Behavior Trees in the Unreal Engine

Function and Application

Herman Båtelsson

Faculty of Arts
Department of Game Design

Bachelor’s Thesis in Game Design, 15 hp
Program: Game Design and Programming

Supervisors: Mikael Fridenfalk, Hans Svensson
Examiner: Masaki Hayashi

May, 2016
Abstract

This thesis presents the implementation and functionality of the user interface for creating behavior trees in the Unreal Engine (version 4.10). The thesis analyzes the final version of the behavior trees in a game development project carried out over one year with a group ranging between four and seven members. The game which is analyzed is a third person adventure game which contains four types of simple behavior trees. These include two enemies who mainly move towards the player to attack whenever in range and two bosses with individual behavior. The thesis describes the various types of nodes available in the Unreal Engine as well as how the behavior trees in the game are structured. Focus is placed on how the structure achieves the required result and how the process resulted in the final version of the behavior trees.

Keywords: computer games, behavior trees, Unreal Engine, game development, artificial intelligence.
# Table of Contents

1. Introduction .......................................................................................................................... 1  
2. Background .............................................................................................................................. 3  
   2.1 Unreal Engine Documentation and Online Resources ......................................................... 3  
   2.2 Behavior Trees in the Unreal Engine .................................................................................. 3  
   2.3 Naar – The Game ................................................................................................................ 4  
      2.3.1 Genre ............................................................................................................................ 4  
      2.3.2 Mechanics .................................................................................................................... 4  
      2.3.3 Enemy Types ............................................................................................................... 4  
3. Materials and Methods .......................................................................................................... 5  
   3.1 Behavior Tree Structure and Communication .................................................................... 5  
      3.1.1 Individual Segments of the Behavior Trees ................................................................. 5  
      3.1.2 Composite Nodes ........................................................................................................ 5  
      3.1.3 Task Nodes .................................................................................................................. 6  
      3.1.4 Decorator Nodes .......................................................................................................... 7  
      3.1.5 Service Nodes .............................................................................................................. 7  
4. Results and Analysis .............................................................................................................. 8  
   4.1 Ghoul .................................................................................................................................... 8  
      4.1.1 Overview ....................................................................................................................... 8  
      4.1.2 Motion ........................................................................................................................... 10  
      4.1.3 Attack .......................................................................................................................... 11  
   4.2 Simm ..................................................................................................................................... 12  
      4.2.1 Overview ....................................................................................................................... 13  
      4.2.2 Catching Nodes ............................................................................................................ 14  
      4.2.3 Movement ..................................................................................................................... 15  
      4.2.4 Attacks ......................................................................................................................... 16  
      4.2.5 Escape .......................................................................................................................... 17  
      4.2.6 Regarding AgroCheck .................................................................................................. 18  
   4.3 Qadher .................................................................................................................................. 19  
      4.3.1 Overview ....................................................................................................................... 19  
      4.3.2 Push Attack ................................................................................................................... 20  
      4.3.3 Enabling Attacks ......................................................................................................... 21  
      4.3.4 Selecting Attacks ....................................................................................................... 23  
   4.4 Shamsi .................................................................................................................................. 24
1 Introduction

When designing AI behaviors for game characters one useful tool is the behavior tree. This system lets the programmers create visual maps of the different possible behaviors an AI can display. It also helps visualize in which way the AI will respond to different situations. Of course the functional benefit of having a user interface in which to easily create, modify and debug the behavior tree is hard to overestimate, especially when comparing the use of an interface to manually scripting the entire behavior tree.

In order to further clarify the structure of the Unreal Engine’s (Epic Games, 2016b) inbuilt behavior tree interface a game development project was examined in parallel to the report. This is done with the intention of producing more direct examples on how the behavior tree functionality is utilized in addition to describing its structure and functionality. For this project the Unreal Engine’s built-in interface for creating behavior trees was used to create several characters with varying behaviors. Two basic enemies who followed the player around with slight variations and two ‘boss’ characters who follow their own pattern, although this pattern, to a certain degree, also includes following the player.

The game project had been carried through for one year at the time of the beginning of this work. Previous work on the game has taken place as part of different courses and focus has been placed on building the base of the game. These bases include character models, animations, movement mechanics and prototypes of the core mechanics. During these stages of development we made the conscious decision to prioritize narrative less than the rest of the game’s content. Instead a version containing the basic mechanics for the game was produced; however it was a demonstration of mechanics rather than narrative. The goal of the current project is to present a demo level of the game containing a tutorial level with several enemy encounters and a simple narrative. While the earliest version of the project, presented at the previous Gotland Game Conference (GGC) in 2015 included functional AI for the boss, it was hampered by a bug in the Unreal engine which made the program crash when attempting to edit the behavior tree. Fortunately by the time this issue occurred we were able to use existing behavior trees as base and make implement the rest of the behaviors using regular scripts. While this worked we completely replaced the AI after that to avoid the same bug. As such the AI for was remade from scratch during the six months preceding this project and further developed throughout the project itself. Because of this the AI for basic enemies existed at the beginning of the current project but needed to be further iterated in order to ensure proper functionality.

This part of the process focused on how we could improve the existing game through the implementation of a full level and implementation or iteration of game systems which included the AI for enemies. This was done within a period of ten weeks in a group of four people unlike the rest of the project which have had a group of six people.

