Bachelor’s thesis

Information Technology

Game Technology

2017

Joonas Rikkonen

IMPLEMENTING A FLEXIBLE ARTIFICIAL INTELLIGENCE SYSTEM FOR A VIDEO GAME

– case Northbound

TURKU AMK

TURKU UNIVERSITY OF APPLIED SCIENCES

BACHELOR’S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Information Technology

2017 | 32 pages

Joonas Rikkonen

IMPLEMENTING A FLEXIBLE ARTIFICIAL INTELLIGENCE SYSTEM FOR A VIDEO GAME

- case Northbound

There are countless approaches to the implementation of artificial intelligence in video games. Nowadays there is a wide range of AI-related tools and extensions available for many of the popular game engines, and as such game developers do not necessarily have to develop their own AI solutions from the ground up. Utilizing third-party tools may save a large amount of development time, but a pre-made general-purpose tool may also often be less applicable than a custom tool designed specifically for the project at hand.

The objective of the thesis was to design and implement an artificial intelligence system for Northbound, an action role playing game being developed by FakeFish Ltd. The system was created by combining and modifying existing AI tools for the Unity game engine. One of the main requirements of the system was a visual editor that could be used to create and modify AI behaviors without the need to write any actual programming code.

During the development process, several types of AI architectures and tools designed for game AI development were reviewed in order to evaluate their applicability for the game project. A tool called Behavior Designer was chosen as the basis of the system, and a number of modifications and custom additions were made to supplement its feature set.

The end result of the project was a workable system that lowered the barrier to modify and create AI behaviors for the members of the development team with little to no programming experience.

KEYWORDS:

Unity, game programming, artificial intelligence, visual editor

OPINNÄYTETYÖ (AMK) | TIIVISTELMÄ

TURUN AMMATTIKORKEAKOULU

Tietotekniikka

2017 | 32 sivua

Joonas Rikkonen

JOUSTAVAN PELITEKOÄLYJÄRJESTELMÄN IMPLEMENTOINTI

- case Northbound

Videopelien tekoälyhahmojen toteuttamiseen on olemassa lukemattomia eri lähestymistapoja. Moniin suosittuihin pelimoottoreihin on saatavilla lukuisia erilaisia tekoälytyökaluja ja laajennuksia, joiden ansiosta pelinkehittäjien ei välttämättä tarvitse kehittää tekoälyratkaisuitaan alusta asti itse. Valmiita työkaluja hyödyntämällä on mahdollista säästää huomattavasti kehitysaikaa, mutta ongelmaksi saattaa kuitenkin muodostua työkalujen puutteellisuus tai huono soveltuvuus kehitettävään peliin.

Opinnäytetyön tavoitteena oli suunnitella ja kehittää tekoälyjärjestelmä FakeFish Oy -yrityksen Northbound-peliprojektin tekoälyhahmojen toteuttamista varten. Järjestelmä rakennettiin yhdistelemällä ja muokkaamalla olemassa olevia Unity-pelimoottorin tekoälytyökaluita tavoitteena saada aikaan kokonaisuus, joka yhdistää eri työkalujen parhaat puolet ja täydentää niitä kehitettävän pelin tarpeiden mukaan. Yksi päätavoitteista oli se, että tekoälyhahmoja pystyy luomaan ja muokkaamaan yksinkertaisen graafisen käyttöliittymän avulla ilman varsinaisen ohjelmointikoodin kirjoittamista.

Kehityprosessiin aikana arvioitiin useiden tekoälyjen kehitykseen suunniteltujen Unity-pelimoottorin työkalujen sekä erityyppisten tekoälyarkkitehtuureiden soveltuvuutta Northbound-projektin tarpeisiin. Tämän lisäksi arvioinnin perusteella valitut työkalut integroitiin peliprojektiin ja niiden puutteita täydennettiin ohjelmoimalla järjestelmään muutamia uusia ominaisuuksia.

Työn lopputuloksena valmistui käyttökelpoinen järjestelmä, joka mahdollistaa monipuolisten tekoälykäyttäytymisten luomisen graafisen käyttöliittymän avulla.

ASIASANAT:

Unity, peliohjelmointi, tekoäly, visuaalinen editori

CONTENT

APPENDICES

FIGURES

TABLES

LIST OF ABBREVIATIONS AND TERMS

3D	Three-dimensional
Agent	An autonomous entity controlled by an artificial intelligence
AI	Artificial intelligence
API	Application programming interface, a set of precisely defined methods of communication between applications of components of an application
BT	Behavior tree
C#	A general-purpose, object-oriented programming language developed by Mircosoft
FSM	Finite state machine
Game engine	A software framework designed for the development of video games
NPC	Non-player character – any game character controlled by the computer instead of the player
Unity	A cross-platform game engine developed by Unity Technologies
Unreal Engine	A cross-platform game engine developed by Epic Games

1 INTRODUCTION

A game is a system in which players engage in an artificial conflict, defined by rules, that results in a quantifiable outcome (Salen & Zimmerman 2003). Games can take a variety of forms, from sports to board games and card games. One of the newest forms of games, video games, shares many characteristics with the more traditional types of games, but often includes a feature that is mostly absent from other types of games: artificial intelligence.

The term "artificial intelligence" could be defined as the capability for a machine to mimic the cognitive functions of a human mind, such as learning and problem solving (Russel & Norvig 2009, 2). In a video game context, artificial intelligence usually means emulating the behavior of non-player characters or other in-game entities. Since the goal of AI in games is only to *simulate* intelligent behavior or provide the player with a challenge that the player can overcome, AI for games can be considered more "articifial" and less "intelligence". (Kehoe 2015)

Most video game AIs need to be able to handle a wide range of tasks. Some type of decision-making logic is essential, but in many games the AI agents may also need to be able to navigate complex 3D environments, perceive and react to their surroundings, control the animations of a 3D model, cooperate with other agents or various other tasks. There are many different techniques to implementing game AIs, but perhaps the most common approaches are systems based on finite-state machines or behavior trees. The A* graph traversal algorithm is also worth a mention, being by far the most common pathfinding solution used in games (Stout 2000).

Even in the context of game AIs which only intend to create an illusion of intelligent behavior, artificial intelligence is a complex subject, and as such the tools that are used to develop game AIs are often complex as well. This is especially true when it comes to full-fledged AI engines that offer a wide range of different features. On the other hand, the problem with smaller and simpler tools is that they may be too restrictive or only applicable in very specific use cases.

An additional degree of difficulty is added by the fact that AI development often requires the ability to use a programming language, which may prevent game designers or other game developers with limited programming skills from being able to create or modify AI behaviors. However, by utilizing a tool that presents the AI logic in a visual manner, for example a flowchart- or tree-like diagram, this issue can be remedied to some extent.

This thesis will describe the development of an AI solution for an action role playing game developed with the Unity game engine. The goal was to build a system that is flexible and powerful, while still being easy to use even for developers with little to no programming skills.

1.1 Aim and objectives of the study

The aim of this thesis was to develop of an AI system for Northbound, an action role playing game being developed by FakeFish Ltd with the Unity game engine. The system will be used to control the various non-player enemy characters in the game. The feature requirements of the system include pathfinding, steering logic, reacting to the environment and to the actions of the player and controlling the animations of the character.

The AI system should be flexible and the enemy behaviours easily modifiable, preferably without the need to write or change any actual programming code. The aim is to offer the AI designers a versatile system that gives them the freedom to create a wide range of different AI behaviors, while still keeping the tools simple to use without extensive programming skills or knowledge on AI theory.

Several types of AI architectures were considered for the system, including finite state machines, utility-based AIs and behavior trees. Some of the benefits and drawbacks of these different architectures are reviewed in the following theory chapter.

There is a wide range of AI-related third-party tools and extensions available for Unity, which is why it was deemed unnecessary to develop all the features of the AI system in-house. Instead, a set of existing tools were selected as the basis of the system. Several AI tools were researched and evaluated as a part of this thesis to find the most suitable ones.

1.2 Methods

The first step in the implementation of the system was to examine various AI-related third-party tools available for the Unity engine. The tools were selected from the Unity Asset Store, a marketplace where developers and artists sell content designed for the Unity Engine. With over 15,000 free and paid-for 3D models, editor extensions, scripts, shaders, materials, audio files and animations available for download, the Unity Asset Store has become the most popular place to get third-party assets for Unity projects (Unity Technologies 2017a).

Due to the sheer number of AI-related tools in the Unity Asset Store, only a small portion of them were chosen for closer examination based on their popularity, i.e. the number of downloads and the rating on the Asset Store. The selection was also narrowed down by ruling out tools clearly not suitable for the project, such as ones designed specifically for 2D games or first person shooters.

When selecting the tools for the final system, one of the most important criteria was that feature sets of the tools are extensive enough to implement the AI behaviors in Northbound. The quality of the documentation and support available for the tools were also key factors. Personal preferences of the members of the development team and past experiences with various AI tools also affected the selection. The team had mostly used an AI engine called RAIN AI in their previous projects, and consequently the tools with a similar workflow and feature set as RAIN AI had a softer learning curve than tools with a very different approach to AI development.

The tools were tested internally and an evaluation matrix was composed to compare their strengths and weaknesses.

2 THEORY

Luckily for game developers, it is not always necessary to program all the aspects of a game AI from scratch. Many game engines, software frameworks designed for the development of video games, offer some pre-made tools that can be utilized in the creation of game AIs. Most popular game engines such as Unity or Unreal Engine can also be expanded by taking advantage of the many third-party extensions available for them; some smaller tools focused on a subsection of game AIs such as navigation or decision-making, some all-inclusive “AI engines” that aim to provide all the features needed to implement a game AI.

2.1 Unity

Unity is a cross-platform game engine – a software framework and a set of development tools designed for the creation of video games. Since its initial release in 2005, Unity has grown to be on of the most popular game engines, used by both hobbyists and professional game studios (Unity Technologies 2017b).

