Thomas Barraclough AI Development

Please look at the tutorial attached to hand in for running the project file.

Artificial Intelligence For Games Assignment:
Repository Link: https://drive.google.com/drive/folders/1hFIG8HroeJtiJ_a-4sCPfMQonPajrXrt
Words: 4065

Introduction:

Game Overview:

The game made for this project was simple. The player has to find three keys around the world and then give them to an NPC. After this has been done the player fights the final boss. During the game the player will encounter many enemies, to fight these enemies the player will find equipment from chests and enemy drops. Each piece of equipment has its own randomly selected stats and element. There are four elements; fire, water, ground and lightning. All enemies are given an element, if your equipment counters the enemy's element you will deal bonus damage. The water element counts fire, fire counters ground, ground counters lightning and lightning counters water. Your equipment stats and element can be checked at any time by pressing the 'Q' key.

Intended Behaviour of AI Systems:

There are two main types of AI used in my game. One is a procedural mesh generation system, and the other is NPC AIs ranging from enemies to dragons.

The procedural generation system created was intended to make designing the map easier, this was done by spawning a few random meshes from a chosen setlist of meshes. From there options were added to manually set the positions of the spawned meshes. This was used in conjunction with a target point system. Therefore when the game starts the meshes are spawned randomly at every target point location. This system intended to allow the map to have different objects placed each time the game was launched.
On top of that, it intends to allow the game designer to place the procedural generation system actor and have what is spawned be the same for every game session. The foliage generator is intended to achieve this by choosing the random meshes when being placed on the map. However, the ruin generator is intended to be random every time due to being placed at a target point when the game is launched.

There were many NPC AIs created for the game. I tried to use every method of creating an AI when creating these NPCs. The first NPC is a basic enemy NPC, this is intended to attack the player with a basic attack and chase the player if they are out of its range. They are meant to drop a piece of equipment on death and play a death animation.

The dragon and main quest NPC are both interactable NPCs. The quest is intended to tell the player what the goal of the game is, once the player has completed that objective, the AI will teleport them to the boss. The dragon AI is a controllable AI. When the player is in range, the player can take over the dragon and fly it around the map.

The main AI of the project is the boss AI. this enemy AI is intended to give the player a challenge at the end of the game. The AI has three moves and two phases. The second phase is intended to ramp up the difficulty by adding a ground slam move that requires the player to be constantly vigilant. The boss is intended to be the most smooth AI out of the

enemy AIs. The AI achieves this by using a complex behaviour tree. Once the AI is defeated the game ends.

AI Techniques Used:

When creating this project I tried to use most of the popular Unreal Engine AI techniques available. This involved using a behaviour tree, animation state machines, procedural generation and pawn sensing.

A behaviour tree is a tree of selectors, sequences and tasks. This tree is used to determine what behaviour an AI should be used, the tree then runs a sequence of created tasks for each behaviour.

Animation state machines are like a normal state machines, however, this Unreal Engine 4 feature runs an animation for each state. An animation state machine is a graphical representation of different states. Each state is decided by a transition code telling the state machine when to switch to a different state. Each state when selected plays an animation. This technique was both used for NPCs and the player character.

Procedural generation is where a game generates content for a game at random. This could be a random dungeon layout or an entirely randomly procedurally generated world in a game like Minecraft. For my game, I created a procedural object creation system. This randomly creates objects in designated places around the map. For some sets of objects, this creates a random object each time the game is played, for others the objects are decided when placed on the map.

Pawn sensing is a technique for allowing AIs to have different methods of perception. Pawn sensing in Unreal Engine 4 gives the AI sight, hearing and sensing. I primarily used the sight feature to figure out if the AI can see the player.

Reasons For Techniques:

I used the behaviour tree so I could have complete control over when the boss switches behaviours. It also allowed me to make tasks for each behaviour and create sequences of tasks for each behaviour. This is also fairly quick to make, it takes longer than coding without.

I used animation state machines for AIs that didn’t use behaviour trees. Animation state machines allowed me to make my AIs and player switch to the correct animation at the exact time it is needed. This was done through the transition code between states. Although this technique is slower than adding the animations into the code, it is very feasible to make and allows for more specific animation triggers, it also can stop a lot of animation glitches from occurring.

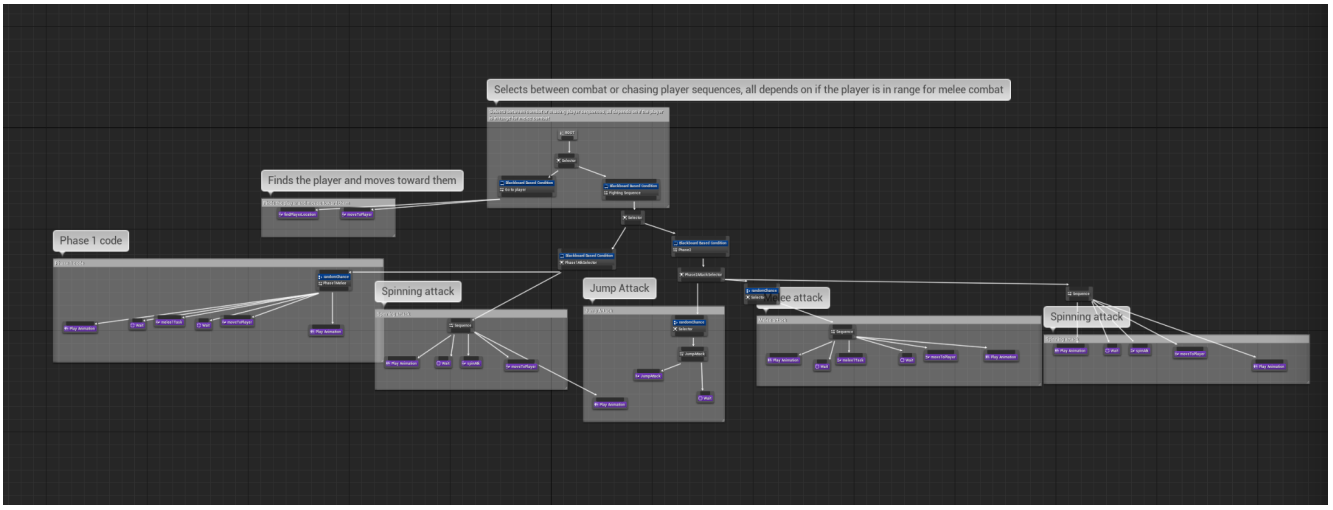
Procedural generation can take a long while to make depending on the desired outcome. Mine was fairly easy to make and allowed me to have random meshes for selected types of outcomes. This did not take an extreme amount of time to make and was extremely feasible.

