Tampere University of Applied Sciences

Behaviour Tree game AI in Unity

Eppu Syyrakki

BACHELOR'S THESIS
April 2023

Degree Programme in Business Information Systems
Games Production

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree Programme in Business Information Systems
Games Production

SYYRAKKI, EPPU:
Behavior Tree game AI in Unity

Bachelor's thesis 32 pages, appendices 1 page
April 2023

As game worlds grow in size and complexity, developers create new ways to manage that complexity, and artificial intelligence in games plays a key part of this management. Apart from its name, artificial intelligence in games has little to do with neural networks or large language models that the public generally perceives as practical examples of AI. In games AI concerns creating an illusion of believable responses by game entities to a dynamic environment.

The first part of this thesis compares the properties and usefulness of four popular examples of systems for managing game AI. The behaviour tree system is explored in depth to offer a theoretical basis on its functionality.

The second part focuses on the author’s implementation of creating that system as a library and an editor tool for Unity game engine, including descriptions of the critical parts of the system and explanations on how and why those parts function as they do.

While the purpose of this library is to simplify management, it is not a visual scripting tool, so understanding programming concepts along with some knowledge of the C# language and Unity is required to understand and implement the library for use in a game development project.

Key words: behaviour tree, unity, game, implementation

CONTENTS

1 INTRODUCTION 5
2 ARTIFICIAL INTELLIGENCE IN GAMES 6
2.1 State Machine 6
2.2 Utility AI 8
2.3 Goal-Oriented Action Planning 9
2.4 Behaviour Tree 10
3 BEHAVIOUR TREE 12
3.1 Pros and cons of the system 12
3.2 Elements within the graph 13
3.2.1 Repeater 14
3.2.2 Sequence 14
3.2.3 Selector 14
3.2.4 Parallel 15
3.3 Other elements 15
4 GOALS AND EXPECTATIONS 17
4.1 The example project 17
4.2 Library architecture 17
4.2.1 Graph - TreeAsset 19
4.2.2 Nodes - TreeNode 19
4.2.3 Agent – TreeAgent 20
4.2.4 TreeResponse 21
4.2.5 Context 21
4.3 Implementation details 22
4.3.1 Implementing the Leaf class 23
4.3.2 Creating a TreeAsset 24
4.3.3 Node Logic Flow 24
4.3.4 Using a Context 26
5 DISCUSSION 28
5.1 Results 28
5.2 Development possibilities 29
5.3 In closing 30
REFERENCES 31
APPENDICES 32

GLOSSARY or ABBREVIATIONS AND TERMS (choose one or other)

AI	Artificial Intelligence
graph	Means of representing data with a series of nodes and their connections to each other.
node	A point in a graph where pathways intersect.
state	The contents of any variables within a program at a given time during its execution.
sensory data	Output data from an abstracted sensor model such as vision or hearing.
pathfinding algorithm	Unspecified way of finding a series of connections that lead from a node to another node inside a graph.
game entity	Abstraction of a self-contained game character.
AI entity	As above but restricted to a computer-controlled game character.

1 INTRODUCTION

Creating human-like behaviour with non-human tools such as programming languages is a contradiction. Human brains are inherently nonbinary systems and have more in common with quantum computing than with binary machine code. Our behaviour is not driven by rigid rules – we weigh the consequences of our decisions, calculate risk against reward and have the ability to adapt to new situations. Computer systems operate under a different paradigm altogether - the rules, prerequisites and effects of any behaviour must be clearly defined before the program is run. Any perceived adaptation to environmental changes in a game should give the player the illusion of this realistic human-like behaviour, while the background for this behaviour is not human at all.

As modern games thrive for more realistic and believable AI behaviour, managing systems that enable such behaviour becomes an increasingly complex task. Managing this complexity is a central pillar in any game AI, and the different tools, patterns and systems game developers use are merely variations on how such complexity is managed. (Orkin, Jeff. 2006. Three States and a Plan).

This thesis attempts to list some of the most common solutions that are used to achieve and manage such behaviour and describes in deeper detail one of those solutions – a behaviour tree, also known as a decision tree. The first part describes the theoretical basis and structure of the system, and the second part illustrates the implementation of such a system as a general use Unity tool kit with an included example project.

