difference between how the NPCs perform their action to show off the randomness of the GOAP system, however with more time and a bigger scale project where the NPCs have a slew of actions to perform which react to constant changing stimuli in the scenarios then the differences between how the two NPCs act would be clearly visible.

Bibliography

Haytam, Z. (2020, January 7). *Behaviour Trees introduction.* Retrieved from Awaiting Bits: https://blog.zhaytam.com/2020/01/07/behavior-trees-introduction/

Orkin, J. (2002). Applying Goal-Oriented Action Planning to Games. In S. Rabin, *AI Game Programming Wisdom 2* (p. 672). Retrieved from MIT Media Lab .

Orkin, J. (2006). Three States and a Plan: The A.I. of F.E.A.R. *Game Developers Conference*, (p. 18). Retrieved from Goal Oriented Action Planning.

Owens, B. (2014, April 23). *Goal Oriented Action Planning for a Smarter AI.* Retrieved from envato-tuts+: https://gamedevelopment.tutsplus.com/tutorials/goal-oriented-action-planning-for-a-smarter-ai--cms-20793

Unreal Engine. (2020, January 7). *Behaviour Tree Quick Start Guide.* Retrieved from Unreal Engine Documentation: https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/ArtificialIntelligence/BehaviorTrees/BehaviorTreeQuickStart/