Unity is developed by Unity Technologies. The engine comes with four license options: Personal, Plus, Pro and Enterprise. All four licenses, including the free Personal license, give the developer full access to all features of the engine. However, only users whose annual revenue is under $100,000 are allowed to use the Personal license. (Unity Technologies 2017c)

Unity boasts a large number of features out-of-the-box, but its AI-related features are very limited. Unity’s built-in navigation system and its animation system Mecanim can be useful in AI development, but implementing any sort of decision-making logic requires writing custom code or utilizing third-party Unity extensions.

2.2 Common game AI architectures

2.2.1 Finite state machine

A finite state machine (FSM), or simply a state machine, is a mathematical model of computation consisting of states, transitions and the conditions for each transition.

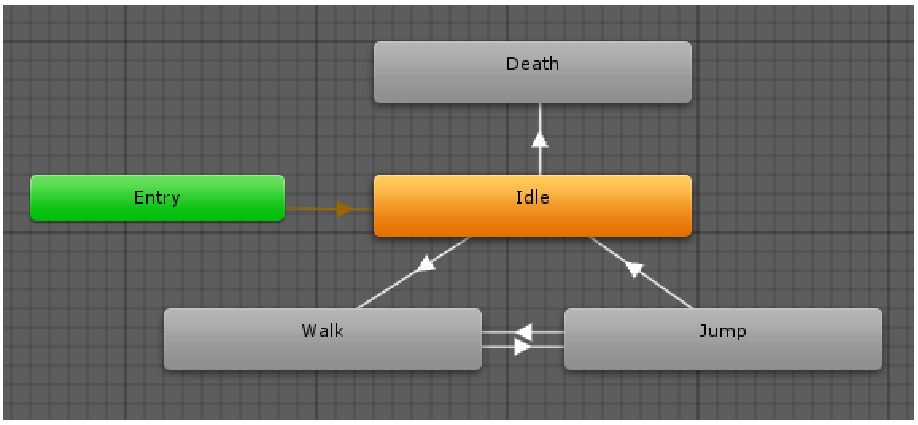


Figure 1. Example of a finite state machine: a simple animation controller made with Unity.

In game AI context, each state is usually a specific action or behavior of an AI agent. For example, a very simple FSM AI could consist of an idle state and an attack state. The idle state may have logic that checks whether an enemy is within sight, and if so, a transition to the attack state is triggered.

The simple, flowchart-like structure of finite state machines makes them an easy method of modeling the behavior of an AI agent, even for game designers with no programming knowledge. (Rabin 2012, 71)

However, finite state machines have some drawbacks that may make them inconvenient for some use cases. As the number of states increases, the number of potential transitions between states increases rapidly. For instance, if a FSM consists of four states and each state needs a transition to all other states, the number of required transitions is 12. A similar FSM with five states would require 20 transitions and six states would merit 30. While this may not be a problem in a simple AI with a small number of states and a limited number of transitions between them, a more complex AI could have dozens of states, making the FSM extremely convoluted and hard to maintain. (Mark 2012)

Another issue is the potential workload involved in adding a new state to the FSM. Every other state that could potentially transition to the new one needs a new transition, and the logic that triggers said transition. (Mark 2012)

2.2.2 Behavior tree

Behavior tree (BT) is a hierarchical tree of nodes used for modelling decision-making logic. The nodes are classified as root nodes, control flow nodes or execution nodes (tasks). The execution of a BT starts at the root node, progressing to its child nodes. Control flow nodes determine which child nodes are executed – they could be seen as analogous to the transitions in a finite state machine. The leaves of the tree, the execution nodes, are similar to the states in a FSM: they carry out a specific task or action.

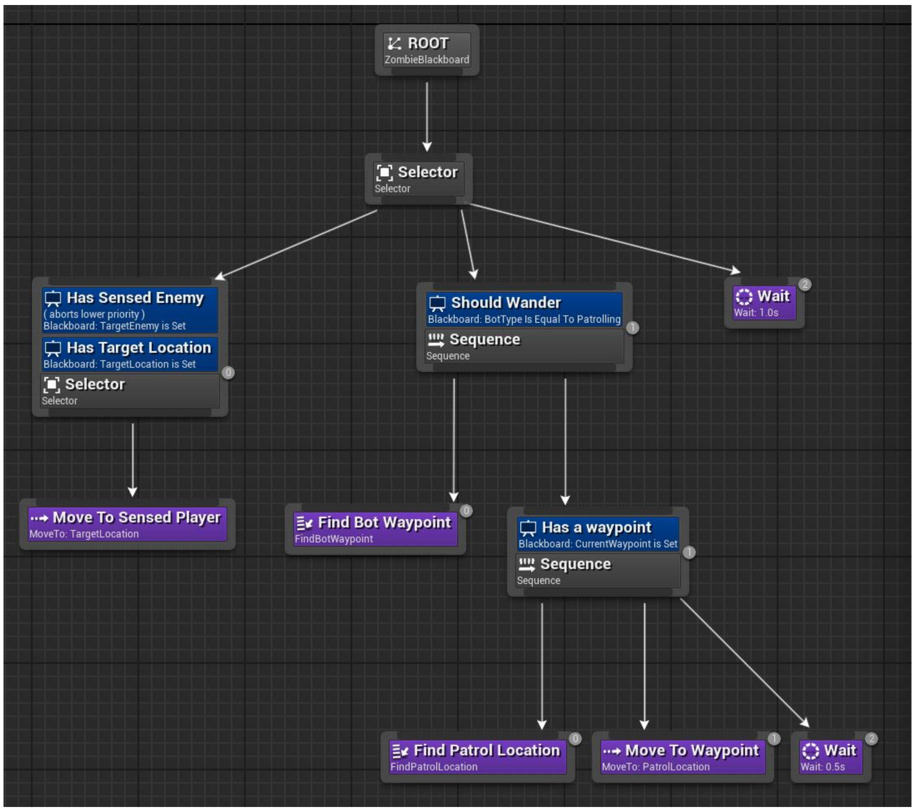


Figure 2. An example of a behavior tree created with Unreal Engine’s Blueprint system (Epic Games 2017).

The advantage of a BT compared to a finite state machine is that the states are separated from the decision making logic. While a FSM with four states may need up to 12 transitions, a similar BT could be done with just one control flow node that determines which of the four states to execute. (Mark 2012)

Control flow nodes are also more flexible than simple transitions between states. For example, a control flow node could execute multiple nodes in parallel, execute a set of nodes sequentially, evaluate the priority of the child nodes and execute the most high-priority one or choose a node randomly. (Simpson 2014)

2.2.3 Utility-based AI

Unlike finite state machines or behavior trees, utility-based systems do not have rigid, predetermined rules for what to do and when. Instead, potential actions are given a utility value, a numerical value that represents how suitable an action is for the situation at hand, and the action with the highest utility value will be chosen.

An example of a utility-based AI system are the AI characters in The Sims franchise. Each potential action is scored based on a combination of an agent’s current needs and the ability of that action to satisfy the needs. The Sims is also a good example of the type of game where utility-based systems are a natural choice: they are more appropriate in situations where there is a large number of potential actions the agent can take, and when there is no obvious ”right answer”. (Mark 2012)

While the lack of predetermined rules can be desirable in some types of games, it also makes it difficult to intuit what will happen in a given situation. While a behaviour tree might have precise rules as to what an agent should do when specific conditions are met, the decision-making logic of a utility-based system is inherently more fuzzy. It may be difficult to design the utility value calculations in a manner that causes the agent to act exactly the way the designer intended in all situations. (Mark 2012)

2.3 AI tools for Unity

2.3.1 RAIN AI (v2.1.18.0)

RAIN AI is a free AI engine developed by Rival Theory Inc. It includes a visual behavior tree editor, pathfinding and steering systems, animation support, a perception system and several other features. It is one of the most popular AI assets in the Unity Asset store and used in a wide range of commercially successful games made with Unity. (Rival Theory 2017)

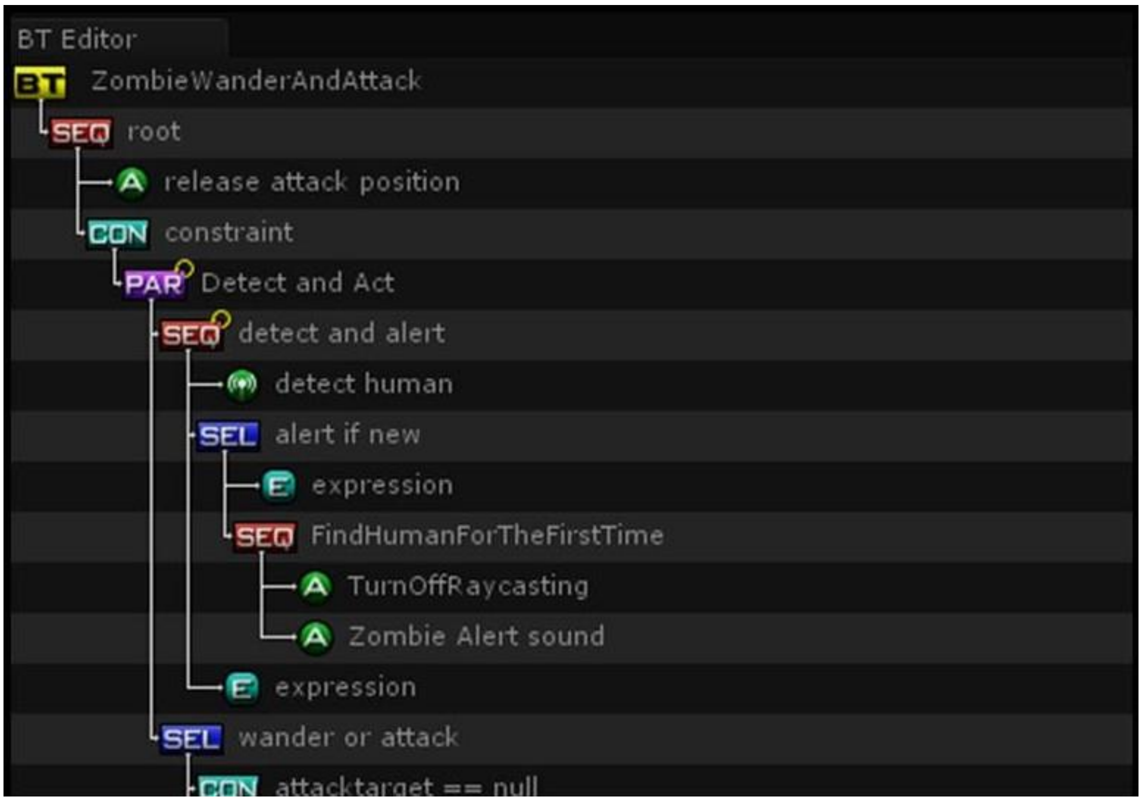


Figure 3. Example of a behavior tree created with RAIN's behavior tree editor (Rival Theory Inc 2017).

