Bachelor's thesis

Information and Communications Technology

2020

Tomi Vahde

DEVELOPING A BEHAVIOUR TREE BASED AI SYSTEM

TURKU AMK
TURKU UNIVERSITY OF APPLIED SCIENCES

BACHELOR’S | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Information and Communications Technology

2020 | 25 pages

Tomi Vahde

DEVELOPING A BEHAVIOUR TREE BASED AI SYSTEM

Artificial Intelligence (AI) is an important aspect of modern game development. With the vast availability of different development tools, it is easier than ever before to develop an AI system that mimics human behaviour.

The objective of this thesis was to create an advanced artificial intelligence system using behaviour trees, document the development process and compare it to a process of working with state machines. The AI system needed to be easily adjustable, visually self explanatory and provide a fast way of solving potential unwanted AI behaviours .The system was created using the Unity game engine version 2019.2.6f1 and the Behaviour Designer version 1.5.12.

The results highlighted advantages as well as disadvantages regarding the usefullness of behaviour trees when compared to state machines. Main advantages being the visual clarity behaviour tree editors provided, the non-existence of transitions between states and the flexibility of the system. Disadvantages included a steeper learning curve when compared to state machines as well as visual bugs and performance issues.

KEYWORDS:

Artificial Intelligence, AI, Unity

OPINNÄYTETYÖ (AMK) | TIIVISTELMÄ

TURUN AMMATTIKORKEAKOULU

Tieto- ja viestintätekniikka

2020 | 25 sivua

Tomi Vahde

KÄYTÖSPUUTA HYÖDYNTYVÄN TEKOÄLYJÄRJESTELMÄN KEHITTÄMINEN

Tekoäly on tärkeä osa nykyaikaista pelinkehitystä. Eri kehitystyökalujen laajan saatavuuden ansioista on entistä helpompaa kehittää ihmisten käyttäytymistä imitoivia tekoälyjärjestelmiä.

Tämän opinnäytetyön tavoitteena oli luoda tekoälyjärjestelmä käyttäen käytöspuita, dokumentoida kehitysprosessi ja vertailla kehitysprosessia äärellisten automaattien kanssa työskentelyyn. Tekoälyjärjestelmän tuli olla helposti säädettävissä, visuaalisesti selkeä ja tarjota työkalut nopeaan ongelmanratkaisuun. Tekoälyjärjestelmä kehitettiin käyttäen Unity-pelimoottorin versiota 2019.2.6f1 ja Behaviour Designer-ohjelmiston versiota 1.5.12.

Tulokset toivat ilmi käytöspuiden hyödyllisyyteen liittyviä vahvuuksia ja heikkouksia. Tärkeimpiä vahvuuksia olivat järjestelmän visuaalinen selkeys, eri tilojen välinen siirtyminen ja järjestelmän joustavuus. Heikkouksiin kuuluivat jyrkempi oppimiskäyrä, sekä visualiset virheet ja suorituskyky.

ASIASANAT:

Tekoäly, AI, Unity

CONTENTS

LIST OF ABBREVIATIONS **6**

1 INTRODUCTION **7**

2 AI DEVELOPMENT OPTIONS **8**
2.1 Finite state machines 8
2.2 Behaviour trees 9

3 CHOSEN DEVELOPMENT TOOLS **11**

4 OBJECTIVE AND EXPECTATIONS **12**
4.1 Objective and requirements 12
4.2 Expected difficulties 12

5 IMPLEMENTATION **13**
5.1 Integrating behaviour tree with pre-excisting character model 13
5.2 Utilizing prebuilt tasks 13
5.3 Creation of custom tasks 14

6 RESULTS **16**
6.1 Advantages 16
6.1.1 Visual scipting 16
6.1.2 Handling transition between AI states 16
6.1.3 Compartmentalization of code and readability 16
6.1.4 Utilizing prebuilt tasks 17
6.1.5 Sequences 17
6.1.6 Debugging 18
6.2 Challenges 19
6.2.1 Conditional aborts and composites 19
6.2.2 External behaviour trees 20
6.2.3 Performance 21
6.2.4 Visual Bugs 22

7 CONCLUSION **23**
7.1 Suitability 23

7.2 Reliability of results 23
7.3 Future uses and projects 24

REFERENCES 25

LIST OF ABBREVIATIONS

2D	Two dimensional
3D	Three dimensional
AI	Artificial Intelligence
NPC	Non-player character

1 INTRODUCTION

Ever since videogames became popular, artificial intelligence has been a crucial part of them. Today most current videogames have some type of an AI in them, whether it is a simple critter whose only job is to run away from the player, or a highly in depth AI in a strategy game that needs to adapt to constantly changing situtations (Lou 2017).

As games have grown, so have the demands from an artificial intelligence and creation of an AI that can react to different stimulations while mimicking human behaviour can be a challenging task to achieve. In a videogame the different possible situations an AI might need to react to can be multiple and in order to make the AI seem believable, it needs to be able to change its actions in a logical way, much in the way an average human would.