Implementation Report:

Algorithms/ Architecture Used:

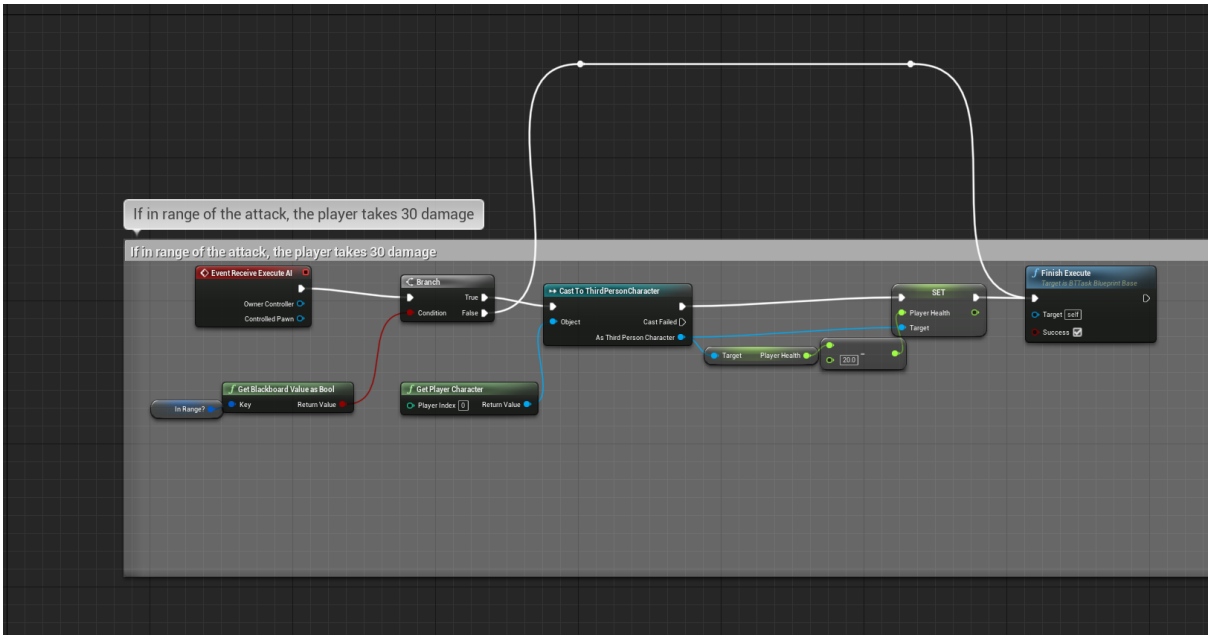
The game and AIs were made using Unreal Engine 4 and were created using the blueprint coding system.

For the behaviour tree for the boss AI many algorithms were used. The behaviour tree was split into two main branches. The first branch of the behaviour tree tells the boss to move toward the player if the player is not in range. If the player is in range the other branch is run. This branch is for attacking the player. The attack branch has 2 branches within it, one for if the boss is in phase 1 and the other for phase 2.



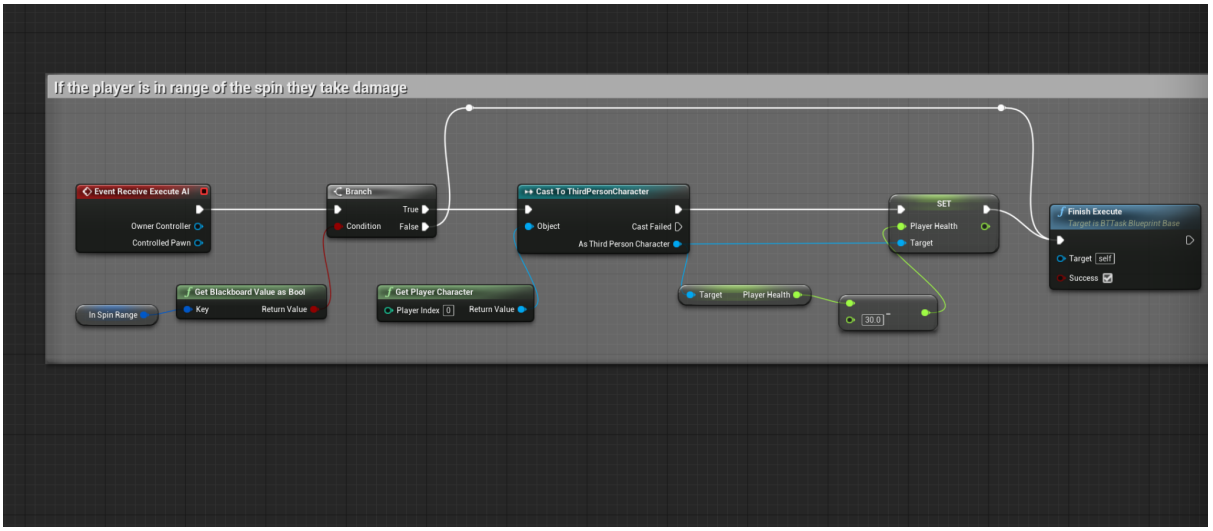
The correct phase branch is determined by a boolean representing if the boss is below 50% hp. The first phase has 2 attacks that can be used. The attack to be used is chosen randomly by a decorator I created. A decorator is a condition for behaviour tree branches. The first attack is a simple melee, this is the same as the basic enemies found in the first area of the game. For this attack, the enemy will first play the attack animation, after a small