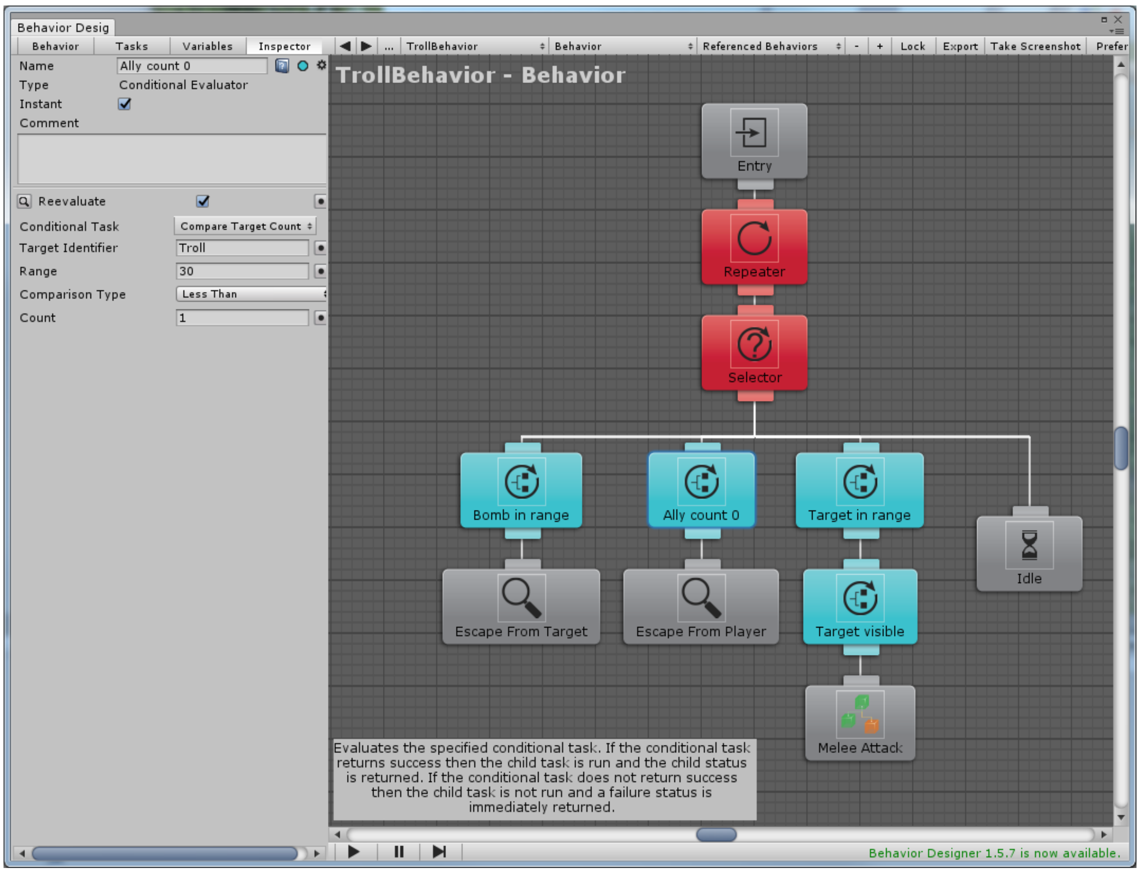
RAIN AI is also highly customizable: it is possible to program new components to the system, including custom behavior tree nodes, sensors and movement systems.

RAIN AI is not open source software, which prevents the users from doing changes to any of the core systems that run the behavior trees. It also means that the users have to rely on Rival Theory to keep the engine up to date as new versions of Unity are released and changes are made to the Unity API. This is may become an issue in the near future, since the future development and support of RAIN is uncertain as of early 2017. Rival Theory has removed all references to RAIN from their company website and moved RAIN-related content to a ”legacy” subdomain. They have also announced that RAIN will be replaced by their upcoming product Sentio. (Rival Theory 2017)

RAIN AI is quite thoroughly documented and Rival Theory hosts an active discussion forum where users can request help or share their own custom components.

2.3.2 Behavior Designer (v1.5.9)

Behavior Designer is a paid Unity extension by Obsive that allows creating behavior trees with a visual editor. In addition to the visual editor, the extension includes hundreds of pre-made behavior tree nodes that can be used as the buildings blocks of an AI. The pre-made nodes include composites and decorators which determine the control flow of the tree, and task nodes which mostly interact with Unity’s built-in features or do simple tasks such as variable assignment or basic arithmetic operations. (Opsive 2017a)



Name	Ally count 0
Type	Conditional Evaluator
Instant	✓
Comment	
Reevaluate	✓
Conditional Task	Compare Target Count
Target Identifier	Troll
Range	30
Comparison Type	Less Than
Count	1

Evaluates the specified conditional task. If the conditional task returns success then the child task is run and the child status is returned. If the conditional task does not return success then the child task is not run and a failure status is immediately returned.

Behavior Designer 1.5.7 is now available.

Figure 4. Example of a behavior tree created with Behavior Designer.

Even though the pre-made tasks are somewhat limited, Behavior Designer is highly extendible; new types of nodes can be written using C# and there are multiple add-ons available that expand the selection of task nodes (for example, Behavior Designer Movement Pack which adds tasks related to pathfinding and steering).

Behavior Designer also makes it possible to use behavior trees as nodes in another behavior tree, which allows reusing behavior logic in multiple trees. An example of this functionality can be seen in Figure 4, where the ”TrollBehavior” tree includes a node that executes an external ”Melee Attack” behavior tree.

2.3.3 Playmaker (v1.8.4)

Playmaker is a popular visual scripting tool for Unity. As of early 2017, it is in the top ten of paid asset on the Unity Asset store and has been used in several commercially successful games such as Hearthstone: Heroes of Warcraft and INSIDE. Unlike Behavior Designer, it is not designed specifically for game AIs, but as a general-purpose tool that can be used as a substitute for actual gameplay code. Another major difference to Behavior Designer is that Playmaker uses finite state machines instead of behavior trees. (Hutong Games 2017)

Playmaker can also be used to create hierarchical state machines, which are essentially finite state machines where an individual state can be another state machine. This makes it possible to reuse logic by creating state machines that can be used as building blocks in multiple state machines across the game project, and makes larger state machines easier to manage.

Despite not being designed specifically for AI development, Playmaker includes several pre-made actions that can be utilized in game AIs. These include actions such as controlling the animations of a 3D model, moving objects around, visibility checks, manipulating physics objects and playing audio. In addition to the pre-made actions, the users can create their own custom actions, and many popular third-party Unity extensions include actions for Playmaker and advertise Playmaker support. (Hutong Games 2017)

2.3.4 Behavior Bricks (v0.2)

Behavior Bricks is a free Unity extension that can be used to create behavior trees with a visual editor. It is very similar to Behavior Designer, the most apparent differences being the user interface of the behavior tree editor and the number of pre-made behavior tree nodes included in the extension.

While the interface of Behavior Bricks is perhaps more visually pleasing than Behavior Designer’s, it has a few quirks that make it slightly more cumbersome to use. Only one of the nodes in the behavior tree can be selected and moved at a time, which makes reorganizing larger trees more time-consuming as every node has to be moved individually. Another quirk is that decorator nodes cannot be moved; they are always automatically placed between their child and parent nodes, which may make the organization of the tree more difficult.

Behavior Bricks includes 18 pre-made control flow nodes and 32 task nodes, which mostly consist of actions for moving the agent, setting variable values and finding game objects or components from the scene. The nodes are much more limited than the ones included with Behavior Designer, and most likely inadequate for anything but the simplest AI behaviors.

The support and documentation of Behavior Bricks are also somewhat limited compared to Behavior Designer. The support website of Behavior Designer includes extensive documentation of all the features and pre-made nodes included in the extension, dozens of sample projects and an active discussion forum with tens of thousands of posts, while Behavior Bricks is limited to a 15-page quickstart guide and an 18-page guide to programming new behavior tree nodes.

All in all, Behavior Bricks could be seen as a free, but more limited alternative to Behavior Designer.

2.3.5 Apex Utility AI (v1.0.6.1)

Apex Utility AI has one major difference to all the other evaluated tools: it is designed for utility-based AIs. It is based on five types of ”building blocks” which are used to define the behavior of an AI agent: qualifiers, scorers, actions, selectors and contexts. (Apex Game Tools 2017)

Qualifiers calculate a score that represents the utility/usefulness of an action. This is done using scorers, which return a value based on some condition. For example, a scorer could return a value of 100 when an enemy is within some specific range. A qualifier can consist of several scorers, meaning that the agent can take several factors into account when determining which action to execute.

Selectors select the best qualifier from the qualifiers attached to it. Usually this means selecting the qualifier with the highest score, but it is also possible to use other types of decision-making logic, such as selecting the first qualifier that scores above zero.

Context is described as ”the information available to the AI when calculating the scores” in the official Apex Utility AI documentation. In practice, contexts are classes that store the memory, sensory input and any other information an AI agent needs to make decisions. Creating and modifying contexts is done by writing C# scripts, and as such using Apex Utility AI effectively requires some degree of programming skills.

The selection of actions included with Apex Utility AI is somewhat limited, but there are paid extensions available in the Asset Store that expand its functionality.