The role of the author in the project is being the only programmer of the group; therefore this report is focused on the implementation of various systems of the game, such as some game mechanics, narrative elements and the AI for non-player characters. The intention of this report is to examine the systems for behavior trees included in the Unreal Engine as well as to determine in what ways the behavior trees can be used to implement the features of the project. This is done by analyzing the final version of the behavior trees used for the project as well as the process leading to the final results. By doing this it will be possible to see how different aspects of the system affected the development process and the final result. It should also be possible to determine from the results how a similar project could be carried out in a
more efficient manner as well as learn what methods worked well for this project. Using this data, as reference could be beneficial to similar future projects involving behavior trees with similar requirements.
2 Background

When creating artificial intelligence for the purpose of making video games, a common way of implementing the systems required, is to arrange them using a behavior tree (J. Champandard, 2008). Champandard describes the basic structure of a behavior tree in one of his presentations (10:27 into the video). By doing this the implementation for an AI character’s behavior process can be made more manageable than creating the entire structure purely using scripts. This process can be made more convenient if the program or engine being used includes a dedicated user interface for creating behavior trees. From one perspective, a behavior tree is a subcategory of a decision tree (Russell and Norvig, 2010). Unreal Engine is one of the engines which have this feature and it was chosen for this project. The version of the engine was 4.10.0 – 4.10.4 as the version had four smaller updates during the project.

2.1 Unreal Engine Documentation and Online Resources

While the Unreal Engine is very much a work in progress, having evolved from the 4.7 version originally used for the first version of the project to the (at the beginning of 2016) current 4.10.4 version which during this development period was then surpassed by 4.11. This could have left the engine in a poor state of documentation due to new features being implemented and old being removed. Despite any negative effects caused by the ongoing developments however the online documentation have proved an invaluable resource in our own process of development. In addition to the official documentation there is also the official UE4 AnswerHUB where developers can post questions and receive advice from their peers or the developers of the Unreal Engine themselves. Because of the large user base of the forum, it was often possible to find another person with a similar problem if we were unable to find a solution on our own. By utilizing both the official documentation and the AnswerHUB forums the process of understanding the structure and use of the engine was made faster and more efficient with less development time needed to be spent on problem-solving (although a considerable amount of time was still spent on problem-solving).

2.2 Behavior Trees in the Unreal Engine

According to their own documentation(Epic Games, 2016c), the Unreal Engine’s behavior trees differ in some ways from other implementations of similar systems. While certainly not the only system to implement this, one of these features is making the behavior trees Event-Driven. What this means is that rather than run the entire behavior tree every update cycle it can instead open up branches on relevant changes to the game variables. For designers this means that the limitations on processing power can be less strict as more complex AI behaviors can be implemented with a smaller cost in processing and in turn performance. This also means that the game could be able to include more individual enemies at the same time as the cost in performance for each one is reduced. The second way is how it implements conditionals as a node of their own (termed: decorators) attached to another node rather than make them leaf nodes at the end of branches. This means conditions does not need to split into two smaller branches with the left returning true or false depending on a condition making the traversal go back up the tree. These both allows for easier overview of the entire tree as well as enabling event-based logic by having the decorators be event-based and only allow access further down the tree (in a representation of a tree structure, where the root of the tree is assumed to have been placed at the top of a chart, branching downward) on a change in the condition. The documentation also mentions the way in which the Unreal Engine handles simple parallel tasks, however for this project all behaviors were simple enough to implement without the use of these techniques. This was mainly as none of the behavior trees had a
requirement for performing condition checks while a task was still running without aborting
the task. Although it is possible that the behavior trees could have been optimized or
otherwise improved by these methods.

2.3 Naar – The Game
At the beginning of this project the game Naar had been in development to various degrees
for close to a year. The goal of the project was to create a functional demo level of about 5 to
10 minutes which contained three main areas containing platforming as well as enemies and
two boss fights, for presentation at the Gotland Game Conference in 2016.

2.3.1 Genre
The game is a third person action game where the player moves around in an environment
doing platforming, solving puzzles and fighting enemies in order to progress. At certain parts
of the game the player encounters stronger enemy bosses which have special behaviors.

2.3.2 Mechanics
The game includes two types of normal enemies, and two enemy bosses. The player
encounters multiple individual enemies of the normal type and engages in boss fights with
single instances of the enemy bosses. The normal enemies have similar behavior but different
abilities which makes the implementation of their behavior somewhat different from each
other. The bosses also have their own individual behaviors which differ from each other. In
total the game employs four behavior trees.

2.3.3 Enemy Types
The four types of behavior trees created for the project are:

Ghoul: A basic enemy whose behavior only involves looking for the player, charging up to
them and performing a close-range melee attack. The AI contains one function which makes
the pathfinding not go in a straight line towards the player until the ghoul is within a certain
distance from the player. This is to prevent a larger number of ghouls from simply moving in
a perfect line towards the player and to give some impression of individual thinking. It was
also implemented as the player originally had a projectile which passed through multiple
enemies and moving in a straight line made the enemies too vulnerable to this attack.

Simm: The second basic enemy type, this type of enemy will not move within a certain
distance of the player on its own accord. It will instead stay at a distance and fire projectiles at
the player. An earlier version contained behavior for also attempting to run away when the
player moved close enough but it was removed due to technical reasons.