delay a damaging task I made is run. This task checks if the player is within melee range, if they are the player takes damage.



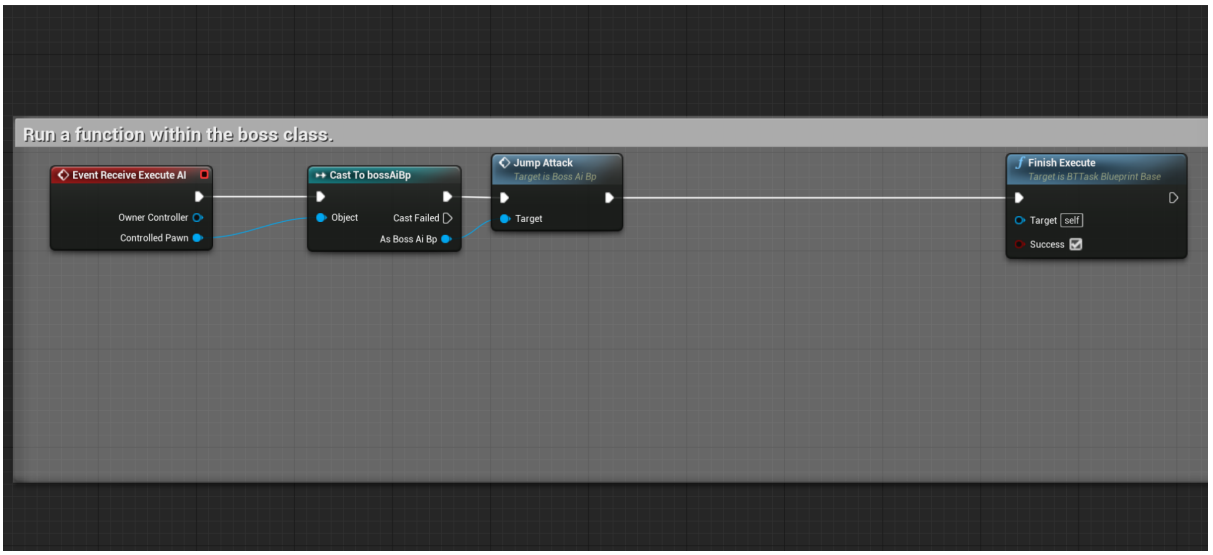
After this, the “move to task” is run which moves the boss to the player.

For the spinning attack, the same sequence of tasks is run, however, instead of the melee task, a spin attack task is run. This task is the same, however, it does more damage.

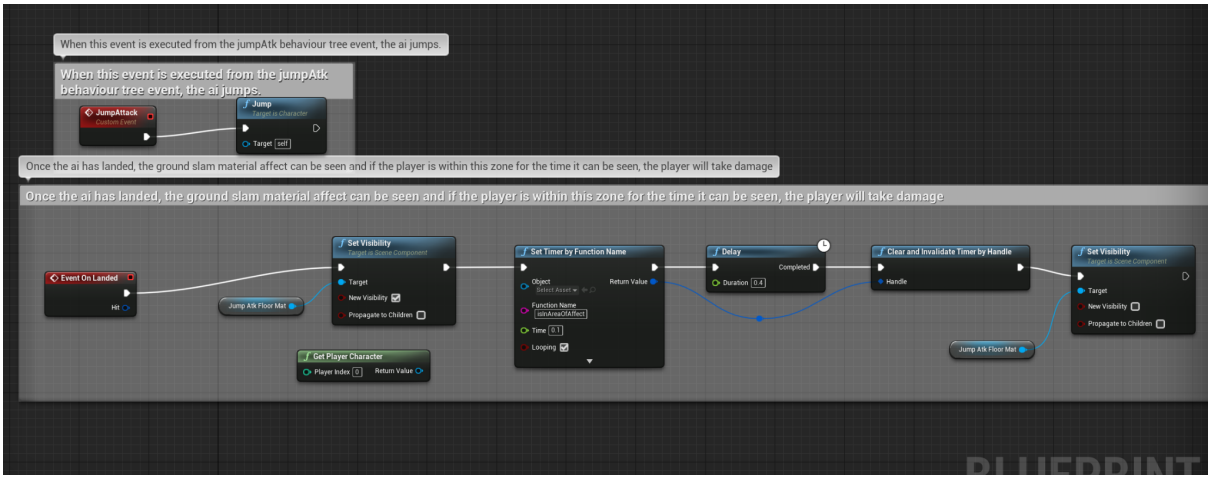


For the phase 2 jumping attack, a jumping attack task is run and then a five-second wait occurs. The jumping task is the most complicated of the tasks, the task itself is simple,

however, in the task, it runs another function based on the boss AI class.