3 DEVELOPING THE AI SYSTEM

3.1 Northbound

Northbound is an action role-playing game with strong comedic elements. The story and setting of Northbound are inspired by Nordic mythology, in particular the Finnish national epic Kalevala. A key mechanic of the game is the ability to switch between three playable characters: Lemminkäinen, a warrior armed with a bow and a lasso, Ilmarinen, a blacksmith who can build various kinds of contraptions and Väinämöinen, an old shaman who can control the nature with his voice. Each character has their strengths and weaknesses, and the player has to choose whoever is the best suited for a given puzzle or a battle. Story-wise the character switching mechanic is a result of a magical spell that binds the heroes together into one body, and the ultimate objective of the game is to find a way to break the spell.

Figure 5. Ilmarinen, Väinämöinen and Lemminkäinen, the three playable characters of Northbound (FakeFish Ltd 2017).

Northbound has been in development since 2014, but in early 2017 the project underwent some major changes. The story, setting and gameplay mechanics were redesigned, and a vast majority of the assets and gameplay code made so far were scrapped. This provided the development team an opportunity to reflect on the technical issues and inadequacies of the tools previously used in the development, and come up with a more effective toolset for the development of the redesigned game. This included replacing the now-deprecated RAIN AI with another AI system.

3.2 Requirements

Northbound will feature three playable characters which can be switched between at will. Each character has a different set of abilities which are effective against some types of enemies and ineffective against some. For instance, one of the characters is powerful against individual, strong ”tank-like” enemies but has difficulty dealing with large groups of enemies or fast-moving targets. It is important that there is enough variety between different types of enemies to encourage the player to switch between characters, and that the enemies are balanced in a way that prevents any of the characters from becoming seldomly used or an obvious go-to choice.

Designing a varied but balanced set of enemies will most likely require an iterative approach with considerable amount of small, frequent changes and tuning. Due to this, the enemy behaviours have to be easily modifiable, without the need to write or change any actual programming code, in order to make small modifications easier for the non-programmers of the development team.

The system will need to be able to handle pathfinding and locomotion of the AI agents. A basic perception system is also required, allowing the agents to see, hear and react to the player character and objects in the game environment – or, not to see the an object if something is blocking the line of sight. The ability to control the animations of the agent is also needed.

Another requirement is the support for ”enemy templates” and instances of a template. For example, there could be a generic ”troll enemy” template that determines the default behavior and attributes of a troll. Instances of this template could then be added to the game world, and the behavior of an individual instance would need to be editable without affecting the template. This makes it easier to add some variety and personality to the individual enemies and prevent them from seeming like clones of each other.

This type of functionality can be achieved with Unity’s prefab system, as long as the components of the AI agent are implemented in a way that allows creating new instances and is compatible with Unity’s serialization system.

3.3 Choosing the tools

3.3.1 Architecture

As a starting point for the development of the system, the game designers of the team created a set of specifications for different types of enemy behaviors. An example of the behavior of a common enemy is shown in Table 1. The conditions are evaluated from top to bottom, and the agent will choose the first action whose conditions are met (or the default action at the bottom if none are met).

Table 1. Description of the behavior of a common enemy in Northbound (FakeFish Ltd 2017).

Troll grunt

Conditions	Action
Distance to bomb < 5m	70% flee, 30% kick bomb further
HP < 20% AND Distance to enemy > 10m	Use item (TROLL_HEALING_ITEM)
HP < 30% AND Taking melee damage	Flee (2s)
Enemy visible AND Distance to enemy > 8m	Use ability (Charge)
Enemy visible	Attack enemy (melee)
Default	Idle

Abilities	Effect
Charge	charge toward enemy, inflict 2x damage if hit

Items	Effect
TROLL_HEALING_ITEM	heals 30% max health

This type of behavior logic could be expressed as a finite state machine, but since the agent can switch from any action to another, it would require a transition from each state to all other states. In this somewhat simple example the number of transitions would already be as high as 30, which makes it quite apparent that a FSM is not the most suitable architecture for this use case.

An utility-based system was another potential architecture for the system. However, the conditions for each action are essentially binary – the condition is either met or not, and as such, the concept of numerical utility values that represent the desirability of an action is not a very natural way to model this type of behavior. The first action whose conditions are met should always be selected, which defeats the purpose of the utility value calculations.

Behavior trees were deemed as the architecture of choice for several reasons. First of all, the structure of the example behavior is very similar to a behavior tree; it could be expressed as a tree where each condition is a control flow node, and the parent node of the conditions is another type of control flow node which selects the first child whose conditions are met.

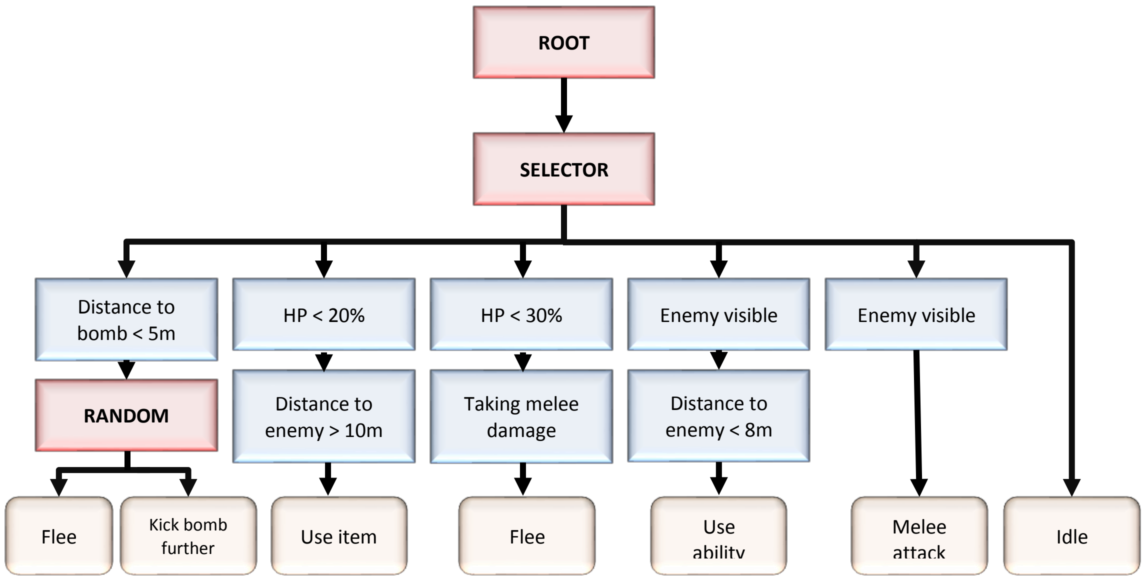


Figure 6. The example behavior in Table 1 expressed as a behavior tree.

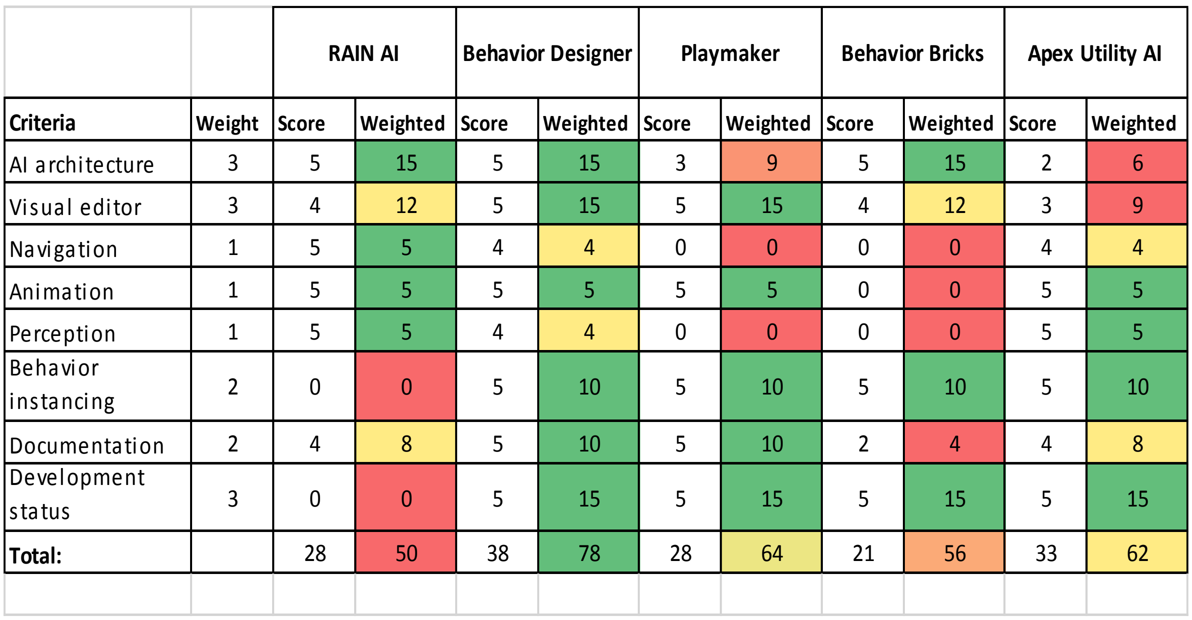
Another reason for the choice of behavior trees is that there are several visual behavior tree editors available for Unity. Several members of the development team are also familiar with behavior trees, having used the RAIN AI engine and its behavior tree editor.

Additional advantage of behavior trees is their flexibility. The control flow does not necessarily have to be driven by conditions as in the troll example – if the need arises, a BT makes it easy to carry out sequences of actions, add randomness to the control flow or execute multiple actions simultaneously.

3.3.2 Third-party tools

A weighted design matrix was composed to estimate the suitability of the evaluated third-party AI tools (see Table 2).