Shamsi: The first and smallest of the boss enemies the player encounters. This enemy is
extremely fast and will dash between positions in a circle within the arena he is encountered
in. He will detect incoming projectiles or the player in his nearby vicinity and quickly dash in
the opposite direction. His behavior tree is not very complex compared to some of the other
enemies.

Qadher: The final boss of the demo, the larger of the two bosses, does not move from his
position in the arena he is fighting in. This boss has three stages which depend on how much
health he has left. The first stage has a single attack and each stage change unlocks another
additional attack which he may use to attack the player. In addition to this he also has a push
attack which he will use whenever the player gets too close. This push attack mode will
cancel all but one of his other attack modes.
3 Materials and Methods

3.1 Behavior Tree Structure and Communication

Behavior trees in the Unreal Engine consist of a single class but require two certain other classes to function. The Behavior Tree class contains the behavior tree itself including location of all the nodes and in what order to execute them. The nodes themselves are their own separate classes of blueprints containing the required functionality. Blueprints are the visual classes used in the Unreal Engine containing various code and internal functionality. Two nodes in the same (or different) behavior tree can have the same blueprint class or different depending on requirements. Each behavior tree is also combined with a blackboard-class which only contains any variables required for the behavior tree to run, the blackboard class does not contain any more functionality than storing variables for use in the behavior tree. These variables are placed into the nodes as arguments and used to execute the logic within.

At the highest level of the hierarchy is the AIController class which contains instances of the behavior tree and the blackboard. This class is then set to be the controller for the enemy blueprint which uses the Character parent class. The Character class contains a character movement component which lets the AIController class move the character around the game world. With the exception of the various types of nodes within the behavior trees, the first three classes are the most relevant for this work. This is because they are directly related to the behavior tree itself. However, the rest will also be mentioned on to explain the individual implementation of each character.

3.1.1 Individual Segments of the Behavior Trees

The Unreal Engine behavior tree systems contain four types of standard nodes. Composite nodes, Task nodes, decorator nodes and Service nodes. Each of these nodes except the composite nodes can take variables as arguments which can then be used in the functionality within the node.

![Figure 1: The three types of composite nodes. From left to right: selector node, sequence node, simple parallel node](image)

3.1.2 Composite Nodes

Composite nodes are the most basic nodes included in the behavior trees. Every tree starts with one root node which can only have one connection to another node. This node functionally does nothing but work as the root of the tree. Connected to this is, in all but the
most basic of behavior trees, another composite node. In the Unreal Engine’s case this second node can be one of three kinds. The selector node which will execute each underlying node from left to right until one of them succeeds. The sequence node which will execute all nodes below it until one fails. And finally the simple parallel node which executes a task until completed and also a potential subtree of nodes simultaneously. The latter of these were not used in the project. Using these nodes as the building components of the tree, it is possible to have the tree grow quite complex once conditions other than tasks failing or succeeding are being set to determine which branches of tree the current program traverses down. It is possible to string together complex subtrees under a specific selector which may in turn be part of a larger subtree.

![Selector Node](image)

**Figure 2: Task node (Purple)**

### 3.1.3 Task Nodes
The task nodes are the equivalent of Leaf nodes in the behavior tree. Once the traversal leads to the fulfillment of the appropriate conditions, these nodes are the end of the line for the path. These nodes contain either the functionality itself or call events within another class. In addition to containing functionality they also return a true or false Boolean value indicating whether or not it succeeded or failed. The tree can be structured so that the execution path changes based on this, either continuing down another branch or going back further up the current branch. For example, a sequence node will stop and send the traversal back up once one of the tasks below fails.
3.1.4 Decorator Nodes
The decorator nodes are where most of the functionality within the behavior tree itself is located. In almost every case these nodes are purely conditionals used to calculate variables and return true or false. The node is placed on either a Composite node or a Task node and acts as the conditional for the nodes below. The true or false result is used to determine whether or not the execution path can continue down the subtree or the task below the decorator.

3.1.5 Service Nodes
Service nodes are the nodes utilized the least in the project. Their main purpose is to update values within the behavior tree. They are placed below composite nodes and can be set to run at specific intervals while the current behavior tree is being executed. This is useful for handling values that needs to be constantly updated without having to make task nodes just to update specific values.
4 Results and Analysis

During the course of the project the way the AI is implemented have gone through several iterations. The requirements of various enemies have been changed, such as the Simm no longer needs to run away from the player, or the Ghoul to find a position near the player rather than running straight towards it. In some cases the way an AI was implemented needed to go through noticeable number of iterations as more was learned about how behavior trees worked.

4.1 Ghoul

The Ghoul behavior tree was originally based on the official Behavior Tree Quick Start Guide (Epic Games, 2016a) and therefore contains the outdated variable HomeLocation which was originally used for returning to the starting position after losing sight of the player. The AI also utilizes a few nodes which are modified versions of the nodes described in the quick start guide such as AgroCheck and RapidMoveTo.