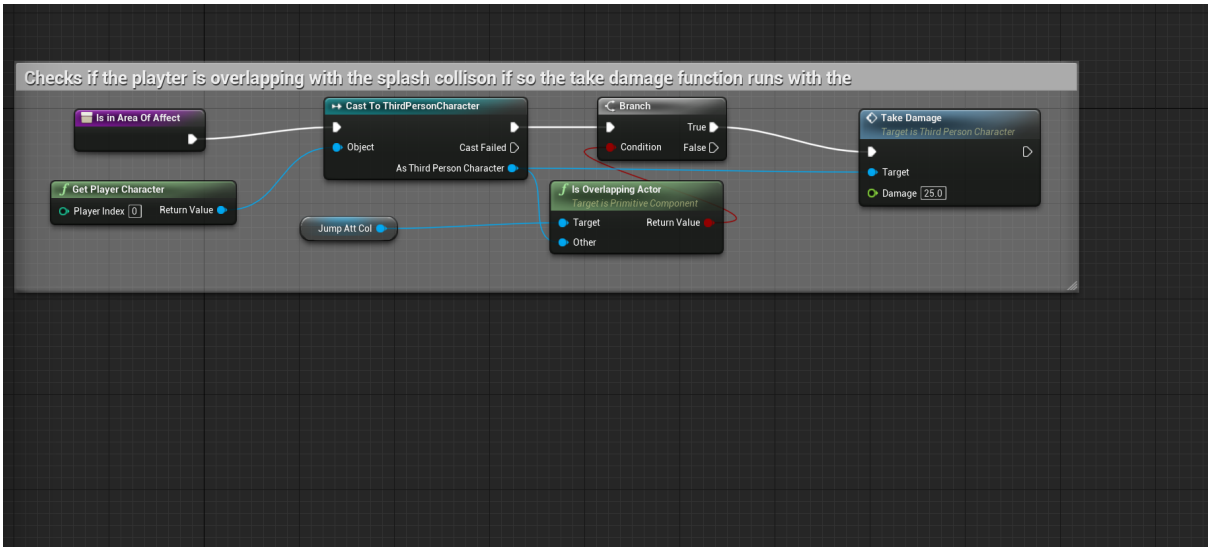


The function ran within the task makes the boss jump in the air. Once landed it makes a red splash zone visible. From there another function ran repeatedly for 0.4 seconds and then the splash zone is set to invisible again.



The function ran within the previous code is for finding the player within the area of effect.

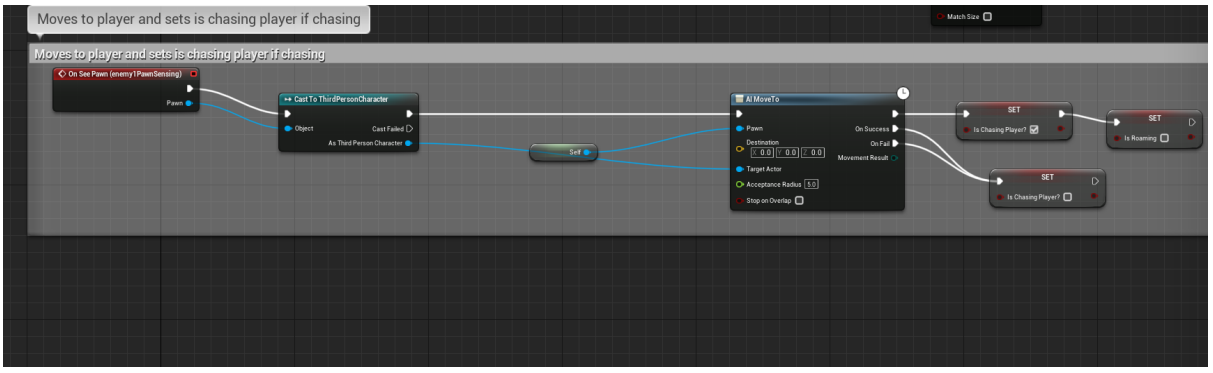
The function checks if the player is overlapping with the splash zones collision if they are yet another function is run with the input for how much damage the attack did.



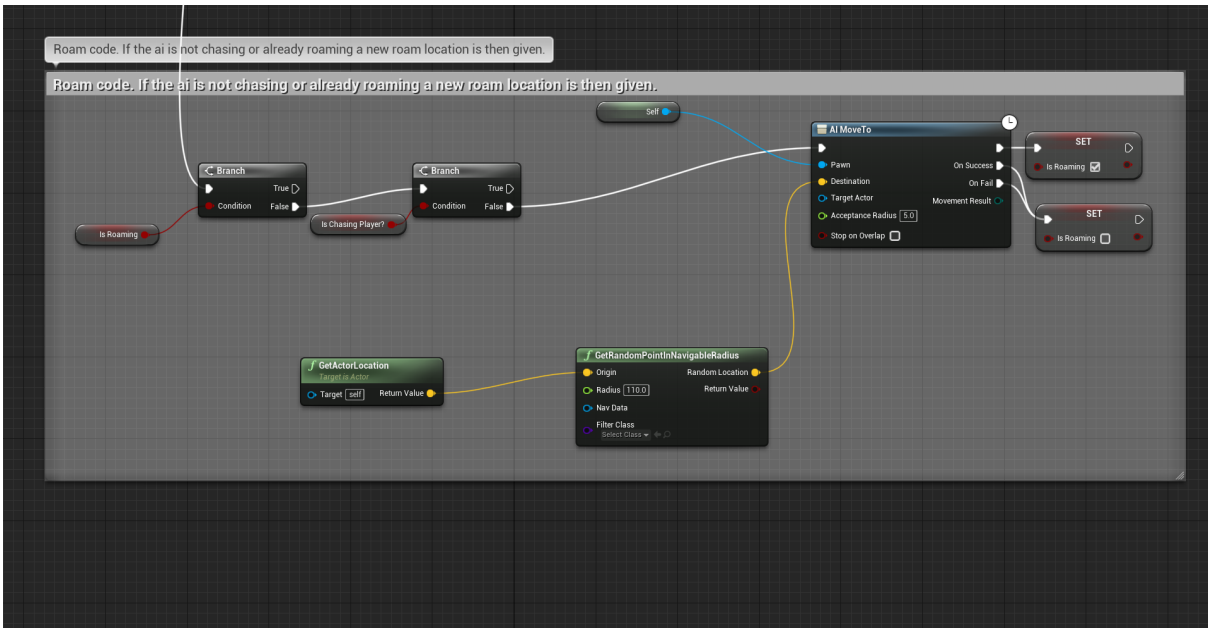
Take damage is a thirdpersonbp class function that takes the damage input and minuses it away from the player's health.