Table 2. An evaluation matrix that compares the different AI tools



		RAIN AI		Behavior Designer		Playmaker		Behavior Bricks		Apex Utility AI	
Criteria	**Weight**	**Score**	**Weighted**	**Score**	**Weighted**	**Score**	**Weighted**	**Score**	**Weighted**	**Score**	**Weighted**
AI architecture	3	5	15	5	15	3	9	5	15	2	6
Visual editor	3	4	12	5	15	5	15	4	12	3	9
Navigation	1	5	5	4	4	0	0	0	0	4	4
Animation	1	5	5	5	5	5	5	0	0	5	5
Perception	1	5	5	4	4	0	0	0	0	5	5
Behavior instancing	2	0	0	5	10	5	10	5	10	5	10
Documentation	2	4	8	5	10	5	10	2	4	4	8
Development status	3	0	0	5	15	5	15	5	15	5	15
Total:		28	50	38	78	28	64	21	56	33	62

The tools were scored based on eight criteria:

1. The AI architecture. Behavior trees were deemed as the most suitable architecture for the system, and so the tools based on behavior trees were given the highest score.
2. Visual editor. The tools with the most easy-to-use visual editor (based on the personal preferences of the development team) were given the highest score.
3. Navigation support. Tools that require the user to buy an additional extension for the navigation features were given a lower score, and tools with no navigation features whatsoever were given a zero.
4. Ability to interface with Unity's built-in animation features.
5. Features that allow the agents to perceive their surroundings, e.g. visibility checks, reacting to sounds, ability to differentiate between objects.

6. Support for creating and modifying instances of a behavior without modifying the original ”behavior template/prefab”.
7. The quality of the documentation.
8. Development status. Tools that are still being actively developed were given a high score, while RAIN AI was given a zero due to the uncertain future of the tool.

The weights were based on the importance of the individual criteria: the AI architecture and the visual editor were given a high weight, because an easy-to-understand architecture and a visual editor were one of the key goals of the AI system. The development status of the tool was also given a high weight, because tools that are no longer in active development have a higher chance of becoming incompatible with newer versions of Unity. The navigation, animation and perception features were considered easy enough to implement using Unity’s built-in features and/or custom code, and were given a lower weight than the rest of the criteria.

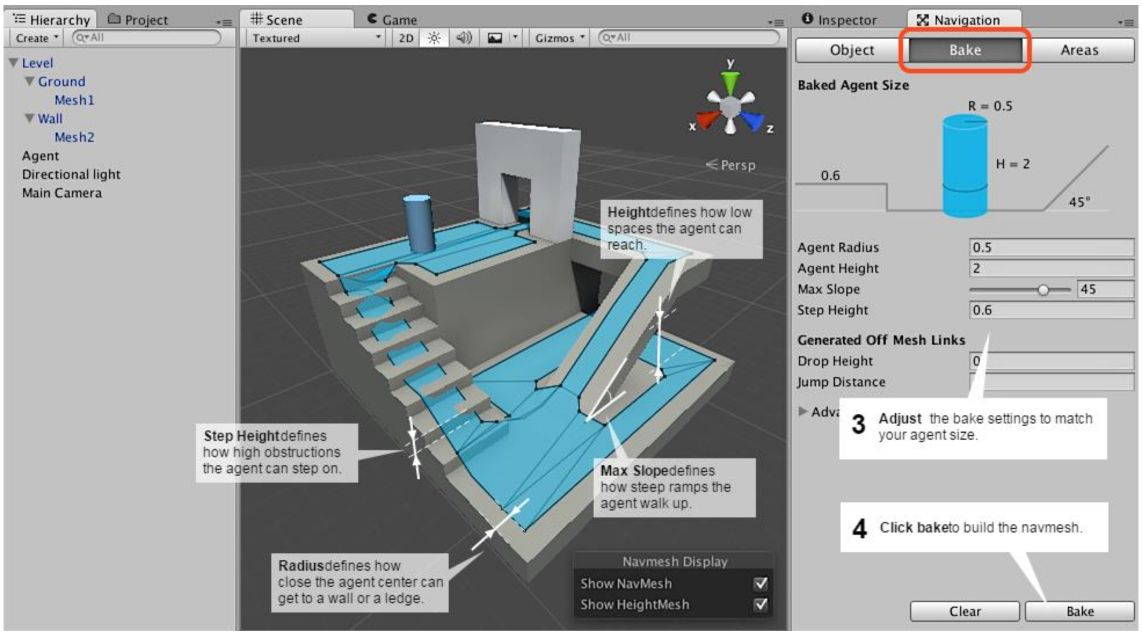
In the end, Behavior Designer was chosen as the basis of the system. Playmaker and Apex Utility AI were ruled out by the decision to use behavior trees, but they also had other drawbacks that affected their score. Playmaker is not designed specifically for AI development and its AI-specific features are very limited. The score of Apex Utility AI was reduced by the complexity of the utility-based AI architecture and the limitations of its visual editor.

RAIN AI was ruled out because its future development looks uncertain, and there is a risk that it will become incompatible with newer versions of Unity. Another issue with RAIN is that it does not allow editing individual instances of a behavior tree, making it difficult to achieve the template/instance functionality.

Behavior Designer was chosen over Behavior Bricks because of its more extensive feature set and high-quality documentation.

3.3.3 Pathfinding

Unity has a built-in navigation system based on navigation meshes and the A* pathfinding algorithm.



Height defines how low spaces the agent can reach.

Step Height defines how high obstructions the agent can step on.

Max Slope defines how steep ramps the agent walk up.

Radius defines how close the agent center can get to a wall or a ledge.

Agent Radius	0.5
Agent Height	2
Max Slope	45
Step Height	0.6

Generated Off Mesh Links

Drop Height	0
Jump Distance	

3 **Adjust** the bake settings to match your agent size.

4 **Click bake** to build the navmesh.

Figure 7. Unity’s Navigation window and an example of a generated navigation mesh.

The navigation system has some obvious limitations. For example, the navigation meshes are automatically generated based on the level geometry and cannot be edited manually. Another limitation is that the pathfinding system is intended to be used with Unity’s ”NavMeshAgent” – a component that allows objects to navigate the game environment and avoid obstacles using navigation meshes. With NavMeshAgent, it is possible to calculate paths asynchronously and choose how many path nodes the pathfinding algorithm processes per frame. However, if the user wishes to use a custom locomotion solution instead of NavMeshAgent, the only available public method for path calculations is a static CalculatePath function of the NavMesh class, which calculates the entire path immediately.

Despite these limitations, the built-in navigation system was deemed suitable for the project: it is easy to use, the team was already familiar with it and the limited functionality was adequate for the purposes of the game.

3.3.4 Locomotion

The movement of the AI agents was handled by Mecanim’s root motion system, one of Unity’s standard features. The idea of the root motion system is to use the movement baked into the animation clips of a 3D model to drive the movement of the agent. For instance, a 3D artist could create a running animation where the model moves forward at a specific velocity, and this movement would be used to move the game object forward.

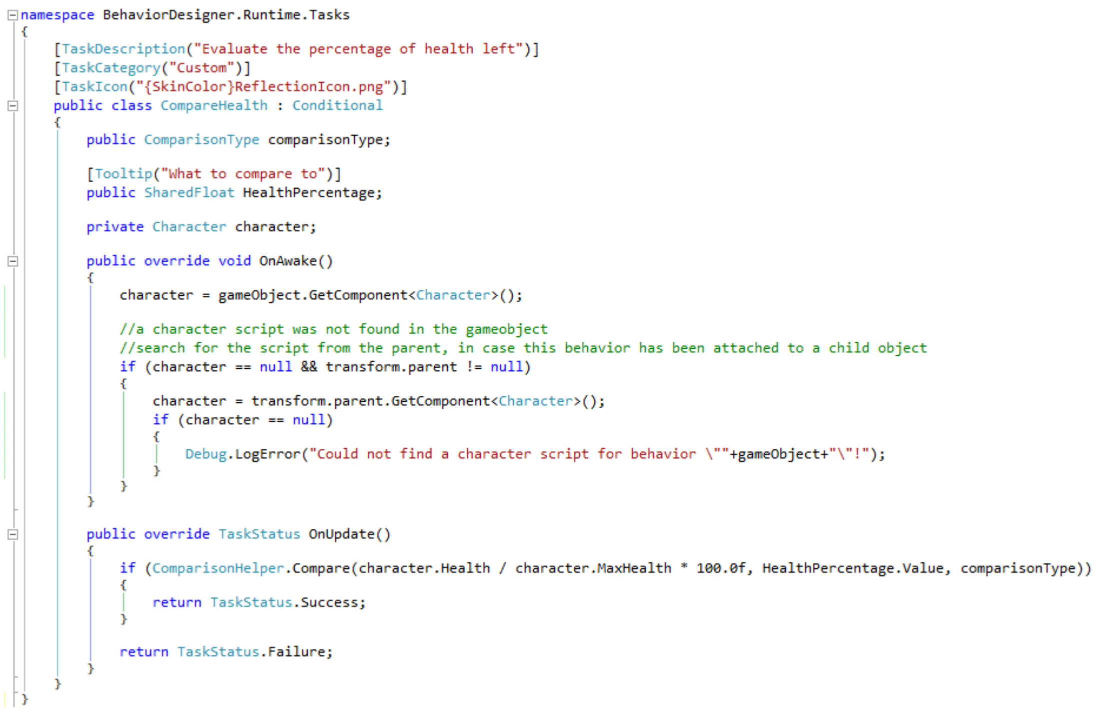
The advantage of the root motion system is that there is no need to adjust the movement speed of an agent to match the apparent movement speed of the animation clips. It also makes it trivial to create non-uniform movement: a limping NPC with an asymmetric gait or a hopping rabbit are easy to implement just by creating a suitable movement animation.

Steering logic was implemented using Behavior Designer Movement Pack, an extension to Behavior Designer that includes behavior tree tasks for several types of movement, for example following a target, fleeing, random wandering and searching for a target.

3.4 Implementation of the system

3.4.1 Behavior Designer

Behavior Designer was utilized almost out-of-the-box with very little modifications to any of its core functionality. Most of the work in getting Behavior Designer ready for use consisted of implementing new custom tasks and modifying some of the existing ones.



```
namespace BehaviorDesigner.Runtime.Tasks
{
    [TaskDescription("Evaluate the percentage of health left")]
    [TaskCategory("Custom")]
    [TaskIcon("{SkinColor}ReflectionIcon.png")]
    public class CompareHealth : Conditional
    {
        public ComparisonType comparisonType;

        [Tooltip("What to compare to")]
        public SharedFloat HealthPercentage;

        private Character character;

        public override void OnAwake()
        {
            character = gameObject.GetComponent<Character>();

            //a character script was not found in the gameobject
            //search for the script from the parent, in case this behavior has been attached to a child object
            if (character == null && transform.parent != null)
            {
                character = transform.parent.GetComponent<Character>();
                if (character == null)
                {
                    Debug.LogError("Could not find a character script for behavior \""+gameObject+"\"!");
                }
            }
        }

        public override TaskStatus OnUpdate()
        {
            if (ComparisonHelper.Compare(character.Health / character.MaxHealth * 100.0f, HealthPercentage.Value, comparisonType))
            {
                return TaskStatus.Success;
            }

            return TaskStatus.Failure;
        }
    }
}
```

Figure 8. A custom conditional task that evaluates the percentage of health a character has left.