2 ARTIFICIAL INTELLIGENCE IN GAMES

While AI as a general term refers to machine learning and handling large quantities of data, AI in the games industry is a very different matter. In games the purpose of AI is not to be as efficient as possible in executing a task, but rather to create an illusion of human-like intelligence and decision-making in game entities.

This illusion is made possible by three things according to Steve Rabin: The first is that players have a desire to see patterns of human-like intelligence in the games they play. Second, we as humans anthropomorphize the world around us, attributing human traits and behaviours to things that don't actually have any. Third and perhaps the most important is that players place expectations on the behaviour of game entities – this relates to a well-known medical phenomenon called the *placebo effect.* If a game entity behaves in the way players expect it to, the illusion of intelligence is reinforced. This can be achieved in a myriad of ways, such as faking an emotional response or a personality on the game entity. (Rabin, Steve. 2017. Game AI Pro 3. Chapter 1.2.).

The most basic form to implement these responses is directly scripted behaviour: As the player crosses an invisible boundary in the game world, entities close by wake up and execute simple behaviours such as move towards the player and shoot. A more complex response like running away if the player is too powerful requires more depth and design in how the game world is assessed - this is where complex behavioural systems come in.

2.1 State Machine

An apt definition of a state design pattern is:

> "Allow an object to alter its behaviour when its internal state changes. The object will appear to change its class." (Gamma, Erich. 1995. Design patterns: Elements of reusable object-oriented software)

A finite state machine is the simplest form of this. In principle the machine has a predefined collection of behaviours that get swapped in and out according to arbitrary rules. There are a limited number of connections between these behaviours, and these connections are not always bidirectional. For example, a game entity might not be able to swap into a "Jump" state from a "Swimming" or "Crouched" state, but could end up in either from a "Jump" state, depending on the game world or user input respectively.

A finite state machine can be described as having four principles to how it behaves:

1. It has a finite set of states it can execute.
2. It can only be in a single state at any given time.
3. It receives inputs or events that guide it (like sensory data for example)
4. States have transitions to other states that are guided by the aforementioned inputs. (Nystrom, Robert. 2014. Game Design Patterns, 91.)

A concrete example of a state machine in Unity is the Animator (fig. 1). It has all the elements mentioned above, and in addition to its original context as an animation controller it can be repurposed to act as an AI state machine.