Finally, there is the move to the player branch. The boss first gets the player's location and then the move to the player task is run. This task finds if the player is in range, if not it then moves to the player. After 0.5 seconds or after failing to get to the player / successfully finding the player the task ends.

The next AIs algorithm/architecture there is the basic enemy. This enemy chases the player when the pawn sensing senses that the player has entered the AI's line of vision. From here the code runs. All code for this AI was done through the class, and no behaviour trees or animation state machines were used. For the chase code, when the enemy sees the player it will move toward them and set the is chasing boolean to true, this is it knows to stop roaming.

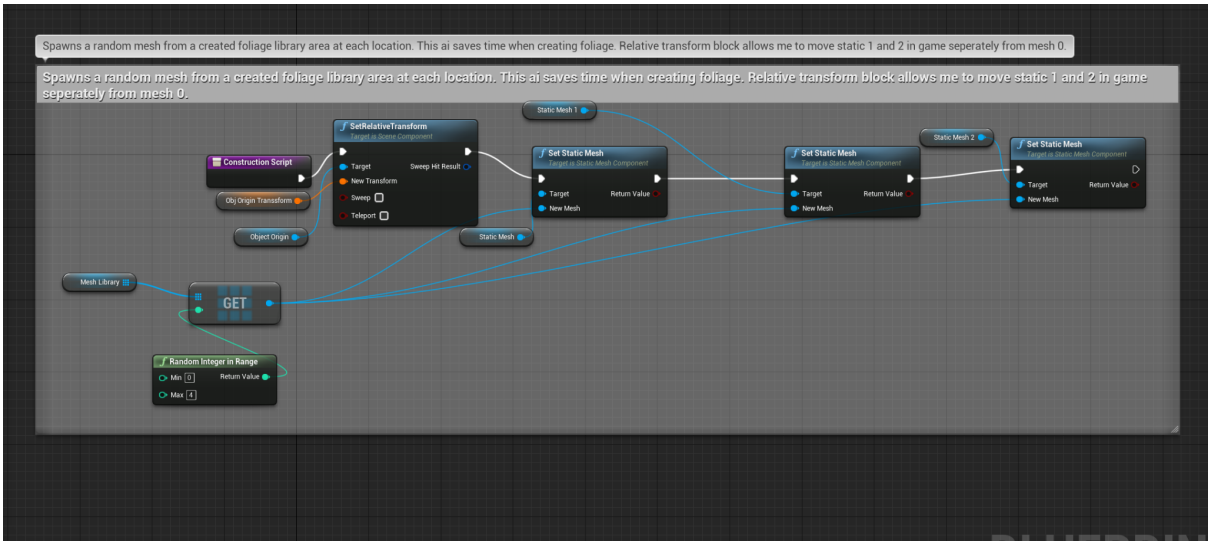


For the roaming code, the boss checks if it is already moving toward a location and if it is chasing the player. If both are false it will be given a random location to move to within a certain radius. It then sets the is moving variable.

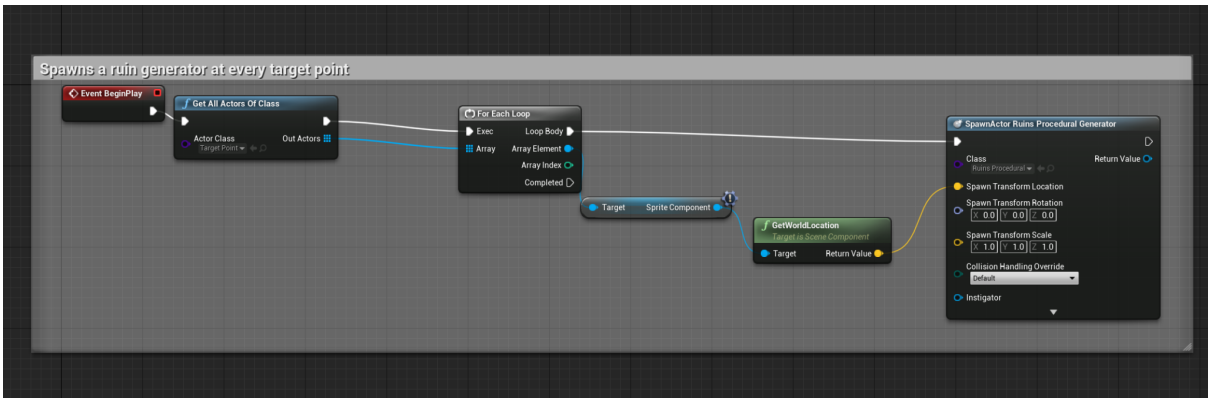


There is also code for taking damage and setting variables, etc. However, due to that not being part of any AI technique or showing anything significantly new I will not go over this. This can all be found in the enemy1 actor class.