Adding new tasks to Behavior Designer is very straight-forward. The custom tasks are implemented in a C# class that inherits from one of four classes: ”Action”, ”Conditional”, ”Decorator” or ”Composite” depending on which type of task it is. The abstract ”Task” base class has several virtual methods that can be overridden in the custom task to implement the actual functionality of the task. The ones that were used most often in the custom tasks written for Northbound were OnAwake, OnStart and OnUpdate. OnAwake is called when the behavior tree is enabled (and can be used in a similar fashion as a constructor), OnStart when the execution of the task starts and OnUpdate when the task is being executed. (Opsive 2017b)

A set of a little over a dozen custom tasks turned out to be adequate to implement the behaviors of the NPCs currently in the game. These included tasks such as evaluating the status of the agent itself (e.g. health), perceiving the environment (see section 3.4.3. Perception) and using or evaluating the state of the combat abilities of the agent. However, additional tasks will most likely need to be added along the development as more specialized and complex behaviors are needed.

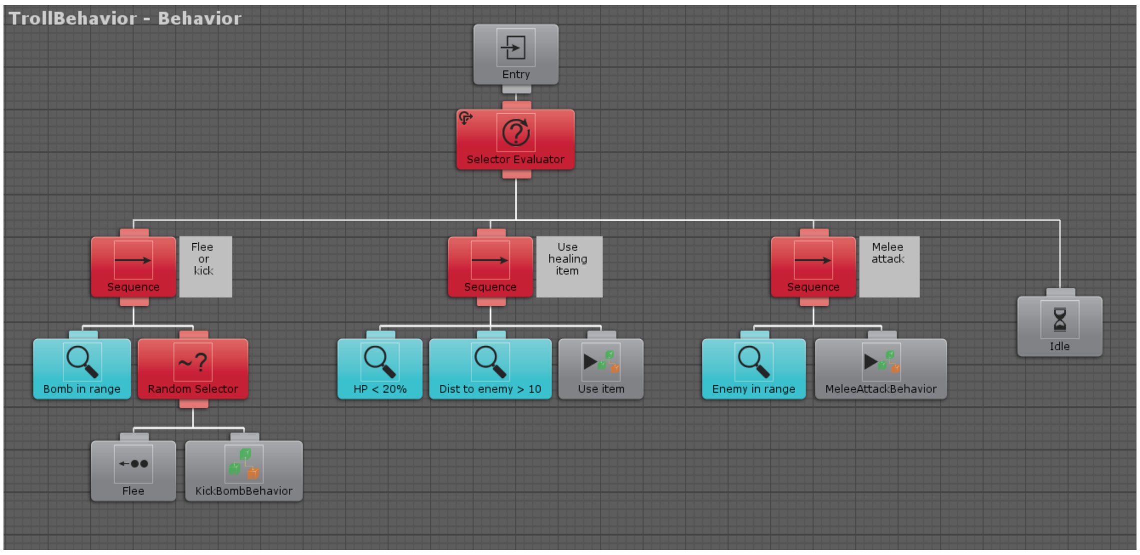


Figure 9. A behavior tree that utilizes external behavior trees.

While Behavior Designer proved to be very versatile, the behavior trees required to implement intricate AI behavior quickly grew so complex that the designers frequently needed assistance from the programmers to modify or debug the behaviors. To remedy this, the team established a workflow where programmers would create separate behavior trees for more complex behaviors, and the designers would use these trees as child nodes in a simpler, high-level decision-making tree (see Figure 9).

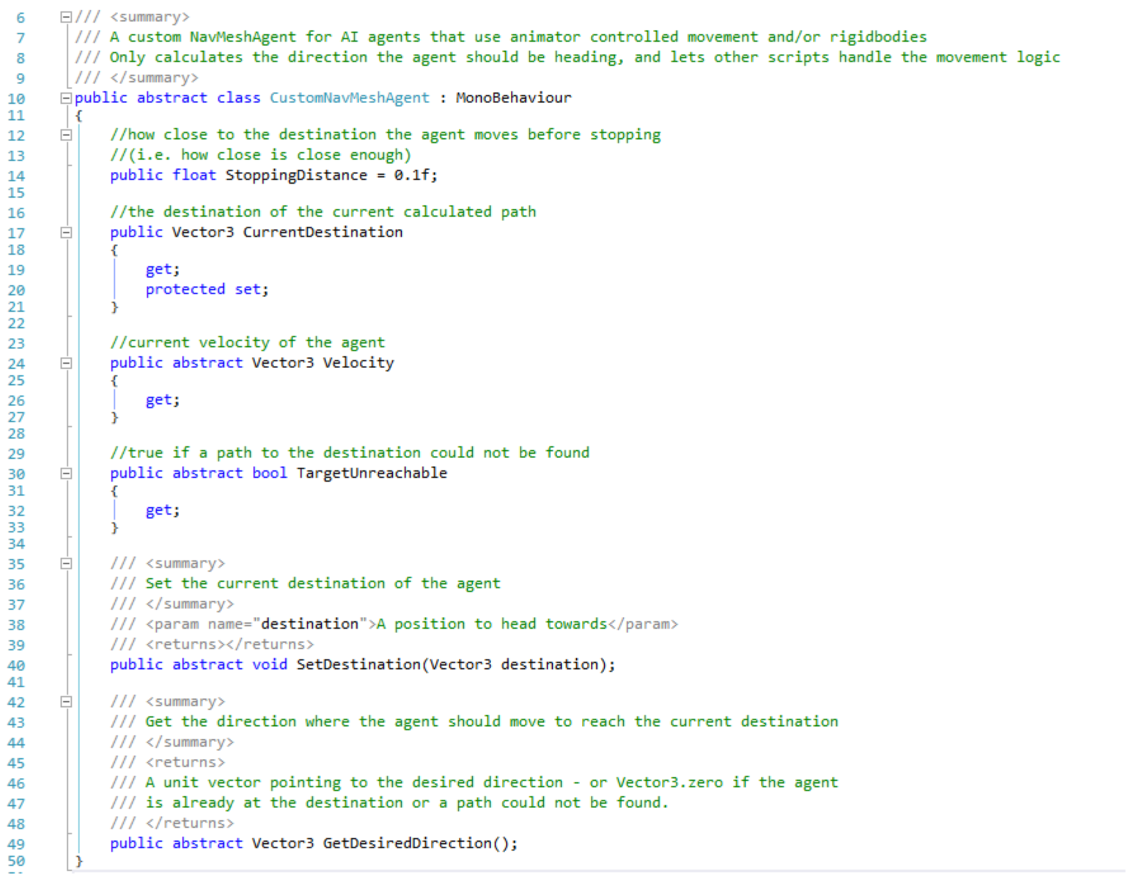
The functionality that allows creating pre-defined ”behavior templates” and modifying instances of these templates did not require any extra additions to Behavior Designer. Unity’s prefab system makes it possible to save the behaviors as a component of a prefab and the behaviors can be edited in the instances of the prefab without affecting the original behavior. However, this functionality has to be first enabled from Behavior Designer’s settings; instances of a behavior tree prefab cannot be edited by default, possibly to prevent accidentally editing an instance instead of the original one.

3.4.2 Pathfinding & locomotion

Despite choosing Unity’s navigation system as the pathfinding solution for the project, the built-in NavMeshAgent component was not used. This was due to the fact that the characters in Northbound use a combination of Mecanim’s root motion system and Unity’s physics engine for locomotion, both of which are somewhat problematic when using NavMeshAgent.

The basic idea of Northbound’s movement logic is that the characters are rigid bodies driven by the physics engine, and the root motion of the animations is used to modify the velocity of the bodies. This provides the advantages of the root motion system, while making it possible to do effects such as making an impact knock a character backwards without the need for a special ”knockback animation”. However, NavMeshAgent does not work correctly if attached to an object with a non-kinematic rigid body. Both the NavMeshAgent and the physics engine may try to move the agent at the same time, causing a race condition that leads to undefined behavior (Unity Technologies 2017d).

NavMeshAgent was replaced with a customized version that is functionally similar to the default, but with a few key differences. The custom version does not move the object by itself, instead it only calculates the path to the destination and provides a public method that can be used to determine which way the agent should be heading at a given moment. The actual movement logic has to be handled by other classes, in Northbound’s case the character classes.



```
/// <summary>
/// A custom NavMeshAgent for AI agents that use animator controlled movement and/or rigidbodies
/// Only calculates the direction the agent should be heading, and lets other scripts handle the movement logic
/// </summary>
public abstract class CustomNavMeshAgent : MonoBehaviour
{
    //how close to the destination the agent moves before stopping
    //(i.e. how close is close enough)
    public float StoppingDistance = 0.1f;

    //the destination of the current calculated path
    public Vector3 CurrentDestination
    {
        get;
        protected set;
    }

    //current velocity of the agent
    public abstract Vector3 Velocity
    {
        get;
    }

    //true if a path to the destination could not be found
    public abstract bool TargetUnreachable
    {
        get;
    }

    /// <summary>
    /// Set the current destination of the agent
    /// </summary>
    /// <param name="destination">A position to head towards</param>
    /// <returns></returns>
    public abstract void SetDestination(Vector3 destination);

    /// <summary>
    /// Get the direction where the agent should move to reach the current destination
    /// </summary>
    /// <returns>
    /// A unit vector pointing to the desired direction - or Vector3.zero if the agent
    /// is already at the destination or a path could not be found.
    /// </returns>
    public abstract Vector3 GetDesiredDirection();
}
```

Figure 10. The abstract base class of the customized NavMeshAgents.