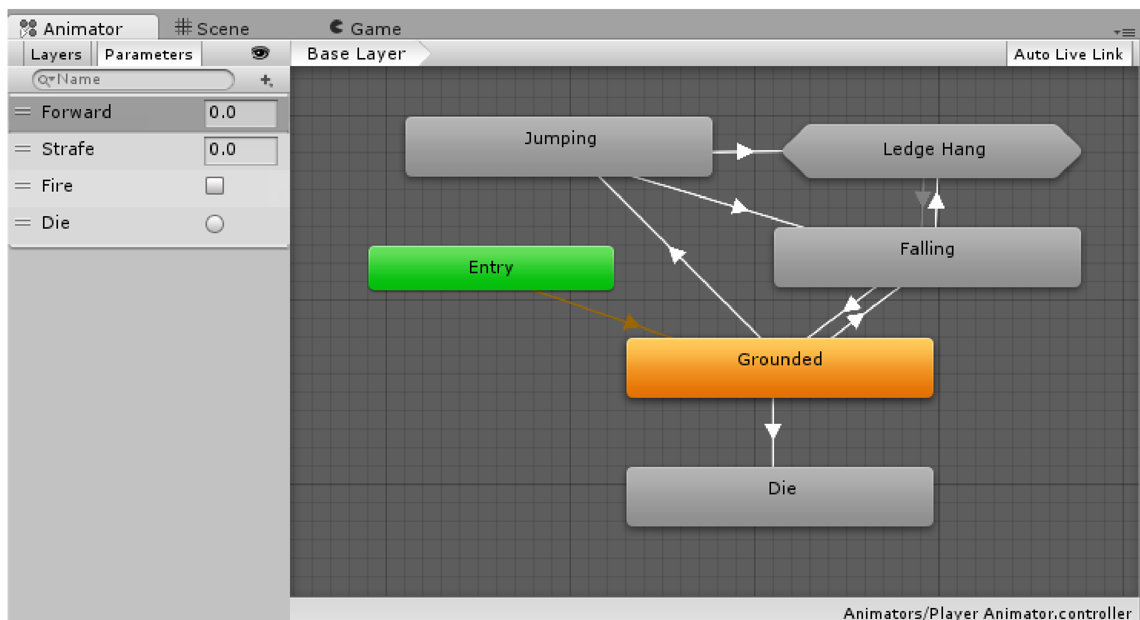

FIGURE 1. Unity's Animator state machine. The boxes are the states themselves, the left side is a representation of the input parameters controlling transitions, and the arrows are transitions between states (Unity User Manual. 2022. Animation).

This system is one of the most commonly used in games, and it is not restricted to AI behaviour – as Nystrom (2014) describes, common examples outside AI include player input, menu navigation, text parsing and other asynchronous systems.

2.2 Utility AI

A Utility AI system scores all possible decisions or actions that a game entity can take with mathematical formulas, and the entity chooses one from the best-scoring options (Lewis, Mike. 2017. Game AI Pro 3. Chapter 13). What the scored data is and how the formulas are handled is largely dependent on the game context itself. In general, the scoring data can be compared to the inputs in a state machine system – they can be world states and/or sensory data.

If carefully crafted, a utility AI empowers game designers in behaviour creation, as naturalised language concepts such as “prioritise” can be used instead of programming language concepts like “state” or “decorator”. Contrary to a state machine, adding new behaviour has no effect on already existing behaviour. (Rasmussen, Jakob. 2016. Are Behavior Trees a Thing of the Past? www.gamedeveloper.com).

A concrete example could be a war game that models soldier morale. A mortar shell exploding reduces the morale of any nearby soldiers. This will decrease the chance of an abstracted “attack” action on the next evaluation cycle because its scoring formula has an emphasis on high morale. The formula could also include the presence of a skilled officer close by, which might reduce some of this negative effect.

While this system makes AI behaviour appear more natural, it also complicates building it as a generalised tool due to the relative contexts of different game world implementations, although such developer tools do exist on Unity Asset Store for example.

2.3 Goal-Oriented Action Planning

Goal-oriented action planning (GOAP) is based on a game entity going through a process of searching for a sequence of actions to achieve some goal state, popularised by the game F.E.A.R. in 2005 (Orkin, 2006). Actions of the game entity change the state of the world, and in turn enable other actions to be executed that are dependent on the new state. It can be formalised as an A* pathfinding algorithm with the actions functioning as nodes and the altered world states as connections between them. Each action can be assigned a score to prioritise some actions over others, like shooting from cover instead of charging in for a melee attack to satisfy a “player dead” world state.

The main benefit of this system according to Orkin (2006) is the decoupling of goals and actions. Orkin draws an example from the game No One Lives Forever 2 (Monolith, 2002) where there were 2 types of police: out of shape and normal. While only the out of shape police characters stopped to catch their breath while in a “chase” state to satisfy a goal, the state machine for the normal police characters was required to have the same “chase” state, implementing a “catch breath” action they never used – adding unneeded complexity to the state machine. With the GOAP system, the “catch breath” action would simply not exist within the regular police character’s available actions.

A second important benefit from this decoupling comes from shared data. In a state machine system, each state only has access to information relating to itself – a “chase player” state doesn’t know about the entity’s regular patrol route. In a GOAP system, all data is shared, and this enables transitioning between any possible actions as long as all preconditions for that action are fulfilled by previous actions in the sequence (fig. 2).

The planning system also enables game entities to act dynamically with relatively small input from a designer. Orkin (2006) describes a scenario where a patrolling enemy walks through a door, notices the player and starts shooting. A variation of this scenario would be the player holding the door shut – the AI will try to open the door but fails. It re-evaluates the plan and finds an action to jump

through an adjacent window to satisfy the same goal of finding a line of sight to the player character.