![Ghoul behavior tree overview](image)

4.1.1 Overview

The Ghoul behavior starts in the AgroCheck node. It searches for pawns of the player class using a sphere trace to determine if a player is within line of sight. If a player is found it sets the actor reference variable TargetToFollow to become a reference to the player. It also sets TargetLocation which is a vector corresponding to the player’s position in the game. Originally this service node was ran regardless of whether TargetToFollow was set or not but this was later changed to only do the majority of its functionality if TargetToFollow was not the player at the time.
All other branches have a decorator making sure TargetToFollow is set. One of them which are placed last in the sequence is inversed and will only enter that branch if no target can be found. In this event the tree will go into a branch ending in a task node named Find AI Node. This task searches the game environment for any instance of a class named AI Node. This class consists only of a blueprint containing a world location. If any instances of this can be found, TargetToFollow is set to reference this instead of a player. This means TargetToFollow is now set and the tree will traverse into the other branches. If no instances can be found, the tree will continue to look until it finds a player or an instance of an AI Node. This branch is also a good example of an occurrence which is common throughout the various behavior trees of the project, a selector composite node which is holding decorators and branches down to a single task node. This setup is almost entirely redundant as decorator
nodes can be placed directly on task nodes to achieve the same effect. In most cases, the composite node is not required if there is only one task node attached to it. However if there is more than one, the composite node becomes useful. There are two main reasons why this method was used in this project, the first being inexperience as the role of AI programmer was changed multiple times during the project. The second reason is to make the behavior trees slightly easier to overview. These are not very strong reasons for building the behavior trees in this manner and it is not recommended to implement it this way unless there is a chance additional branches will be added from the composite node.

4.1.2 Motion
Assuming the behavior tree has found a target, either player or AI Node, the behavior tree will traverse one of three branches based on range. The first branch is traversed if the Ghoul is more than 600 of the engine’s length units from the player. This distance is set through a float variable. This branch leads to the task node named MoveTo_PlayerSide and is a custom made node which instructs the character to move to a location.

![Figure 8: The first movement branch of the Ghoul behavior tree](image)

It begins by setting a random location from an axis 90 degrees to the left and right of the target (which again, does not necessarily have to be the player) from the direction of the Ghoul. This vector and the vector from the Ghoul forms a T shape when seen from above. The Ghoul will set its goal to be this random location and move towards it until it is within 100 units from the player. The reason for this is to prevent multiple Ghouls from running towards the player in a long line when multiple instances are following the player from the same direction. This method was a replacement to a flocking behavior of the ghouls as it was considered too complex to implement when an almost identical result could be achieved using this method which creates a similar result with less time needed to implement it. Once this
close to the target, traversal will be performed down the next branch; this ends in a simpler task telling the Ghoul to move towards the location of the target rather than a point near it.

![Image](image)

**Figure 9: The second movement branch of the Ghoul behavior tree**

This node, called RapidMoveTo as well as MoveTo_PlayerSide, is written in a manner which causes it to return false both on success and failure to reach their goal, which causes the behavior tree to continue into the next branch regardless of the result of the task. The reason for implementing the branches in this order was to make sure the following branches would be checked regardless of whether one of the movement nodes were completed. This behavior is close to redundant however as the task node’s execution would result in the behavior tree traversal passing over the current branch on the next update cycle. It is also possible that the same effect could have been achieved in a more efficient manner by changing the order of nodes and utilizing observer aborts to let it pass down other branches while running the movement nodes.

### 4.1.3 Attack

The attacking branch starts with a check that will always pass as the distance checker’s range variable is set to zero which is closer than what is physically possible to go for the Ghoul as both actors will have some form of collision. Unlike the two previous branches, this branch contains a sequence composite node rather than a selector. The sequence node branches off to three task nodes. The first, named BTTask_Ghoul_Attack1 or Ghoul_Attack for convenience, returns false if the current target is not a player, thereby making the behavior tree traverse back up the branch and wait for a player to appear. If the current target is a player however, the behavior tree will edit a Boolean inside the Ghoul blueprint which when changed causes the ghoul to begin its attack animation and enable the hitbox on its attacking hand. The node then returns success which causes the sequence node to move on to the following task node which is a standard task named Wait. This node simply holds for a set period of time which is equal to the length of the attack animation. At the end of this time the node returns true and the traversal is made on to the final task node named BTTask_StopAttacking. This node calls
a function within the Ghoul blueprint which disables the hitbox on the hand, and stops the attack animation from looping. The node then returns success and thereby makes the behavior tree execution traverse up to the root before resuming.

This class was one of the first behavior trees developed for this project and several of its functions performs poorly in certain situations causing the Ghoul to get stuck behind walls or not canceling its attacks properly. Many of these were implemented using more efficient methods which were also easier to implement within the other AI types, especially the bosses.

4.2 Simm
The Simm is the second enemy type of the game, an enemy behaving in a similar manner to the Ghoul. It searches for the player, runs to within attack range and then attacks. However the Simm originally had one important difference compared to the Ghoul. Because the Simm is an enemy which attacks the player using long-range projectiles, it was intended to run away from the player if it came too close. This leads to some more complex condition checks being required making the Simm the most advanced behavior tree while this was implemented. However this functionality was later removed, partially because it often worked poorly.
4.2.1 Overview

The general structure of the Simm behavior tree is similar to the Ghoul as they both share the general behavior and were based around the same instructions. (Epic Games, 2016a) The Service Node at the top of the tree works identically to the corresponding node in the Ghoul behavior tree by setting the variable TargetToFollow to be either the player character or an AI Node. The Simm is missing one branch as it has no reason to move to the side of the player the same way the Ghoul does. There was originally another branch in this tree which regulated if the player was too close and in that case performed calculations to find a position away from the player and moving to that position before continuing.
4.2.2 Catching Nodes