The public interface of the custom agent is displayed in Figure 10. The actual logic was implemented in a subclass to make it easy to add different types of movement in the future; for example, a flying agent would use completely different navigation logic than a walking one.

A subclass called ”WalkingNavMeshAgent” was created and the methods implemented as follows: Calling SetDestination sets the position the agent is heading towards, and if the destination differs from the previous destination or if no path has been calculated yet, also sets a flag to indicate that the path needs to be recalculated. GetDesiredDirection uses the NavMesh.CalculatePath method to calculate the path if the flag has been set, and returns a normalized vector indicating the direction from the agent’s position towards the next node in the path.

While this implementation was already somewhat functional, some simple performance optimizations were made to reduce the number of path calculations. The agents do not flag the path for recalculation if the destination changes by an insignificant amount, since the already calculated path will take the agent very close to the destination at which point the agent can move directly from the last path node to the destination without any path calculations. This reduces the amount of path calculations drastically, as the agents do not constantly recalculate the path when following a moving target.

Another optimization was adding a time limit that prevents a new path calculation from being done until 0.1 seconds has passed since the last calculation done by the agent. The drawback is that this makes the agents less responsive, but in general the delay is not noticeable unless the target is moving extremely fast.

In addition to the performance optimizations, some extra measures were taken to handle edge cases that might cause the agents to get stuck. With Unity’s default NavMeshAgent and the first iterations of the custom NavMeshAgent it was possible for an agent to get stuck if they ended up outside the walkable area of the navigation mesh. Ideally, the game level and the navigation mesh should be created in a way that all areas where an agent might end up are covered by the navigation mesh, but in practice this is extremely difficult and time-consuming to achieve unless the level geometry is very simplistic. This issue was remedied by making the agent attempt to find the closest edge in the navigation mesh and use it as the starting point in the path calculations if the agent itself is not on the mesh.

Another issue with the first iterations of the custom NavMeshAgent was that agents would occasionally get stuck at the corners of obstacles when they are trying to navigate around them as illustrated in Figure 11.

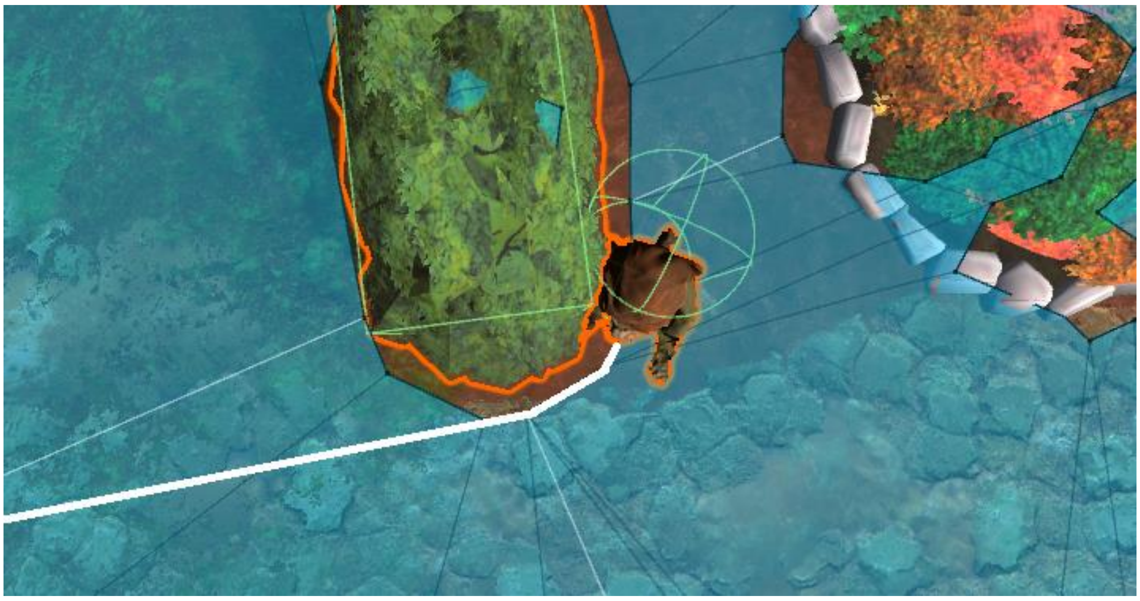


Figure 11. A gnome stuck against an obstacle when trying to navigate the path marked with the while line.

When baking a navigation mesh in Unity, it is possible to select the radius of the non-walkable area around obstacles. This setting can be used to prevent the aforementioned issue by using a radius equal to or larger than the collision radius of the agents. However, if agents of different size have to be able to use the mesh, increasing the radius will prevent smaller agents from being able to go near obstacles, and decreasing the radius will lead to larger agents getting stuck.

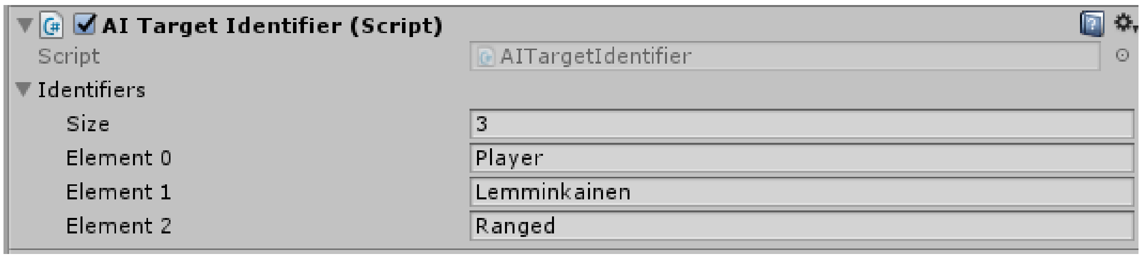
This was solved by making the agents steer away from obstacles. When the agent detects a collision with the level geometry, it saves the contact point and adds an ”avoidance vector” to the current desired direction until the contact point is far enough from the agent. The avoidance vector is simply a vector pointing from the contact point towards the agent, and its magnitude is interpolated from one to zero as the agent moves further from the point.

3.4.3 Perception

One of the requirements of the system was the ability for the AI agents to react to the player and the environment. For instance, an enemy might flee if they see a bomb appearing nearby, or climb into a tree if the player character is visible. Making the game world feel living and dynamic is a key objective in the development of Northbound, and as such, there needs to be a flexible system in place that makes it simple to make AI agents perceive other objects and react to them.

Behavior Designer offers built-in tasks that can be used to determine if the agent can see or hear another object. Both tasks can be configured to either check the visibility to a specific object or to an object with a specific tag. The problem with this approach is that it makes the system somewhat inflexible. Selecting a specific object to do the visibility checks on is usually not an option, as the object does not even necessarily exist before runtime (e.g. a bomb dropped by the player). The agents also need the ability to perceive multiple objects – it would be extremely cumbersome if making an agent attack the nearest of ten enemies required creating ten visibility check tasks. Tags, a Unity's built-in system that allows marking an object with an arbitrary string, would be a better alternative, but also has some limitations. Unity only allows assigning one tag to each object, which could cause some complications. If some AI agent should only attack one of the three playable characters while another should attack all of them, the latter would require a separate check for each player character tag. Another potential issue with tags is that Northbound already uses them in some of the gameplay logic. Doing changes to the tags for AI purposes could easily cause some of the gameplay logic to break, especially if the person doing the changes is not a programmer familiar with the inner workings of the game.

Due to the aforementioned limitations of the built-in tasks, a custom perception system was implemented . The system is similar to the tag-based logic: objects can be assigned any number of ”AI identifiers”, which are essentially arbitrary user-defined strings.



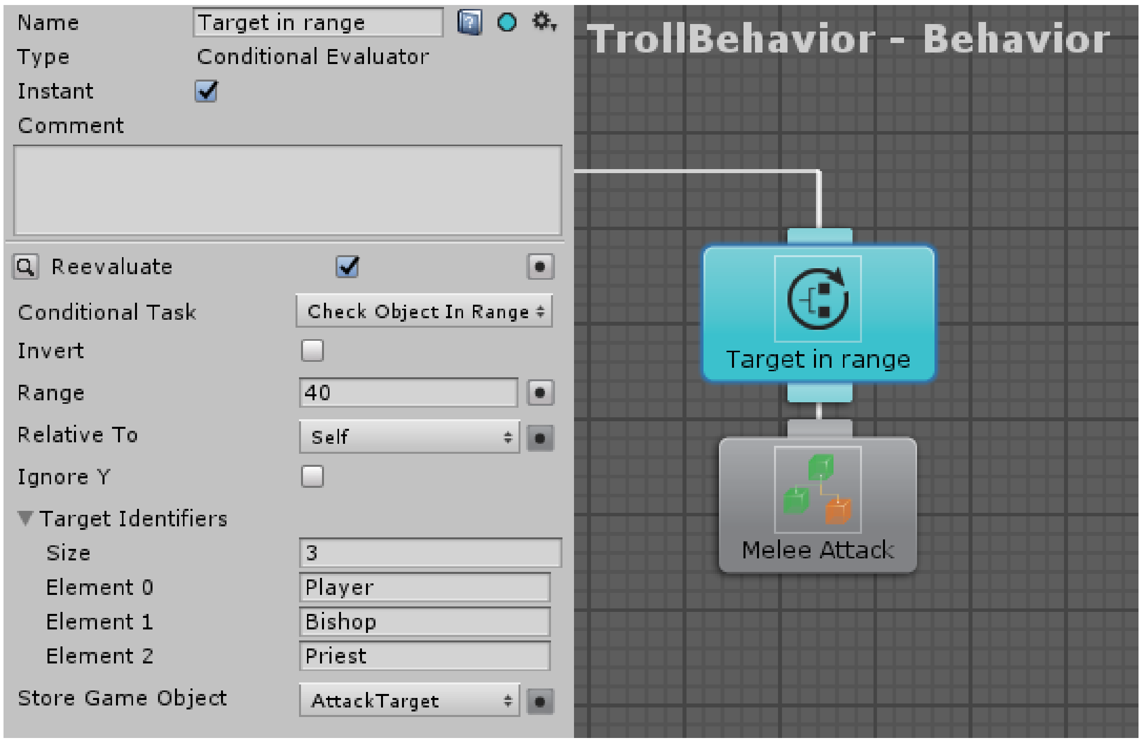
AI Target Identifier (Script)

Script	AITargetIdentifier
Identifiers	
Size	3
Element 0	Player
Element 1	Lemminkainen
Element 2	Ranged

Figure 12. Example of the identifiers of Lemminkäinen, one of the playable characters.