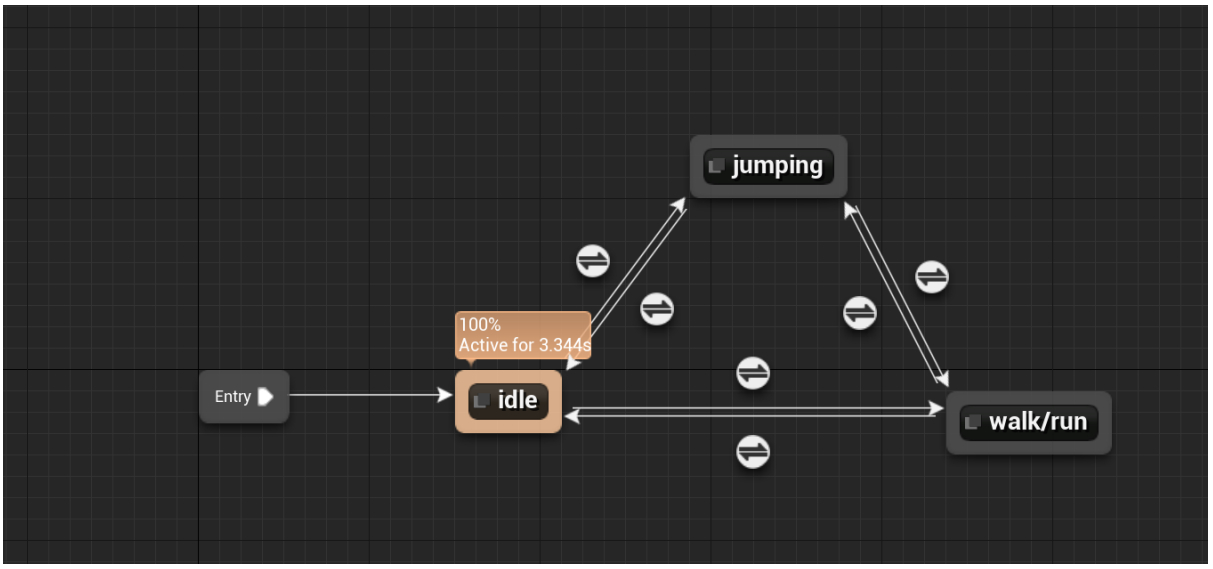
For the procedural mesh generator, the code is all done within the construction script. This script is always run when the actor is placed on the map. This is why the meshes are decided when placing the object instead of when the game starts, this is unless the class for the meshes used target actors. The foliage generator did not use target actors but the ruins generator did. The construction script first allows the user to add custom coordinates for the meshes through a public transform variable. From here it selects random meshes for each static mesh within the actor. The random meshes are chosen from an array list of different meshes.



For the ruin generator, a new class was created called “spawn at target point”. At the beginning of the game, all target points on the map are gathered and for each of these, a ruin generation actor is placed.

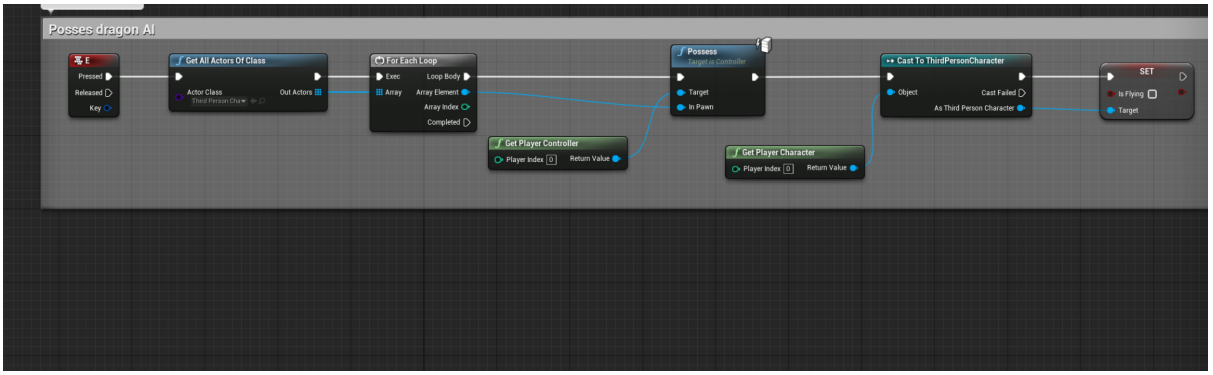


Here is the animation state machine used for the player and quest NPC.



For the animation state machine to be selecting idle, the player/ AI has to be going under 10 speed. For the player/ AI to be jumping the player's “isInAir” variable needs to be true. For the walk/run state to run the player/AI needs to be going 10+ speed.

Finally for the controllable AI, the dragon. If ‘E’ is pressed next to the dragon, the dragon is possessed by the player.



Design Decisions Made:

Throughout the entire process of this project, many changes were made to the AIs and the plans I had for them. At the start of the game, I wanted to spend most of my time on the main boss and also make a procedural dungeon with many types of enemies and rpg game elements. However, as time went on I realised most of my time would be spent on the procedural dungeon and then would not have time to make different types of enemies using many different techniques offered by Unreal Engine 4. Therefore, I simplified the game and focused more on different NPC types, whether that's enemies or dragons. This allowed me to use many more techniques. This made the scale of the project a lot more feasible for me.