Currently the development of games and AI is more approachable thanks to multiple different and free to use game engines and tools, as well as a plethora of widely available tutorials. One of the most common ways of programming a functional game AI has been the use of different state machines or more recently the usage of behaviour trees. Both of these approaches can create highly indepth AI systems that handle multiple different actions an AI needs to take an well as reaction to outside stimulations. (Lou 2017).

The objective of this thesis is to develop a fully functional AI system using a behaviour tree approach, analyze the development process and compare how behaviour trees compare as a development tool to state machines. The results aim to provide insight into what AI development tools developers can use if they are new to AI development and which ones are more suitable to handle AI systems of different scales.

2 AI DEVELOPMENT OPTIONS

2.1 Finite state machines

A finite state machine is a collection of different actions that a game character can be in at any given time. These individual actions are referred to as states and they are often used in games to handle actions such as different types of movement from running to jumping or calculating combat actions such as target selection. (Unity Technologies 2018).

State machines work by combining different states with transitions between them. Since state machines can only support one active state at any given time, each state must have a transition to the next state, a visual example of this can be seen in figure 1. These transitions are manually set by the developer based on the wanted behaviour of the AI. A transition should only lead to a state that makes logical sense for the AI to take, for example a landing on ground state should always be preceeded by a fall state. It is also noted that not every state needs or should have a transition to every other state inside the state machine. (Bevilacqua 2013).

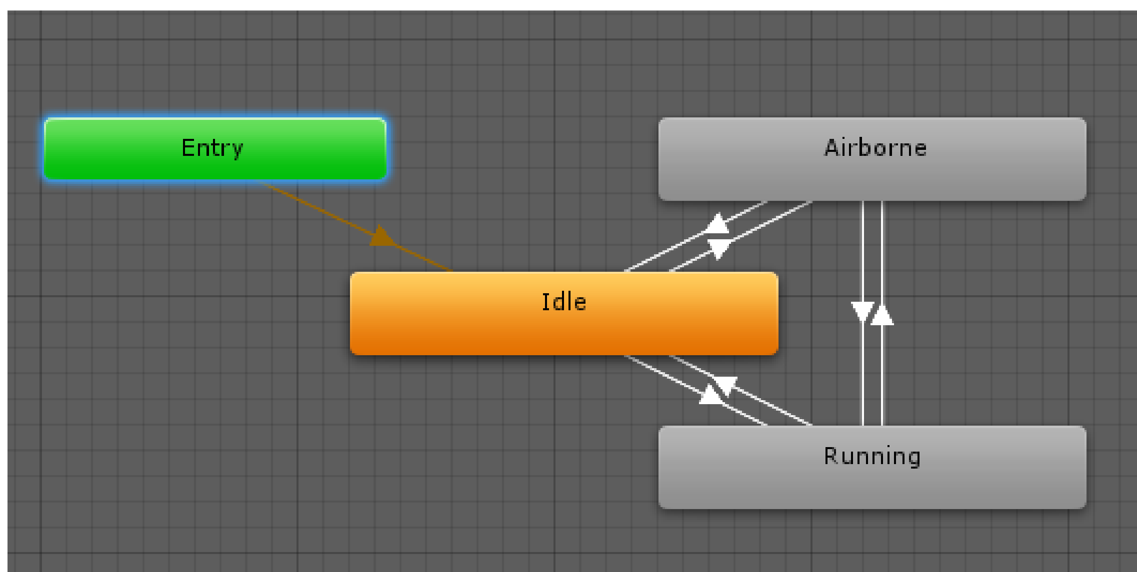


Figure 1. State machine in Unity consisting of three different states with currently active state marked in orange and transitions represented with white arrows. (Unity Technologies 2018).

While state machines are relative easy to understand and set up inside the Unity editor, they do have their own downsides. These come mostly in the form of increased transitions as the amount of different states for the AI increases. If an AI needs to support, for example, 10 different states and each state needs to have a transition to every other state, the amount of transition that would have to be manually set would be 90. Managing so many transition would take up significant amount of development time and can easily lead into unwanted bugs as well as making the state machine visually look cluttered and hard to read as seen in Figure 2.