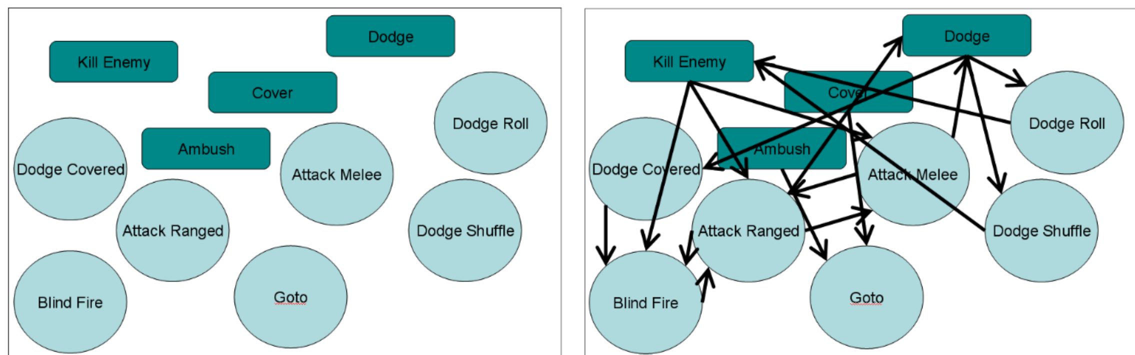

FIGURE 2. A GOAP system allows dynamic transitions between actions depending on their preconditions (left) instead of creating a rigid system of premade transitions (right) (Orkin. 2006).

2.4 Behaviour Tree

The behaviour tree (also called a decision tree) can be described as a graph: a tree of hierarchical nodes that control the flow of decision making of an AI entity. The extents of the tree, the leaves, are the actual commands that control the AI entity, and the branches are various types of utility nodes that control the flow of the AI query to reach the sequences of commands best suited to the situation (Simpson, Chris. 2014. Behaviour Trees for AI: How They Work. www.gamedeveloper.com).

In principle the AI entity sends a query from the root up the tree. When the query reaches an actionable leaf node, it returns one of three options: Success, failure, or running. If the leaf is running (its action is being executed), the entity will stay in that state. When the node returns a success or a failure, another query is sent to find the next “running” result. The branch nodes along the path can modify this result according to their own rules before sending it on.

For a leaf node to be reached, all the logic governing its preceding nodes must be satisfied as well. The utility nodes that guide the query towards the leaves can be described as logic gates – for themselves to return a success to a preceding node, they might require all of the following nodes to return a success

(AND), only one to return a success (OR), invert the result (NOT) and so on. This logic effectively forms a programming language in itself whose complexity can easily overwhelm the architecture. For this reason, it is important to keep the variations of logic nodes to a minimum in any implementation (Francis, Anthony. 2017. Game AI Pro 3. Chapter 9). The branches' generality makes their logic universal, while the implementation of the leaves depends on game mechanics and development decisions. Some implementations might include additional modules or nodes as preconditions to access their child nodes (fig. 3).

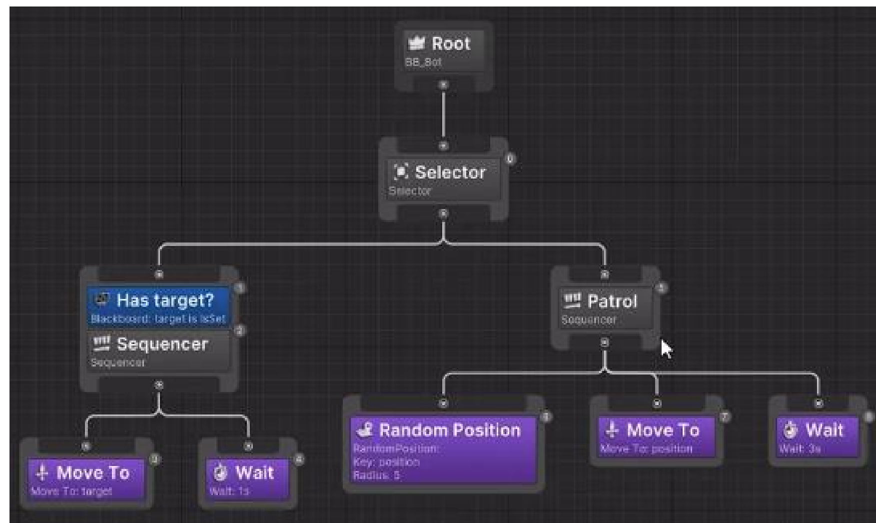

FIGURE 3. An example of a simple behaviour tree. The grey boxes represent the branch nodes (logic gates) and the purple boxes are the leaves (actions for the AI entity). The blue box is a precondition attached to a branch. (Renowned Games: AI Tree. Unity Asset Store. 2022.)