After this change, many smaller changes were made. For example, the procedural mesh generation system was not planned until I realised that it would not only be a useful feature but would be a good example of a simple procedural generation AI.

The boss was changed a few times originally it was intended to have three phases with unique movesets for each phase. However as said before, this and the dungeon generator were not feasible for the time frame if I wanted to show off a large amount of the techniques Unreal Engine 4 has to offer. By changing this the bosses desired behaviours were much less complicated but instead worked within a game with much more content and other types of AIs.

Another small change made was with the enemy1 common enemy. Originally these were done with behaviour trees as well as the boss. After a while, I realised the boss already did a better job of showing off behaviour trees than the enemy, therefore I decided to remake it entirely through blueprint code and also used a pawn sense for it. This ultimately made the boss's behaviours much jankier than desired. However, it proved that an AI can be made in Unreal Engine without behaviour trees or any other methods available. By doing this the game experience was slightly lowered for the player.

The dragon was originally supposed to be a set piece in the background of the game, however, after every few weeks of it being in the game, I couldn’t help but think that it would add a lot to the game experience if the player could ride the dragon. After a bit of research, I found making this possible was highly feasible. With that said, I did leave this feature as a bit of an easter egg instead of a major component of the game, I did this due to the jerkiness of the camera when flying the dragon.

Overall a lot of small changes were made. With that said my main goal of having a large showcase of the AI features Unreal Engine 4 gives users was obtained. With more time I would love to have expanded my knowledge of procedural generation and made a dungeon generator, however, changes like this were just not feasible at this moment in time.

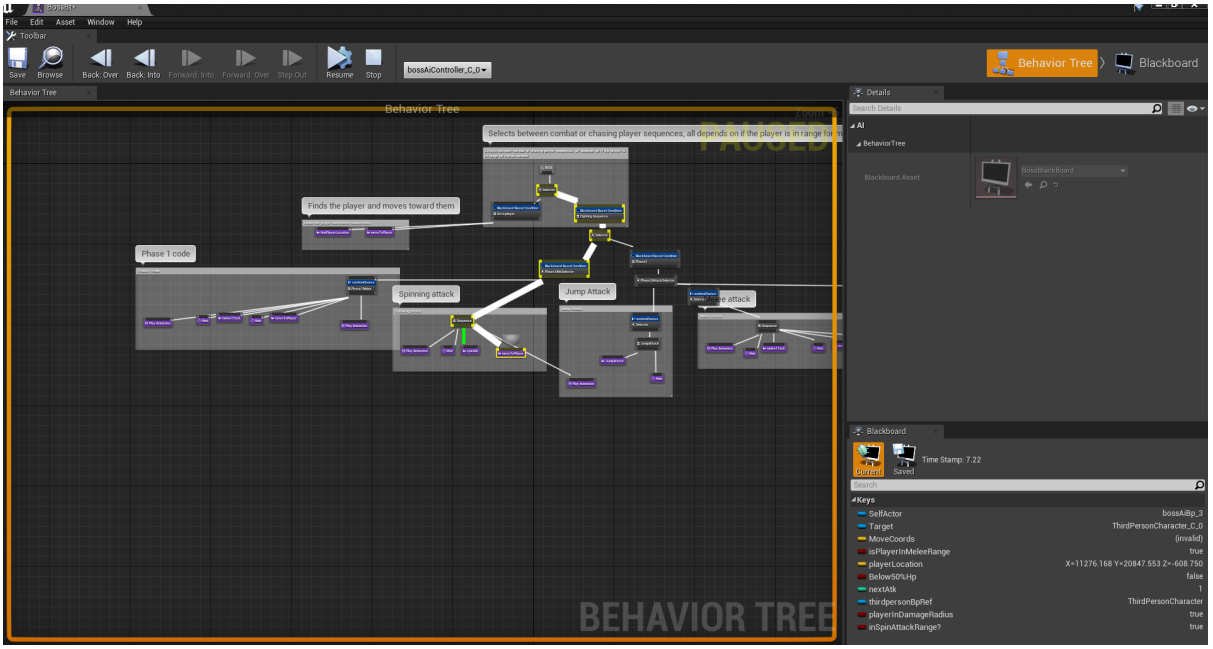
Libraries:

Unreal Engine comes with all libraries needed. Unlike other game development engines, Unreal Engine 4 does not rely on the user needing to browse a marketplace. Therefore everything like behaviour trees and animation state machines come by default with the engine.

Discussion:

Approach Success:

For the boss, I used a behaviour tree. In hindsight, the approach I took with making this AI was the best I could have chosen for the behaviours I desired from it. I wanted the boss to work smoothly and allow for a multi-phase system with various attacks. A behaviour tree was fantastic for this, this was due to the pure amount of control it gives the programmer over the AI. On top of this, it also allows for the programmer to simulate the behaviour tree when the game is running to find any potential problems and see exactly what branch and task is running at any given time.



This allowed me to patch out bugs and make the AI work as smoothly as possible to create a fun and fair boss fight. I also decided not to use animation state machines for the boss AI. By deciding this, it allowed me to control even animations from the behaviour tree, this gave me even more control over the AI.

The behaviour tree approach was also great for both the amount of time I had and the feasibility. Although this feature can take a while to learn, once learnt it does not take much longer compared to coding within the event graph in the class. With that said the extra time spent is well worth it for the improved bug-finding techniques within the behaviour tree and the control you gain over the AI.