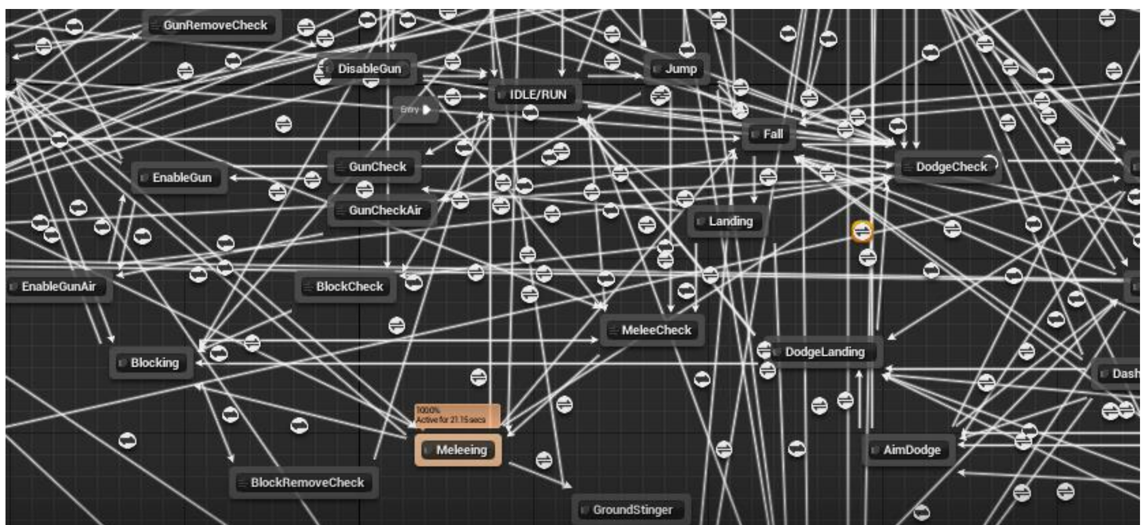


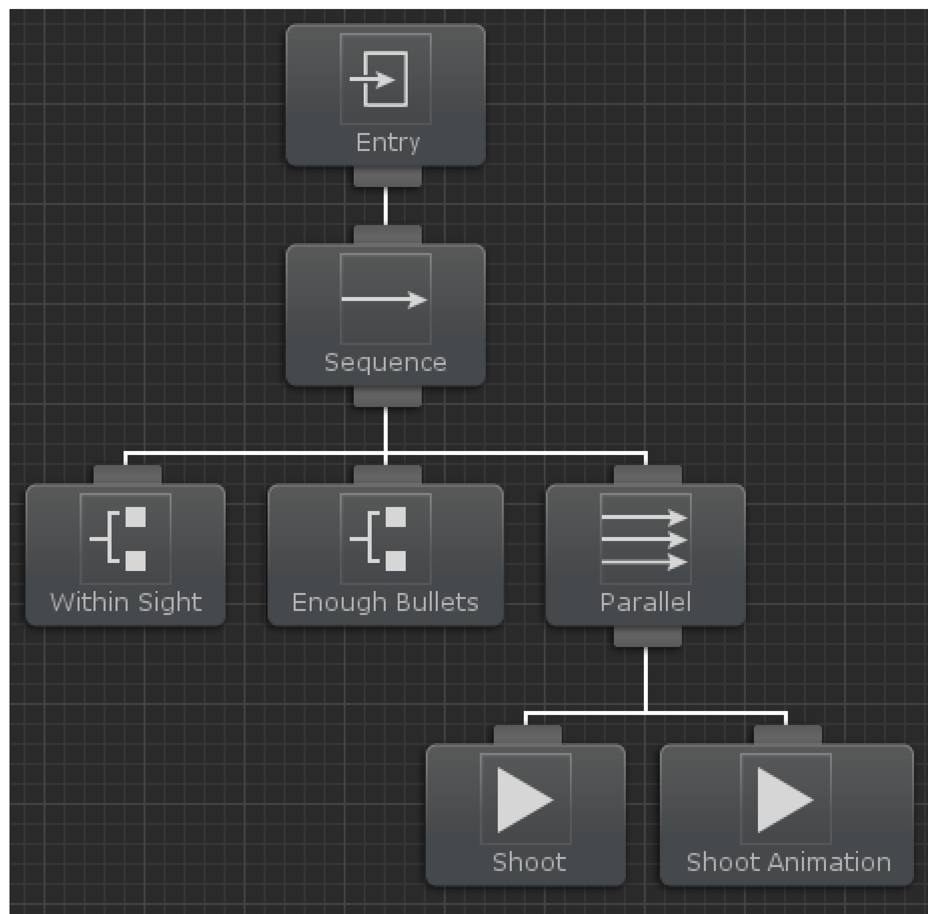
Figure 2. Cluttered state machine with a high number of states and transitions (Opsive 2020).

2.2 Behaviour trees

One alternative to using state machine for handling AI logic is the usage of Behaviour trees. Compared to state machines, the usage of behaviour trees in game development is still rather new with the first major usage of them appearing in the game Halo 2 released at 2004. Behaviour trees differentiate themselves from state machines by not relying on manual transitions and states, but rather by having a collection of different tasks the AI can excecute in different sequences. (Opsive 2020)

The tasks that behaviour trees consist of work by running code that checks wanted parameters and returns either a failure, success or currently running status. The sequence will then decide whether to proceed to the next task, wait or to quit the

sequence altogether. The excecution order of these tasks mainly follow a preorder binary tree traversal, where the logic flows from top to bottom and left to right, but this behaviour can be manipulated with different composite tasks. For example, a parallel task seen in Figure 3 will run both child tasks shoot and shoot animation at the same time. This is also a differentation from state machines, since behaviour trees can run multiple different tasks at once.



.

Figure 3. Example of a behaviour tree consisting of an eyesight check and shooting tasks. (Opsive 2020).