The AI identifiers are defined in an ”AITargetIdentifier” component that can be assigned to any gameobject. Instances of the AITargetIdentifier class add themselves to a static list of identifiers during initialization. The class includes static methods that can be used to query said list and fetch a list of gameobjects – either by finding all objects with a specific identifier, or only objects that are within a specific distance of a position in the game world.

The identifiers can be queried by using one of two custom tasks that were added to Behavior Designer. CheckObjectsInRange finds all the objects that are within a specific distance from the agent and have the correct identifier, and stores them in a variable. CheckObjectInRange works otherwise the same, but it only finds the closest object.



Name	Target in range
Type	Conditional Evaluator
Instant	✓
Comment	
Reevaluate	✓
Conditional Task	Check Object In Range
Invert	
Range	40
Relative To	Self
Ignore Y	
Target Identifiers	
Size	3
Element 0	Player
Element 1	Bishop
Element 2	Priest
Store Game Object	AttackTarget

Figure 13. Example usage of the CheckObjectInRange task.

For example, the task in Figure 13 searches for a target that is within 40 units of the agent and has either the identifier ”Player”, ”Bishop” or ”Priest”. The resulting behavior is that the agent will attack these three types of characters.

4 FINDINGS

The AI system was used to implement all the NPC behaviors in the game. This included behaviors for trolls that engage with the player in melee combat, gnomes that climb into trees and throw the player with apples, a peaceful giant that wanders around and curiously observes the player and nearby NPCs, monks that provide magical assistance to their allies, crows that flee when the player gets too near them, and a variety of other types of NPCs. The behaviors were mostly created by a game designer who had very little prior experience of AI development, while the programmers of the team mostly focused on creating custom AI tasks and occasionally helping the designer with more complicated AI behaviors.

The aforementioned game designer and one of the programmers of the team were interviewed to find out their thoughts about the system (see Appendix 1). Despite his limited programming and AI development experience, the designer found Behavior Designer easy to work with, and the programmer described it as intuitive and an improvement over the previously used RAIN AI engine.

Even thought the editor was seen as easy to use, the control flow logic of larger behavior trees often became very complex, which frequently caused unintended behavior and forced the programmers to spend additional time debugging the behavior trees. A workflow where the programmers would create separate behavior trees for more complex behaviors and the designers would use these as “building blocks” in a high-level decision-making tree remedied the issue to some extent, but did not solve it completely.

The identifier-based perception system was described as useful and easy to use by the game designer responsible for the implementation of the AI behaviors (Appendix 1). The identifier system was also integrated it into some of the non-AI systems of the game. For example, the identifiers could be used to restrict which kinds of objects can activate specific triggers or be affected by specific magical abilities or physical attacks.

The pathfinding and locomotion features were perhaps the most problematic part of the AI system. Both of the interviewed team members brought up their bugginess when inquired about the shortcomings of the system.

5 CONCLUSIONS

Despite the large number of AI extensions available for the Unity engine, it appears there is no magic bullet to overcome the complexity of AI development. Each tool has their strengths and weaknesses, and the applicability of a tool is heavily dependent on the types of behaviors required from the AI agents and the skill sets of the developers using the tool.

With the addition of a few custom features, Behavior Designer turned out to be applicable for for the purposes of Northbound, and an improvement over RAIN AI which was previously used by the team. As of writing this thesis, FakeFish is planning on using the new system to create the AI characters in the game and continue the development of the system.

Behavior Designer’s visual behavior tree editor turned out to be simple to use even for the developers with very limited programming experience, allowing them to easily modify and create AI behaviours.

The identifier-based perception system proved to be very versatile and an effective way to create various types of interaction between the AI agents and the environment.

The pathfinding and locomotion system is workable, but still much more limited and faulty than Unity’s built-in NavMeshAgent component. In its current state, the custom solution is not robust enough to handle many edge cases such as minor errors in the navigation mesh, moving obstacles or unusual agent movement properties (e.g. a combination of a high movement speed and a slow turning speed), which frequently causes the agents to move in an undesired manner.

5.1 Further development

5.1.1 Automated character creation tool

The AI system consists of multiple subsystems (behavior logic, animation, navigation, etc), and likewise the AI characters in the game also consist of several game objects, components and scripts. Creating new characters turned out to be somewhat cumbersome, as the user has to remember which components they need to add, which parameters they need to set, how the hierarchy of the game objects should be laid out and so on.

An automated tool or a “wizard” that initializes the AI characters based on a few user-configurable settings would help streamlining the creation of new characters and make the process less error-prone.

5.1.2 Simplifying behavior tree creation

Creating and modifying large and complex behavior trees with Behavior Designer turned out to be very error-prone, especially for the game designers with limited programming skills and no prior experience in behavior trees. A potential solution to this could be to take the idea of separating the complex, low-level behaviors from the high-level decision making logic even further. The basic idea is similar to the workflow described in the previous section: the designers are in charge of creating the root of the behavior tree and configuring the conditions that determine which branch of the tree should be executed, and programmers are in charge of implementing the more low-level behavior logic in the branches.

This type of separation between “high-level AI” and “low-level AI” could be enforced by creating an additional tool that allows creating the high-level decision logic without the need to make changes to the actual behavior tree. The user could simply choose appropriate the appropriate low-level behaviors from a list of pre-made behaviors and set the conditions that determine when each of them should be executed.

However, the user must still have some degree of control over the low-level behavior to prevent the system from becoming too restrictive. This could potentially be achieved with a similar approach as the Unity inspector, a feature of the Unity editor which displays information about the currently selected game object. The inspector only allows editing fields which have been explicitly made accessible from outside a script, which allows the programmers to restrict access to the “inner workings” of the scripts. Likewise, only some of the parameters of the low-level behaviors could be exposed to the high-level tool while access to the more critical and error-prone parts could be restricted.

REFERENCES

Salen, K. & Zimmerman, E. 2003. Rules of Play: Game Design Fundamentals. Cambridge, MA: MIT Press.

Russel, S. & Norvig, P. 2009. Artificial Intelligence: A Modern Approach. New Jersey: Prentice Hall.

Kehoe, D. Designing Artificial Intelligence for Games (Part 1). Consulted 4.4.2017 https://software.intel.com/en-us/articles/designing-artificial-intelligence-for-games-part-1

Unity Technologies. 2017a. Company Facts. Consulted 13.5.2017. https://unity3d.com/public-relations

Unity Technologies. 2017b. Unity Asset Store. Consulted 13.5.2017. https://www.assetstore.unity3d.com/en/

Unity Technologies. 2017c. Unity Store. Consulted 13.5.2017. https://store.unity.com/

Rabin, S. 2002. AI Game Programming Wisdom. 1st ed. Hingham: Charles River Media

Mark, D. 2012. AI Architectures: A Culinary Guide. Consulted 4.4.2017. http://intrinsicalgorithm.com/IAonAI/2012/11/ai-architectures-a-culinary-guide-gdmag-article/

Epic Games. 2017. Survival sample game tutorial. Consulted 4.4.2017. https://wiki.unrealengine.com/Survival_sample_game

Simpson, C. 2014. Behavior trees for AI: How do they work. Consulted 13.5.2017. http://www.gamasutra.com/blogs/ChrisSimpson/20140717/221339/Behavior_trees_for_AI_How_they_work.php

Rival Theory. 2017. RAIN AI Overview. Consulted 8.4.2017. http://legacy.rivaltheory.com/rain/

Hutong Games. 2017. PlayMaker Showcase. Consulted 9.4.2017. http://hutonggames.com/showcase.html

Opsive. 2017a. Behavior Designer Overview. Consulted 8.4.2017. http://www.opsive.com/assets/BehaviorDesigner/

Apex Game Tools. 2017. Apex Utility AI Scripting Guide. Consulted 13.5.2017. http://apexgametools.com/learn/apex-utility-ai-documentation/apex-utility-ai-scripting-guide/

Unity Technologies. 2017d. Using NavMesh Agent with Other Components. Consulted 8.4.2017. https://docs.unity3d.com/Manual/nav-MixingComponents.html

Opsive. 2017b. Behavior Designer Documentation - Tasks. Consulted 13.4.2017. https://www.opsive.com/assets/ BehaviorDesigner/documentation.php?id=3

User interview

Two members of the development team who had been using the AI system were interviewed on June 8th, 2017. Game Designer A had very limited experience in AI development and programming prior to using the system, while Programmer A had used RAIN and custom AI solutions in the past.

Which aspects of the system do you like?

Game Designer A: It’s easy to create behavior trees with Behavior Designer. The AI target identifier system is useful and it’s easy to find the identifiers.

Programmer A: Creating custom “building blocks” for Behavior Designer is easy, and the visual editor is intuitive and easy to use.

Which aspects of the system you do not like?

Game Designer A: The pathfinding system is too buggy. Having to break prefab connections in order to do extensive changes to instances of a prefab is cumbersome.

Programmer A: Navigation is too buggy.

How would you compare Behavior Designer to RAIN?

Game Designer A: Not enough experience in using RAIN to comment on this.

Programmer A: The core logic is essentially the same, but Behavior Designer’s behavior tree editor is better. Behavior Designer also seems less buggy.

What kind of changes or new features you would like to see in the system?

Game Designer A: There should be a way to make NPCs move in more specific ways, for example to “orbit” around some target.

An automated wizard that creates NPC prefabs would be useful.

The behavior tree editor should align blocks on the same line.

Programmer A: The blocks in the behavior tree editor should be automatically colored depending on the type of block, so we wouldn’t have to do it manually.