In a manner similar to the Ghoul the behavior tree of the Simm has two branches catching different conditions other than the main behaviors. Note that the Ghoul is missing the branch for checking if it is alive as it solves that issue within its character blueprint. The first node ran is an inversed decorator node on the left side of the tree which stops the tree executing down while the character is still alive. This branch ends in a Wait node to keep the tree occupied while the character’s death animation plays and the actor is removed from the game. Every other branch of the tree contains a Blackboard Based Condition decorator node which prevents the process of traversing down the branch if the TargetToFollow is empty. On the right side of the tree there is a branch with this same decorator node but inversed, which means this branch will only be traversed down if the character cannot see the player character from its current position. If this is the case the TargetToFollow will be set to any AI Node placed in the world. This is used to make the Ghouls and Simms move to a predetermined location if they cannot see the player. This is also where the check to compare distances to multiple AI Nodes would be placed if that system was to be implemented. If none of these predetermined locations can be found they simply stand still until the player is spotted by the AgroCheck service node.
4.2.3 Movement

The Simm moves in a fairly simple manner where it will utilize the inbuilt pathfinding system included in the Unreal Engine to find the shortest path to the player and move along it. The difference between the two enemy types is the acceptance radius. For a Ghoul the acceptable distance is 600 units while for the Simm that corresponding distance is 2400 units. This means execution will only traverse down the branch if the distance to the player is greater than 2400 units. At the end of the branch is a task node named RapidMoveTo_Simm which is an identical copy of the same node in the Ghoul behavior tree. In fact the two trees could have used the same node blueprint as none of the functionality within refers specifically to a Simm or a Ghoul. Regardless the node instructs the movement component of the Simm to move towards the location of the current target which is the player or an AI Node. The function used to give this instruction takes a variable for acceptable distance which is why the task node does the same. In the case of the Simm this value is set to 2400. This causes the node to return success up the branch of the tree which then restarts the search from left to right, eventually reaching the attack branch.
4.2.4 Attacks

Originally the Simm’s attack behavior tree was an even more complex system, the redundant nodes have been kept in the figure but they are not used. The branch decorators are the same as the movement branch but with the CloseEnough_Simm node inversed meaning the branch will be traversed down if the decorators on the previous branch (the movement branch) failed. In practice this means that this branch’s task nodes will be executed if the Simm character is within range of the target, regardless of whether it is a player or an AI Node. The nodes themselves have been made more efficient from their original iteration. Currently the first node is the Simm_Attack node which automatically returns true if TargetToFollow is an AI Node, and calls a function within the Simm character blueprint named DoAttack1. This function rotates the character towards the player and sets a Boolean within to let the animation blueprint move into attack animation. The main difference between the final version of the behavior tree and the previous is that the spawning of the projectile is tied to an event called in a certain frame of the animation rather than a delay within the behavior tree. The Simm_launch_spit node originally called the LaunchSpit function from the Simm character blueprint but it meant that a delay node was required in front of it to synchronize the creation
of a projectile to the animation. Similarly the same method made Simm_StopAttack node redundant. The node was used to reset the Boolean within the Simm’s character blueprint which then let the animation state machine exit out of the attack animation. Instead this is now done at the end of the attack animation by calling an event at the end of the animation blueprint. While calling events on specific frames of the animation in this manner leads to those event triggers (not the functionality itself however) needing to be re-implemented if the animation is changed, it is still preferable to handle the sequence in the behavior tree due to the increased flexibility. In a project where animations are being changed often however, it may be more efficient to keep the delay within the behavior tree until the animation files have been finalized. This makes it slightly more difficult to cancel the execution of the branch during the animation, especially if certain values need to be set, but should still be possible.

![Figure 15: The unused escape branch of the Simm behavior tree](image)

### 4.2.5 Escape

The branch of the behavior tree used to run away from the player if it got too close was not use in the final version of the project but contains some interesting functionalities. Originally
placed between the movement branch and the attacking branch this contains functionalities for seeing if the player was too close, setting a point on the opposite direction and moving to that location. The second only lets the traversal path proceed if TargetToFollow is in fact a player as the Simm would have no reason to escape from an AI Node. The third decorator checks if that target is within a set range to the character. The two selectors below the highest both contain the same decorator, although one is inversed. This decorator checks the variable HasBackoffPoint which is a Boolean set true or untrue respectively in the task nodes below. It is initially false as the Simm begins with no path to escape on. The blackboard based condition decorator could have been used to check if the BackoffTarget variable is set but at the time of development The Unreal Engine did not contain a way to un-set variables after giving them a value. This is the reason for the HasBackoffPoint variable. The left Task node named SetBackoffPoint does as the name suggests, the variables it takes is the blackboard key BackoffTarget which is the vector set, TargetToFollow which is the player although it is to escape from the player in the current process, HasBackoffPoint which is not used but is set depending on the result of the node and finally step length which determines how far to escape. The function of the node consists of calculating a point in the opposite direction of the player that is the appropriate distance away. It then sets BackoffTarget to that vector. It also sets HasBackoffTarget to be true before returning false which make the traversal move down the other tree. The second node, MoveToBackoffPoint_Simm then calls the AI MoveTo function with a vector as its target rather than an actor. Resetting values and finishing execution successfully on successful or failed pathfinding. The reason for doing this is to not get the execution stuck in the escaping branch. Because of the way AI Move To works it is moderately easy to make an AI follow another actor, slightly more intricate to follow a non-static vector but moving away from another actor is more difficult. Mainly because of the difficulty determining if the destination vector is a valid point in the world. This is an important thing to consider during the design phase of AI behaviors as it may mean the implementation of escaping states can take longer to build and debug. Finally it is worth noting that we made some progress to implement a way for the Simm to determine if it was stuck trying to escape and if so, turn back and attack the player. This behavior, omitted from the figures, took a considerable time to implement and was eventually scrapped as the initial results were found to be unreliable and exceedingly unsatisfactory.