As games are created in an iterative process where the product is continually tested (both pre- and post-release), a behaviour tree can initially be designed as very simple, and later expanded to take into account new goals and even new game mechanics.

3 BEHAVIOUR TREE

3.1 Pros and cons of the system

A behaviour tree's usefulness lies in its simplicity and extendibility. The logic in the branches is straightforward and can be implemented relatively quickly (Dawe, Gargolinski, Dicken, Humphreys, Mark. 2013. Game AI Pro, 53). Because the tree is in practical terms a graph, it's also easy to represent with existing visual graph libraries. If the leaves can be atomized to simple, compact and independent units, they can be reused across a variety of different behaviour trees within a game project, and variations on a single tree can be easily created by duplicating an existing tree and swapping single leaves or adding new sequences or conditions.

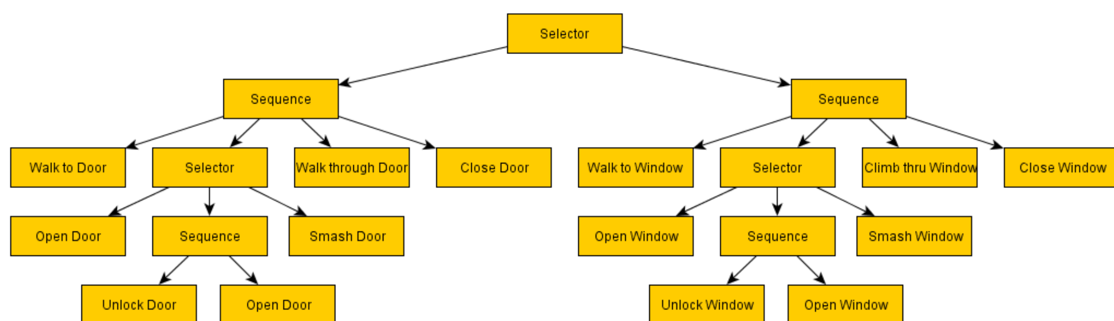

FIGURE 4. A Low-complexity behaviour tree represented as a graph. This particular graph relates to an AI game entity trying to enter a building, executing actions from left to right. First it will try to enter through the door by simply trying to open it. Failing this, it will try to unlock it, and if that fails, try to break the door open. If that fails, the entity will attempt to enter through a window with a similar sequence (Simpson. 2014).

There are also a few downsides to the system. If the tree grows large, the cost of a single query up the tree from the AI entity can be expensive in terms of processing power, especially if the tree is evaluated every frame (Simpson. 2014). Complex behaviour trees also run into the same problem as complex state machines: maintaining, understanding and debugging the tree becomes more difficult. An implementation of sub-trees is sometimes used to manage this complexity. A smaller tree can be inserted in place of a single leaf, somewhat simplifying management of the main tree (Rasmussen. 2016). The child tree could be

a shared entity between trees that use it, so modifications to one sub-tree would affect all other trees that use it. Sub-trees still merely hide the problem of complexity and address ease of use and visual representation, not the underlying problem of increased management.

These advantages and disadvantages make the behaviour tree system a good fit for low to medium complexity games. A modern open world game with a multi-million budget is not a good match for a behaviour tree due to its complexity with a multitude of possible behaviours. In medium or small-scale games however, it has clear merits stemming from relative simplicity and visual representation over other systems like utility AI.

The different parts of any implementation of a behaviour tree can be divided roughly into two parts – things inside the graph and things outside the graph. The inside parts represent the tree logic, while the outside parts are things such as a component that executes the actions from the tree and queries the tree when necessary.