3 CHOSEN DEVELOPMENT TOOLS

Unity itself does not have an integrated behaviour tree system, therefore it was necessary to look for third party developed tools in the Unity Asset Store. The Asset Store for Unity consists of free and purchasable extensions to the for the Unity editor, that can extend it's functionality in different ways.

For this project the chosen system to use was Opsives Behaviour Designer for Unity. The main reasons for choosing this specific asset was it's support for other third-party assets currently used in the game project, most notably A* pathfinding, which handles all the AI movement logic and Dialogue System for Unity that is used for managing quests and character interactions by talking to different NPCs.

Other notable reasons for selection were it's large userbase and documentation, this in the hope that finding solutions to unexpected problems would be easier with a larger community and more active forums than some other assets. Since Behaviour Designer also receives frequent updates, it is unlikely that compatibility issues will rise with newer version of Unity. It is not uncommong for third party assets from the asset store to become deprecated with newer version of Unity, if the asset hasn't been update to support the newest release.

4 OBJECTIVE AND EXPECTATIONS

4.1 Objective and requirements

The objective is to create a basis for an AI, that can change it's actions between states, works with pre-excisting code and supports future expandability.The finalized system needed to support the addition of new tasks and creation of different AI variants, which would utilize the already existing behaviour trees.

In addition to this, it needed be possible to adjust the different variables of certain tasks on the fly. For example the range of an AI's eyesight could be impacted by gameplay, so different tasks need to dynamically support these changes.

4.2 Expected difficulties

Reading the documentiation of Behaviour Designer and getting familiar with some of it's tutorials projects, it was expected that a few challenges would arise during the development.

The most obvious challenge was expected to be handling of the tree traversal logic. While simple behaviour trees could consist of a single sequence, an extensive behaviour tree that handles everything from movement, target acquisition, dialogue interactions and cutscene logic would end up being much more of a challenge, since it needs to tie different sequences to work together.

It was also unclear how manageable it would be to integrate individual tasks to work with already excisting scripts in the project. One fear was that it might be required to set up constantly running tasks, which do not take into account the current state of the behaviour tree, for the reason that some of these scripts were not initially developed to work with behaviour trees in mind.

5 IMPLEMENTATION

5.1 Integrating behaviour tree with pre-excisting character model

Prior to working on the AI system, the game project already contained scripts for handling character animations. These animations scripts consisted of roughly 2000 lines of code, therefore it was important to get the AI system working with this codebase, in order to not having to re-write the logic behind the animation handling. Secondly it is expected that the original animation scripts will receive new functionality or alterations, therefore it was highly important for not having separate animation handlers for the player and the AI.

The majority of these animation handlers were located in the base character class. This was an extendable class used by the PlayerInput class, that would automatically change the characters current animations depending on current player inputs. Solution to this was to create an NPCInput class, that would also extend the base character class. This new class would consist of different functions that the behaviour tree could freely use inside different tasks.

5.2 Utilizing prebuilt tasks

Behaviour Designer has by default multiple prebuilt tasks for developers to use in their behaviour trees. The usage of these was as simple as dragging them into the currently edited behaviour tree and setting up all the needed references in the inspector, these mostly consisting of the gameobject that you wanted to track or the possible angle for a field of vision. While the amount of prebuilt tasks varied based on their type, the more commonly used gameobject components did have a wider selection. For example the animator component that is used to handle all character animations from walking to running had an extensive amount of tasks allocated to it as seen in figure 4. But the lesser used ones like the BoxCollider2D component had only two prebuilt tasks, get size and set size.

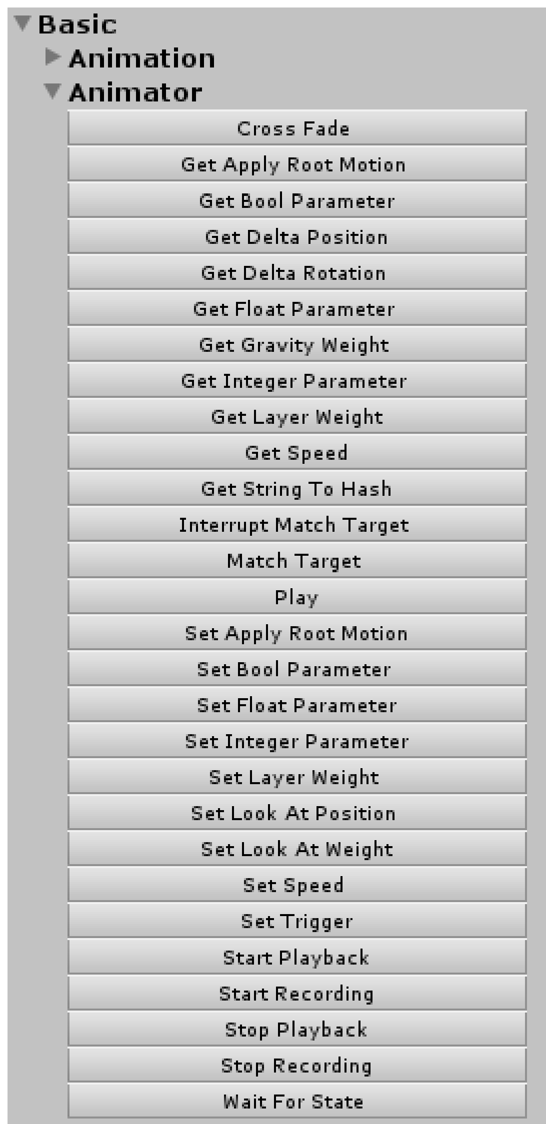


Figure 4. Example of animator related tasks.