The behaviour tree for the boss also computes time and memory usage compared to creating the boss within the behaviour tree. This is the case because unlike in the event graph the code is not having to check many segments of code every frame to see if the AI should be roaming or taking damage etc. Instead, the tree figures which branch to go down and it works its way downwards. This lowers the amount of things code has to check and thus decreases lag and memory usage.

In conclusion, for the boss, I think I choose the best possible approach available within Unreal Engine 4, by doing this the boss fight game experience was much smoother.

For the procedural generation system, I choose the most simple approach of keeping everything within the construction script for the code, however, I also added a separate actor class for spawning these at target points around the map, this was done through the event graph which is the normal area for Coding with Unreal Engine 4. This approach is the obvious best answer for the simplicity of this system, no behaviour trees were needed. This system adds a lot of subtle detail to the mesh around and on the map which improves the atmosphere of the game and thus the game experience. This simple system achieves the exact desired behaviour.

Out of all the AIs, this approach made this AI the most feasible to make and least time-consuming. This AI still took a fair while to make mainly due to wanting to add the option for the programmer to manually change the positioning of certain meshes when placed on the map. The time spent on this was definitely worth the AI created.

Due to this approach mostly working before the game even starts, this does not use barely any memory or computing time. This is because I decided to run the auto-generation system in the construction script this means that it creates the random objects when placed in the world by the programmer and not when the game starts. On the other hand, there are some ruins that use the actor that spawns the random ruins at set target points around the level. This code runs from the event graph, this makes the code run at the start of the game. Even so, this barely costs any memory usage or computing time.

Overall I think I used the best possible approach for this system.

The common enemies are made from the event graph, they are also entirely coded from the event graph. This made the enemies fairly janky movement-wise and somewhat ruined the way their behaviours were intended to run. However, the pawn sensing system worked amazingly for the AI's sight. This system felt realistic in that sense and fun to play and try to avoid the enemy's detection. With that said using this approach definitely negatively impacted the game experience but not by an extreme margin.

This was very feasible and not extremely time-consuming to make. With that said spending extra time patching bugs and making the Ai's behaviour more smooth made this approach not worth it.

This approach to making a basic roaming enemy used a lot more memory and computing time, this is due to the code constantly checking whether it should be roaming or chasing the player / many other events. With that said I could not notice any lag with these enemies, however, this may not be accurate due to me running the game on a high-end PC with an RTX 3080ti.

Overall this approach should not be used for making enemy AIs. It fails in comparison to a behaviour tree in nearly every way. The game experience from this is worse due to the jank within the Ai's behaviour, it uses more time depending on how smooth you want the AI to run the intended behaviour and it uses more memory/computing time.

Animation state machines were used for both the player and quest NPC. This was great for expressing the correct behaviour needed through animations. For example, if the user runs or jumps the animation played will represent this. This approach is amazing for the player's game experience because the animations played will always represent perfectly what the AI is doing (as long as the animation state machine is set up well and is complex enough for the AI it belongs to).

This method does not take a massive amount of time to complete and is very feasible. In some cases, this should not be used even though it is a fantastic approach. An example of this would be behaviour trees as you can also perfectly control an AI's animations through a behaviour tree in Unreal Engine 4.

Animation state machines are good at not using a lot of memory or computing time. This is due to the lack of code being checked at any given moment.

Overall this is a brilliant approach that has a fantastic effect on user experience, however, this does not mean it should be used for every AI.

Limitations:

The behaviour tree method comes with very few downsides! With that said, one common problem a lot of people have reported with behaviour trees is how messy they can become with a lot of code. I had this same problem with my behaviour tree.

For the procedural mesh generator, I used the construction script. This approach was the only one I could use for this system as the mesh needed to be generated before the game starts. Therefore there were no limitations.

For the roaming common enemy, I used event graph code. This caused many problems and limited the AI in many ways. From the get-go, it gave me much less control over the AI, this was due to the difficulty in switching behaviours, as seen with the roam code where multiple

booleans were needed to be set and checked before knowing if the roam code behaviour should be running. Checking every frame also caused a larger amount of memory usage and computing time. Working without a behaviour tree also made simulating the code impossible, this is a huge downside when looking for bugs/errors within the AIs code. Putting together these costs me more time than I would have liked to fix and optimise.

The animation state machine did not limit the project in many ways. It could potentially limit the AI if you are using it in conjunction with a behaviour tree, as it is best to have both in one place.

Alternate Approaches:

For the procedurally generated content, I had no other options available as I had to use the construction script to generate the random meshes before even launching the game.

For every other AI, I could have used any of the systems I used in this project. With that said the clear best approach in Unreal Engine 4 in my mind is the behaviour tree. This outdoes uses event graphs and even animation state machines in many ways. It allows the programmer to layout all code into clear sequences and tasks which can be made and edited by the programmer at will. It also allows for easy testing and error fixing.

Whether the programmer uses the animation state machines or just manually places the animations within the behaviour tree is completely up to them. I personally preferred the behaviour tree for animations but I could see many people thinking the opposite.

I could have used other approaches to making an AI. One such approach was with path driven AIs. To do this I could have used the Unreal Engines spline path feature. This allows for AIs to follow a set path, this could have added some more movement for the quest NPC AI as I could have had it walk a set path.

In another version of this I would have liked to have used the pawn perception component again, this time I would have added it to my boss. I could have given the boss both hearing and better sight, this would have improved the user experience. As of the current version for the boss's perception of the player I used a sphere collision zone.