3.2 Elements within the graph

The most important part of the graph is the leaf node that represents an action the AI entity will try to execute. These actions are also known as *behaviours* (Dawe et al. 2013) such as “walk” or “open door”. This node will return a “running” value until some condition is met - for example reaching a destination in the case of a “walk” behaviour – after which it will return either a “success” or “failure”. How the behaviour is represented is up to the implementation - it could be an “execute” method built into the node itself, a pointer to an existing object or game asset, or even a separate class constructed by the node on demand. The leaf node should not have any child nodes as it should always be the final node in any chain.

Some implementations separate conditional checks or preconditions from leaf nodes for clarity (Dawe et al. 2013). Conditional checks occupy the same place

in the graph as a leaf node, being the final link in a chain, but they lack any execution context and return only “success” or “failure”.

The second fundamental part is the branch node, also called *composite*. It can have a parent node as well as an arbitrary number of child nodes. It will process its child or children and pass the result on to its parent according to its specific function. The most important composites are *sequence*, *selector,* and *repeater* (Simpson, 2014). A fourth composite called *parallel* can handle early exits from an action node (Champandard & Dunstan. 2013. Game AI Pro, 81).

3.2.1 Repeater

Of these four, the repeater is the simplest one. When it receives something other than “running”, it simply restarts itself and passes on a “running” result, possibly resetting its children before starting again. A slightly more complex implementation could execute a variable number of repeats before passing the received success or failure to its parent, or repeat its children until it receives a “failure”.

3.2.2 Sequence

Sequence is the composite equivalent of an AND logic gate. It will start running its children in order and with each “success” it receives, it will move on to the next child and pass “running” to its parent, until it has no more children left to run, in which case it will pass on “success”. If the sequence receives a “failure” result, it will pass it on directly, practically executing a sequence of children as far as possible.

3.2.3 Selector

Selector is the OR to a sequence’s AND (Simpson. 2014). It will try to run its children in order and pass on “running” until it receives a “success” from a child, which it will then pass on. If it runs out of children without receiving a “success”

result, it will pass on a “failure”, so in effect it will *select* a single child it can run and pass its result to a parent.

3.2.4 Parallel

An additional problem to consider is how the tree handles interruptions. These are situations where the execution of an action should be continually dependent on a condition external to the current action, like interrupting a guard’s patrol cycle when a hostile entity is detected. An effective way of handling this is with a Parallel composite node (Champandard & Dunstan. 2013, 81). It can run multiple leaf or condition nodes at the same time, which in practice is usually a series of conditional checks first, and an actionable node last. Dunstan and Champandard suggest the Parallel composite to be configurable to require either a single condition to fail or all of them to fail before passing a value to a parent.

3.3 Other elements

Things outside the graph can be implemented in a myriad of ways, but most implementations have a few things in common, such as some sort of blackboard (a memory storage for things such as sensor data and world states), and an agent that handles execution of the leaves. Most of them also include some sort of serialisation or export mechanism, or a more general saving system for the trees.

While the blackboard is an important feature of the system to enable the tree logic to access variables in the game world, Francis (2017, 123) warns against tying the blackboard too tightly with the overall logic of the tree - this might make refactoring one or the other a difficult task. His suggested alternative is strong decoupling or avoiding the use of a blackboard altogether in favour of having the leaves communicate directly with each other.

These external elements are highly dependent on the execution environment (choice of game engine and/or programming language), and the following implementation chapters will describe these in more detail in the contexts of the implementation for the example project.

4 GOALS AND EXPECTATIONS

The author's behaviour tree library implementation is designed to be as generic as reasonably possible to achieve compatibility across a variety of use cases. A simple example project is defined that requires a number of features to be present in the final product. The chosen platform is Unity 2021 with whichever Long Term Support version is latest at the time of writing. The library and example project should be wrapped into a .unitypackage file for easy dissemination, and be backwards compatible with earlier Unity versions up to Unity 2020 if possible.