5.3 Creation of custom tasks

The creation of new tasks was mostly only needed to bridge the gap in delivering and receiving information from the NPCInput and BehaviourTreeReference classes. This was usually done by taking pre-existing tasks and modifying them in order to create a new variant. Since most of the low level AI logic was done inside the NPCInput class, the tasks classes ended up usually only having a few lines of code that called certain functions inside the NPCInput class. Figure 5 provides a good example how it was possible to keep custom tasks down to approximately 40 lines of code by mostly doing

comparison tasks inside the OnUpdate function and making the task in question return the wanted outcome.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using BehaviorDesigner.Runtime.Tasks;
4  using UnityEngine;
5
6  public class NPCDrawnWeapon : Action
7  {
8      NPCInput myNpcInput;
9      Renderer gunHolsteredRend;
10     SetBehaviorTreeReferences myBehaviorTreeReferences;
11
12
13     public override void OnStart()
14     {
15         base.OnStart();
16         myNpcInput = gameObject.GetComponent<NPCInput>();
17         myBehaviorTreeReferences = gameObject.GetComponent<SetBehaviorTreeReferences>();
18         gunHolsteredRend = myBehaviorTreeReferences.HolsteredGunGO.GetComponent<Renderer>();
19     }
20
21     public override TaskStatus OnUpdate()
22     {
23         //Check if we have a weapon to draw from holster
24         if (myNpcInput.MyHolsteredWeapon == null)
25         {
26             //Can't equip anything if no weapon in holster
27             return TaskStatus.Failure;
28         }
29
30         myNpcInput.EquippedWeapon = myNpcInput.MyHolsteredWeapon;
31         myNpcInput.MyHolsteredWeapon = null;
32         gunHolsteredRend.enabled = false;
33
34         return TaskStatus.Success;
35     }
36 }
```

Figure 5. Example of a custom task

6 RESULTS

Working with behaviour trees proved to be very different from programming finite state machines. Here we will look at the biggest advantages and challenges that rose up during the development.

6.1 Advantages

6.1.1 Visual scipting

Since Behaviour Designer works by a visual editor inside Unity, setting up different AI tasks and behaviours is a completely separate process from actual programming. For this reason the AI development process can be divided between multiple people in larger development teams. A programmer can focus on creating the individual tasks the AI has to execute, where as a second person can handle setting up the tasks inside the visual editor. In this case the second person does not neccecarily need to know actual programming, since organizing the behaviour tree can be done by visually dragging different tasks to their correct positions. When working with finite state machines this is not a possibility, since all the work that goes into handling AI states and behaviours requires actual programming.

6.1.2 Handling transition between AI states

Main advantage of using a behaviour tree from a programmers point of view was the fact of not having to worry about handling transitions between states. When all the selectors were set up correctly the logic flow behaved excatly as it was required. This also helped keep the visual editor looking clean, compared to what a state machine can look like when the amount of different states and transition grow. (Add picture here)

6.1.3 Compartmentalization of code and readability

Since Behaviour Trees work by running individual tasks, this had the positive effect of automatically forcing you to separate your code into individual classes that the tasks use.

This made reading through the written code much more easier, since the common mistake of writing script classes that consist of thousands of lines of code wasn't really a possibility.