4.2.6 Regarding AgroCheck
The service node AgroCheck which is identical in both the Ghoul and the Simm could be replaced with a more efficient system. While it was made more efficient by stopping the search if a player was already found, the line tracing process could have been made more efficient if the blackboard set an internal reference to the player actor at beginning of play. The node could then simply search for the distance to the player on regular intervals and only continue to the line tracing function if the player was close enough. This would save needing to run the blueprint function Get All Actors of Class multiple times within the Service node as well as the line trace blueprint function, both of which are relatively resource intensive functions for the engine to run if called every update cycle. It is also possible that the same functionality could have been achieved using another branch containing a decorator checking the distance to the player, a decorator checking the line trace and a task setting the TargetToFollow variable to reference the player. Of course this would also include a way to set the TargetToFollow variable to reference an existing AI Node. Another note regarding the AI Node system is how it currently assumes that the level has only one instance of the class placed in the world. The initial use for this was to make enemies run to the center of the arena if they could not see the player immediately when spawned; however it would not be too difficult or resource intensive to add a comparison in order to find the closest of these nodes.
Of course, as long as no multiple areas with nodes are too close to each other in the level. This function is not necessary for this project but is a simple way of making AI act without needing to see the player.

4.3 Qadher

The AI for the final boss of the demo was also the last of the behavior trees created for the project. One thing that was notable during the process was how lessons learned from other behavior trees could be applied in this tree as well. The behavior of the boss is fairly simple as it does not need to move around the arena in which the player fights him in. Most of the attacks performed by the boss are randomly chosen from a list of unlocked attacks. This list is filled when the boss loses health and goes into another stage. The reason that attacks are enabled this way is in the potential event that new attacks were added in the final stages of development. The Qadher does have one attack which is based on the player’s actions. If the player gets close enough, the Qadher will cancel most of his attacks and initiate an attack which pushes the player away from him.

4.3.1 Overview

Qadher’s behavior trees have the fewest keys in its blackboard, having only three in total. The keys are: PlayerActor and SelfActor both of which are references to the player actor and Qadher’s own actor respectively, Number of Attacks Enabled which is an integer meant for keeping track of how many attacks are currently enabled. One thing worth noting in Qadher’s behavior tree is how almost all of the functionality for actual attacks is placed outside the behavior tree itself. The tree only modifies enumerators within the character blueprint meaning the tree can remain relatively unaffected by the methods the attacks are implemented. The main downside is how the waiting times used for the task nodes in each of the attack branches are not set by the variables. For a future version it may be relevant to make the...
waiting time between each into a blackboard key and replace the Wait nodes with Wait Blackboard Time nodes or make them a single value which is changed based on which attack is chosen in the service node above. Note that the tree also contains a small branch on the left for stopping the behavior tree traversal when the health of the character has reached zero. Finally, the tree does not contain a service node for searching for the player. When the boss is added to the world it does so through an event scene in which the player character stands facing the boss, the arena is also open enough for the boss to not lose track of the player, meaning that the boss can have the player located at the beginning of the boss fight and that functionality for losing track of the player will not be necessary.

![Image](image-url)

**Figure 17**: The close range push attack branch of the Qadher behavior tree

### 4.3.2 Push Attack
The first branch traversed down as long as Qadher remains alive is the push attack path. The reason for this is that this attack should take priority over all others except for one. In order to stop the traversal the first decorator created was CheckIsCurrentAttack_Qadher which compares the character blueprint’s enumerator CurrentAttack to an argument of the same type. By inversing this decorator it functions as a gate to prevent traversal down the branch. Multiple instances of this node could potentially be added in case any future attacks should not be canceled by this. The second decorator is the main check of the branch and compares the distance to the player character to a minimum distance variable. Should the player be within that distance the sequence node is engaged which runs the two task nodes at the bottom of the tree. The first simply set and enumerator named Current Attack within the character blueprint to the push attack which will create a blast sphere to push the player away from Qadher (and play the correct animation). The second task is a standard Wait node that holds
the tree up for the same amount of time it takes for the animation to play. The branch then returns success back up to the top of the tree. It should be noted that in order to cancel one attack for another in this manner (changing an enumerator) the state machine in the animation blueprint needed to have a transition path between all of the cancelable attacks and the attack which to cancel into, regardless what causes the transition.