4.1 The example project

At the time of writing, the FIFA World Cup 2022 is underway (FIFA. 2022. www.fifa.com), and to celebrate the popular sport the project is a highly abstracted physics-driven football match played by AI entities. The example pits two teams of 2-5 agents against each other in trying to move a ball inside a goal area. The agents and game area are represented with primitive 3D models. For simplicity, there is no need to enforce rules such as offside, and the game area is walled to avoid losing the ball. The agents within a team utilise at least two different behaviour trees: defensive and offensive. The agents are able to judge for themselves if they are close to the ball and decide whether they should shoot or pass the ball to a teammate, or to move to a new position without the ball.

Optimising the project is not necessary, but recognizing possible bottlenecks in processing and informing users about them is important. The example must also be able to run for at least 30 minutes without crashing or halting.

4.2 Library architecture

The library leans heavily on an external Unity library called XNode (Brigsted, Thor. 2021. https://github.com/Siccity/xNode/wiki). It provides the base node

and graph classes and the editor code to represent them visually in a specialised Unity editor window. The graph and node inherit from XNode's corresponding classes (fig. 5), allowing them to share any functionality present in the base classes. However, as noted in XNode's documentation, its built-in functionality is limited to viewing and editing graphs (Brigsted. 2021), so the node logic and the system querying the tree is implemented with custom code. The whole library can be summarised with 10 base classes, the roles of which are discussed in the following sub chapters.

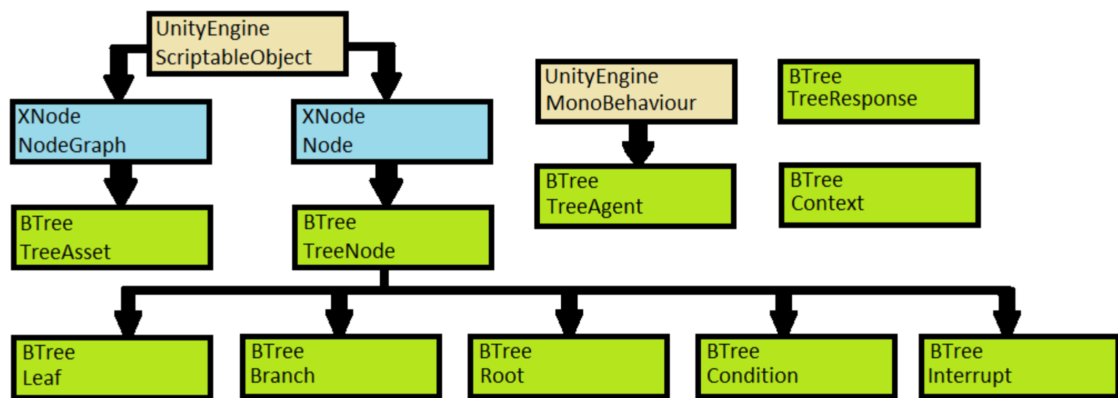

FIGURE 5. Inheritance of the library classes. Parts of the implementation are marked in green, XNode classes in blue, and Unity classes in light brown. An arrow signifies "inherited by."

The graph is the primary component of the library, named TreeAsset in the implementation. As its base class XNode.NodeGraph inherits from Unity's ScriptableObject, its creation can be done with custom menu items inside the Unity project window. This makes creating new trees as game assets and cloning existing ones as variations easy.

The nodes do not exist as instances on their own outside the graph. While their inheritance also traces back to ScriptableObject, they can't be created as independent assets. This behaviour can be altered by implementations by adding the CreateAssetMenu attribute from the UnityEngine library to the nodes. The implementation forces all nodes to inherit from an abstract base class TreeNode. The tree is forced to accept only nodes that inherit TreeNode to prevent user errors.

4.2.1 Graph - TreeAsset

The tree itself does not require much functionality apart from initiating a query at the root of the tree and delivering the result to the agent using it. Storing the graph along with the nodes and their connections as a Unity asset is handled by XNode. The Editor window provided by XNode handles the visual representation (fig. 6).