The visual editor also proved helpfull in a way that it allowed you to quickly find parts of the AI's behaviour that you wanted to edit. For example if you needed to adjust how the AI's eyesight works while it's chasing the player, all you had to do was to visually identify the correct task and edit the code it contained. Again contrasting this to a non-visual approach, where locating the correct lines of code can take considerably longer.

6.1.4 Utilizing prebuilt tasks

As Behaviour Designer offers a library of prebuilt tasks that you can use, this resulted in not having to write custom tasks for navigation, eyesight and proximity detection. Luckily these tasks had support for 2D projects, so the initional fears of custom tasks only working in a 3D project did not come true.

Another positive is that although these tasks wouldn't have been extremely difficult to create for someone who knows programming, it also meant that it might be entirely possible to create a simple AI for a basic game completely from these prebuilt tasks, without having any programming skills.

6.1.5 Sequences

Creating actions for the AI that should happen in a sequence one after another turned out to be extremely simple and effective. Being able to edit these sequences by dragging new tasks between the already existing ones proved that the system can easily be expanded in the future without having to worry about breaking to original AI logic. For a simple example of an attack sequence, see figure 6.

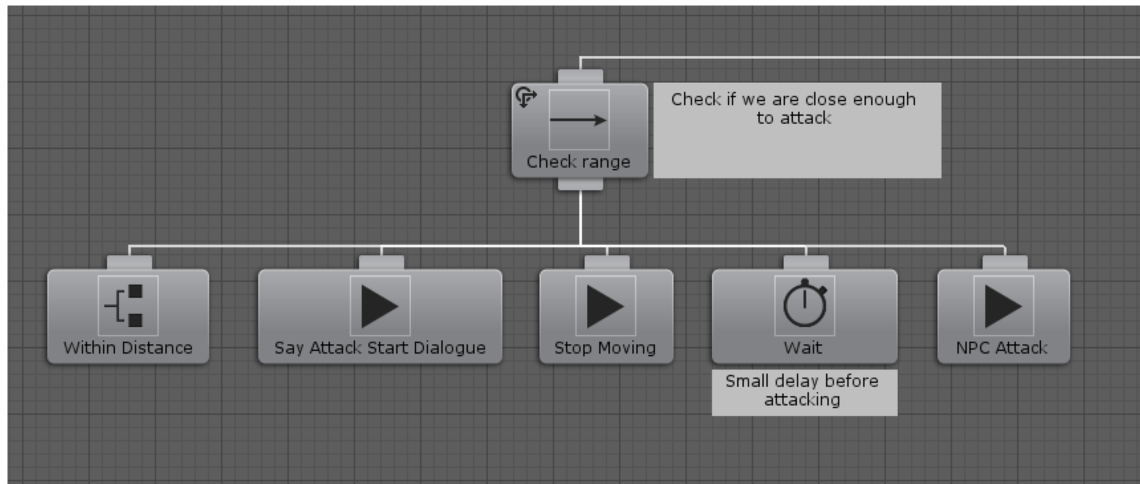


Figure 6. Sequence for cheking target distance and starting attack functions.

6.1.6 Debugging

Behaviour Designer offers a runtime view for each behaviour tree currently active in the scene. This proved to be a very effective way of visually debugging and keeping track of the different AI tasks. Tasks that would return success would be highlighted in green, therefore it was easy to understand what state the AI was currently in. Example of the runtime editor view can be seen in figure 7.

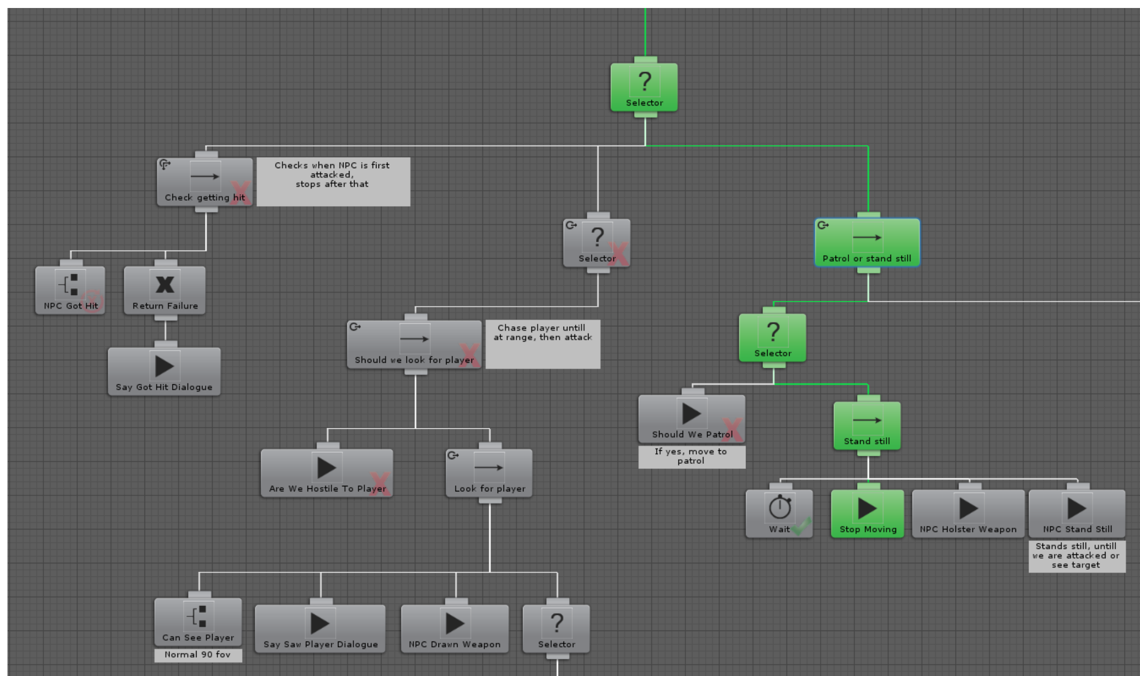


Figure 7. Runtime view of Behaviour Designer.

Compared to a completely script based state machine AI, where the main ways of debugging is setting up breakpoints in code or printing out statements into the console, the visual nature of Behaviour Designer proved to be a much faster and user friendly way of debugging the AI, since you always knew if the unwanted actions of the AI were the cause of being in the wrong state at the wrong time, or a certain state not working as intended.

6.2 Challenges

6.2.1 Conditional aborts and composites

One of the largest obstacles to overcome was understanding how Behavior Designers conditional aborts work. Conditional aborts are used to determine how often a composite task should re-evaluate itself. Meaning should the chosen task change in the future, if the prequisites for that task itself change from what they currently are.

Main problem with these was the non-descriptive naming of the abort type variables. Since the different abort types of None, Self, Lower Priority and Both (as seen in figure 8) are not in themselves very descriptive of the actual functionality when a task should be reevaluated or not, adding of adjusting them in the editor almost always resulted in having to go back to the documentation.

Further problem was that the documentation itself was not very user friendly to newcomers, which meant that the fastest way to get the conditional aborts working as intended was to basically test all the different types while the game was running and see which abort type was the one needed. This was especially the case if you had not edited the conditional aborts for some time, since their logic is very different from other programming tasks, it was easy to forget how each abort type worked.

Figure 8. Conditional Abort types (Opsive 2020).

6.2.2 External behaviour trees