Figure 18: The branch of the Qadher behavior tree which checks for and enables different attacks

4.3.3 Enabling Attacks
The method for enabling new attacks is the most complex part of Qadher’s behavior tree, the initial selector node has one inversed decorator on it which prevents traversal down the branch once every attack is unlocked. The node takes the blackboard key Number of Attacks Enabled and checks if it is equal to the total number of attacks. This value is simply inputted as an argument into the node but could just as well have been a blackboard key value. The
first branch down (left) begins by running the same inversed decorator but instead checks the value of two. If this decorator returns success that means that the current number of attacks enabled is not the correct number for the second stage. Note that the correct number of attacks in the second stage is two by coincidence. Not because it is also stage number two. The next decorator confirms whether or not the current stage is the second stage, if both these decorators passes that mean Qadher is in his second stage but does not have the correct number of attacks enabled. In this case the Enable_Attack_Qadher task node is reached and executed. This node sets Number of Attacks Enabled to a desired value, in this case two, before returning success and ending the traversal of the Enable Attack branch. If the left branch does not return success however, the selector node instead traverses down the right branch. Technically this could also have a CompareEnabledAttackNumber decorator but that would be redundant as there is already an instance of that node at the top of the current branch. Therefore this smaller branch only has one decorator, CompareFightStage. This decorator node checks if the current stage of the fight is the third. If it succeeds that means the current stage of the boss fight is the third and that the number of enabled attacks is not three. As such, the branch traverses down to the Enable_Attack_Qadher node which in this case takes the number three as its argument. This also stops the entire branch from executing as the current number that will cause of enabled attacks is now three which is the number the decorator at the top of the branch to return false when it is found. The branch is then effectively closed for future cycles as there is no way for the boss to “forget” attacks which have been activated.
Figure 19: The attack branch of the Qadher behavior tree, including unused task nodes for individual attacks

4.3.4 Selecting Attacks

The branch for selecting attacks may contain more nodes than the other branches but the amount of functionality contained is quite small. The service node Select_Attack_Qadher is run a set number of times per second (the number in figure is non-final) and it simply checks if a cooldown timer is finished within the Qadher character blueprint. If it is, a random integer is selected between one and the current number of attacks enabled and the next current attack is selected based on this using a switch node within the blueprint. Regardless of this node, the tree continues traversing down the branch until it finds a path corresponding with the current attack. Each of the paths have a decorator named checkCurrentAttack_Qadher which each contain a different argument and passes the execution through on finding a match between it and the current attack enumerator within the Qadher character blueprint. Originally these all had a node for each attack in order to activate them but all that functionality was transferable to within the animation blueprint so it was decided to only keep the delay nodes rather than keep another timer within the character blueprint. Another way to implement this branch which was considered after the initial version would be making these delays be based on the internal cooldown within the character blueprint. This way this entire branch could be reduced to a single selector node which allowed passage while the cooldown between attacks was not running. If this node passed it would traverse down to a task node containing the same
functionality as the Select_Attack_Qadher service node currently does. The decision would be based on if it was easier to implement and update in the behavior tree or in the character blueprint, the latter was chosen for this project.

4.4 Shamsi

The AI for Shamsi is the simplest of the ones used in the project. The encounter is meant to teach the player how to use a screen-clearing attack which deal damage to every enemy on the screen and is therefore moderately simple compared to the other boss fight. What was important while making this AI was that it should dodge every attack by the player as well as run out of the way when the player gets within close enough range.

![Shamsi behavior tree overview](image)

4.4.1 Overview

The Shamsi Behavior tree is simpler than the Ghoul and Simm behavior tree in that it only has three branches and no service node at the split. Its variables are ArenaCenter, a vector indicating the position of the center of the arena, and a float named ArenaRadius; together they dictate the circle the AI may move along the outer edge off. These values are set upon
initializing in the AIController class after which the character is moved to the outer edge of the arena. This way the character can be moved around in the world during editing of the map without the need for editing the ArenaCenter variable. The blackboard also contains an actor reference named ForeignObject used to easily get the position of potential objects getting within close range to the character. This position is later used to determine which direction to dodge in.

Figure 21: The branch stopping the Shamsi behavior tree from executing if the character is no longer alive

4.4.2 Death
The first branch is simply meant to stop the tree in the event of the death of the character. By running the decorator node named CheckAlive_Shamsi inversed it will make sure the tree skips this branch until the character blueprint is registered as being dead through variables of the blueprint. Once down the branch it will simply run a standard Wait task node with the main task is to keep the tree from running after that character has died. The character is meant to disappear after dying but in the case of a malfunction or unintended delay this method will traverse down the main branch and continue waiting in 5 second increments and do nothing else.
4.4.3 Dodge

The dodge function is the most significant feature of this character. The requirement for the boss fight was that the character should be able to cancel any behavior in the event that either the player or an incoming projectile is detected within a short distance of the character. In order to accomplish this, the branch was the first to be ran every update other than the check if the character was alive. The branch is entered through the decorator node named CheckPersonalSphere, this node is custom-made and searches an area around the character for a list of objects including the players and spells casted by the player. This is achieved by checking for objects overlapping a Collision Sphere object attached to the character blueprint. If objects overlapping this sphere can be found the behavior tree variable ForeignObject to the overlapping actor and the tree advanced down the branch. At the end of that branch it reaches the Dodge node. This node is the most advanced node in this behavior tree, taking the highest amount of arguments and containing functionality for finding a place for the character to move. As arguments it takes ForeignObject which it uses to determine the angle between itself and the incoming projectile, Arena Center and Arena Radius to get the center and size of the arena, the Dodge Angle which determines how far the character moves when dodging and finally Time Dilation which changes the character’s time dilation variable when dodging. The task itself functions by using these values to find a position specific angle to the left or right of its current position in the arena ring and uses a standard blueprint node named Move To Location or Actor after setting its local time dilation to the received value. The node then restores variables used after successfully reaching the target. It does this by resetting the time dilation, animation state and makes the character face the player before returning a success result.
4.4.4 Attacking