The original plan was to divide tasks into external behaviour trees, that would each handle a very specific AI logic. For example one external tree would handle patrolling between waypoints in the game world, where as another would focus on target selection. The finalized version of the AI would then take and combine these external behaviour trees into one larger tree at runtime.

While in theory this would be a very organized and efficient way of handling the behaviour trees, the development highlighted a few issues with this approach. First problem was passing references to each external behaviour tree. Since the external behaviour trees

wouldn’t be instantiated by Behaviour Designer untill runtime, you ran into the issue of the reference script not finding the external behaviour trees.

One solution would have been to make each task get their needed references from the reference script individually. The downside to this is that certain tasks would require the AI gameobject to have a specific script component, which would limit the reusage of said tasks.

Therefore the chosen option was to make the different AI types consist of one complete behaviour tree which would have all the needed tasks.This resulted in not being able to quickly re-use the planned external behaviour trees, but having to copy paste individual tasks and selectors from one behaviour tree to the other. Another downside to this is that if in the future the logic for example the targeting behaviour needs to be changed, then these changes would needed to be made manually into each AI type. Overall this was decided to cause less problems in the future, compared to having the individual tasks be dependent on specific script components.

These problems also tie in with the performance of Behaviour Designer, which we will discuss next.

6.2.3 Performance

Another reason for giving up on external behaviour trees was the overall performance of Behaviour Designer.

Opening a behaviour tree for editing caused frequent freezes, making Unity completely unusable during this time. These freezes seemed to not be effected by the number of tasks or overall complexity of the opened behaviour tree and could last anywhere from two to ten seconds. This time would quickly add up during development, when each external behaviour tree would need to be closed and opened multiple times to adjust their tasks and logic flow.

It is advisable to note that these performance issues might not happen for everyone and might not occur in future versions of Unity or Behaviour Designer.

6.2.4 Visual Bugs

Behaviour Designer also had atleast one very frequent visual bug with viewing the chosen behaviour tree at runtime. Viewing a tree that consist of many tasks would cause some of them to visually overlap, resulting in having to drag them to their correct places with the mouse each time, as seen in figure 9.

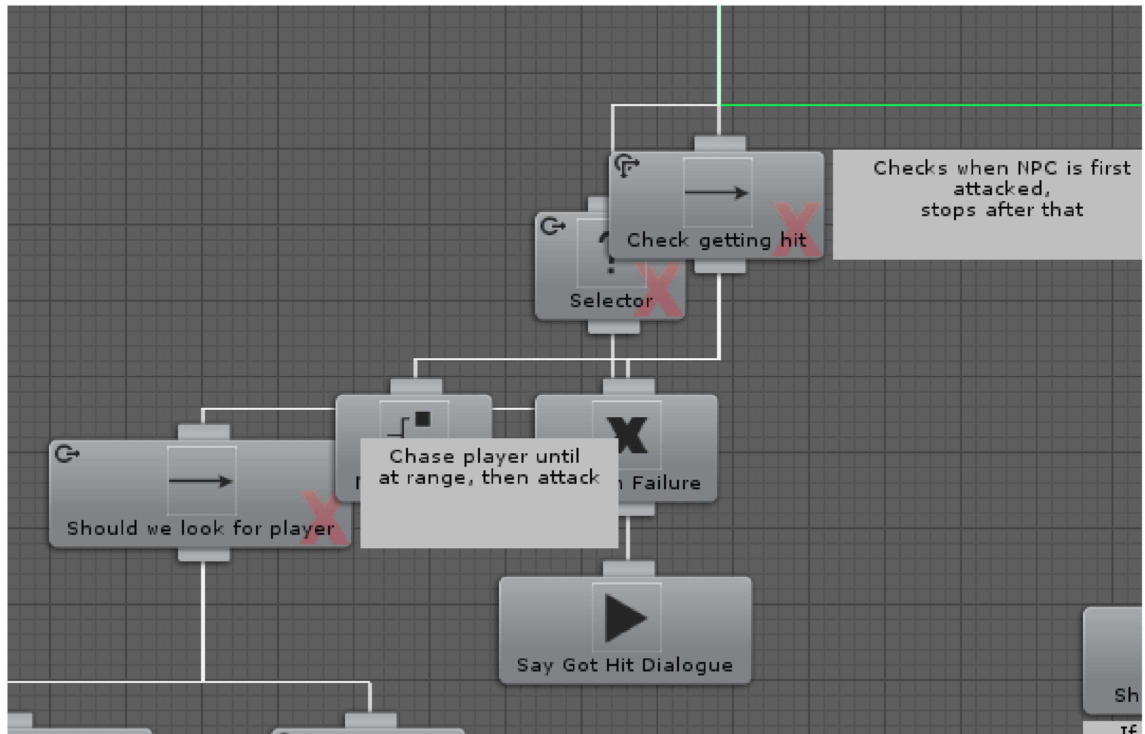


Figure 9. Visual bug of overlapping tasks.

7 CONCLUSION

7.1 Suitability

After developing the AI using a behaviour tree and comparing the development process to a more traditional approach with state machines, it is safe to say that both approaches have their uses in different scenarios.

Due to the fact that a substantial amount of time had to be spent on becoming familiar with the logic flow of behaviour trees and how to tie it together with already existing code structure, developing the same functionality with a state machine might not have been as time-consuming. Having mentioned this, it should be taken into account that the time required to understand the basics of the logic flow and tree traversal mechanics will vary from person to person. For someone with previous experience with behaviour trees and state machines these results might not be accurate.

7.2 Reliability of results

Reflecting on the positive aspects of behaviour trees, the results and advantages should be easily reproducible, with the expectation that the requirements for the AI would stay the same.

When it comes to the challenges encountered, it is safe to assume they are not the case for every user. The perfomance and bugs of Behaviour Designers editor might be tied to a certain computer, other third party tools in use or the current versions of Behaviour Designer and Unity.

It should also be taken into consideration that since the Unity Asset Store consists of many different behaviour tree tools, some of them might not share the same advantages or challenges as those of Behaviour Designer. Starting out with a free asset such as Behaviour Bricks might be a good way to prototype and become familiar with the workflow of a behaviour trees, without having to purchase a more expensive third party software, such as Behaviour Designer.

7.3 Future uses and projects

Although there were multiple pre-built tasks to use, it should be noted that most of these were mostly suited for a very specific AI, in this case most of them were based around movement and actions such as eyesight and distance checking. For example, AI in a cardgame would not find these pre-built tasks very useful, which would result in the developers having to write more custom tasks. Therefore becoming familiar with the different tasks that Behaviour Designer comes with through documentation should help in figuring out how much development time can be saved by the usage of these tasks.

It would also be beneficial to evaluate the potential of behaviour trees in the future for handling more than just AI. For example, it would be possible to handle all player input inside a behaviour tree with customs tasks. This would have helped in unifying the base character class and have resulted in not having to spend as much time worrying about different player states when deciding which player inputs should be accepted.

In the end, developers should base the decision of what AI development tools to use on the scope of the project and the AI requirements. For a simple AI that does not require multiple different states and transitions, it might be better to use traditional state machines and build ther code around them. This is especially true in case i there is no need to expand the AI's functionality in the future, since the advantages of fast reorganizion and reusage of tasks in behaviour trees cannot be utilized to their full potential. On the other hand, in projects where the AI needs to accommodate many changing states, behaviour trees offer a very effective way of creating large and complex systems, without having to spend extensive time worrying about state transitions.

REFERENCES

Simson, C. 2014. Behaviour tree for AI: How they work. Consulted 27.4.2020. Available: https://www.gamasutra.com/blogs/ChrisSimpson/20140717/221339/Behavior_trees_for_AI_How_they_work.php

Bevilacqua, F. 2013. Finite-State Machines: Theory and Implementation. Consulted 27.4.2020. Available: https://gamedevelopment.tutsplus.com/tutorials/finite-state-machines-theory-and-implementation--gamedev-11867

Lou, H. 2017. AI in Video Games: Towards a More Intelligent Game. Consulted 30.4.2020. Available: http://sitn.hms.harvard.edu/flash/2017/ai-video-games-toward-intelligent-game/

Opsive. Behaviour Trees or Finite State Machines. Consulted 26.4.2020. Available: https://opsive.com/support/documentation/behavior-designer/behavior-trees-or-finite-state-machines/

Opsive. Conditional Aborts. Consulted 26.4.2020. Saatavilla sähköisesti osoitteessa: https://opsive.com/support/documentation/behavior-designer/conditional-aborts/

Unity Technologies 2018. State Machine Transitions. Consulted 26.4.2020. Available:

https://docs.unity3d.com/2018.2/Documentation/Manual/StateMachineBasics.html

Unity Technologies 2018. State Machine Transitions. Consulted 26.4.2020. Available: https://docs.unity3d.com/2018.2/Documentation/Manual/StateMachineTransitions.html