The list of decorators in the final branch begins with CheckPersonalSphere like the last branch but in reverse, meaning that if traversal is not performed down the previous branch it will go down this. It also means that this branch only will be executed when no incoming objects can be detected. The other decorators are used in an inverted node to check two variables within the character blueprint, one which is set to true during the attack animation and one which is true during the cooldown period between attacks. Together these two make sure that traversal is not made down the end of the branch until after the appropriate time has passed since the last attack. Functionally the difference is small but one important difference is that this method does not hinder the rest of the behavior tree from running, something which was needed in order to make the character able to dodge incoming projectiles. Finally the branch reaches its end where a task node named Attack_Shamsi which does two things, first it calls a function within the character blueprint which sets the first of the Booleans checked further up the same branch. The same function also makes the character face the player. Secondly, it changes an enumerator within the character blueprint which lets the animation state machine move to the “attack” animation. This animation calls an event within the animation blueprint which appropriately spawns a projectile based on circumstances and engages the timer between attacks. The node returns success immediately but the tree cannot go down the branch until both the attack has been finished and the cooldown between attacks have ended.
4.5 Unused Functions

In addition to the behavior tree functionalities used during this project, there is some functionality of note which were not implemented, either because there was no need or because of unfamiliarity with the system. Two significant examples of this are the Environment Query System (EQS) and AIPerceptionSystem which is a component used to detect other actors in the world. The first is a system which is run as part of a behavior tree in order to effectively examine a number of objects in the world and return the instance with the highest weight. This weight is determined by custom blueprints and could be based on various values in the instance of the object, such as distance, size, state, etc. The main reason EQS was not used for this project was that all types of AI required never needed to keep track of more than at most two other objects. Because of this it was easier to simply make the AI prioritize one of the objects over the other (as done in the case of the Ghoul and the Simm) rather than implement a specific system to do so. AIPerceptionSystem is a component added to the character blueprint and is a more sophisticated way of getting a list of the items the AI can sense (A list which can then be examined by an EQS for example). This system requires less processing power for letting an AI sense other actors than the method used in the Ghoul and Simm during this project, especially if the AI needs to sense a larger number of items in the world. This is also one of the reasons it was not used for this project, as it was more efficient to have the enemies either run a service node at the top of their behavior tree or simply keep an actor reference of the target. Another reason is that the behavior of the enemies was very simple regarding detection: if the player was in sight range, go to the player. The player could not be lost once found. If no player could be found, search through the list of AI Nodes on the level and go to any of them found and wait there until seeing a player. If no AI Nodes could be found or were out of reach, stand still until finding a player. If the game required more sophisticated behavior such as enemies losing track of the player and therefore being in need to keep searching, this system would have been more useful.
5 Conclusion

The system for implementing Behavior Trees in the Unreal Engine is certainly a powerful tool for game development; so powerful that one of the hardest tasks for users can be simply recognizing the multitude of features. Many of the inefficient features which were implemented were later made redundant, by more efficient implementations for similar features in other behavior trees. The most prominent example of this is the attacking function within the Ghoul which was tied to the behavior tree when the same method could have been achieved with events in the animation graph. This method was used to make the attacking branch in the Simm behavior tree more efficient but it was easier to keep the old method in the Ghoul as it still works.

Regarding implementation time, there was a noticeable difference in the time taken to implement the various behavior trees throughout the project. The earliest behavior trees took considerably longer to implement than the trees developed late into the project. While the initial trees for the two basic enemies were slightly more advanced than the trees for the bosses the difference in both implementation time and effectiveness can be seen as a result of varying levels of experience with both behavior trees as well as the user interface of the Unreal Engine. This is an important factor to consider when planning a project. As the experience level of the developer creating the behavior trees varies, the time which needs to be allocated for creating behavior trees does the same. For the developer themselves it is important to do as much research into the system as possible to understand exactly what type of functionality can be utilized and how it could be used to fill the specific needs of the current project.

Finally something that became obvious during the project was that actual functionality and logic could be separated to different degrees. The behavior tree itself could contain various levels of actual functionality depending on the situational requirement, such as spawning a projectile or editing a value within an actor instance. However as experience in the Simm’s attack branch as well as the Ghoul has shown that placing functionality in the behavior tree itself tend to make the system brittle and hard to affect from the outside. Instead it is recommended that the behavior tree call a function from within the blueprint which needs to be affected. By doing this the behavior tree can often remain unchanged despite making large changes to a certain behavior of an enemy as long as the conditions for entering this behavior remains unchanged. It should be noted however that this conclusion does not take into account whether or not it is more efficient to keep the same functionality in a task node as opposed to within a function contained in another blueprint.
References