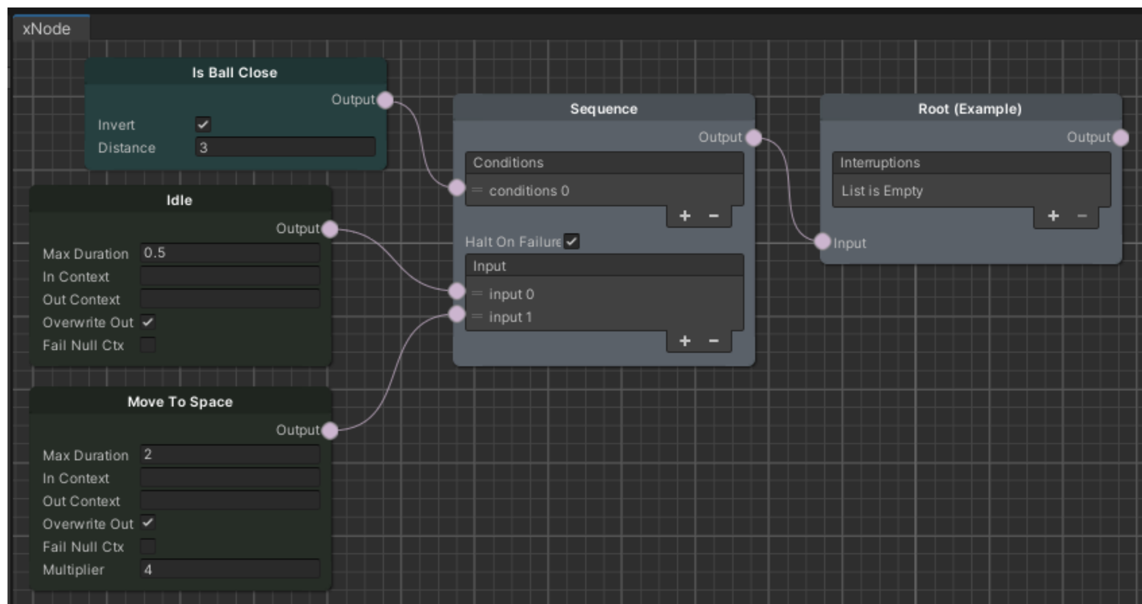

FIGURE 6. A simple TreeAsset represented as a graph by XNode's editor window. Grey nodes are logic or auxiliary nodes guiding the query. Dark green nodes are Leaf classes, and the turquoise node is a condition attached to a logic node. Functionally this tree will make the agent roam the play area if the ball is not close by.

4.2.2 Nodes - TreeNode

The **TreeNode** abstract base class holds functionality shared between all nodes - a reference to the agent who owns the tree, a single output port to guide the query and finding its possible children. The particulars of handling the delivery of the query result is left to each node type, depending on their function.

The abstract **Leaf** type is an actionable node - an object that can be hot-swapped to the agent. Any node inheriting from it holds practical code for an

agent to interact with the world, or some other way for the agent to act. Any branch of logic should always end in a Leaf.

The abstract **Branch** type is reserved for nodes that handle the tree logic, such as sequence or selector. Any Branch can have Conditions attached. In the implementation, the ability to attach conditions to any branch covers the responsibility of the parallel node type as described by Champandard and Dunstan (2013, 81–82). Users can implement their own Branch nodes, but the functionality present in the included Branch nodes can already cover a wide range of cases. Anthony Francis warns against making the architecture of the control flow branches too complex, as this can needlessly complicate refactoring the system (Francis. 2013. 117).

The abstract **Condition** signifies a boolean operation that is run on every frame. If it fails, the branch it is attached to fails as well. Conditions can also be used as Leaf nodes - in this case they are checked only when that Leaf is checked instead of being checked continually while another Leaf is being executed.

A **Root** node is the starting place for every query, and the only node the TreeAsset has a direct reference to. A tree both requires one of, and has no more than one Root in them.

Interrupt is a special type. It can't be reached through the tree's normal query, but can be forced to launch from outside the tree. In the example project, receiving a pass is handled with an interrupt: The agent passing the ball triggers an interrupt on the agent chosen to receive the pass. The ability to stop execution and jump to a different Leaf enables a significant part of event-driven behaviour trees' functionality (Champandard & Dunstan. 2013. 88).

4.2.3 Agent – TreeAgent

The **TreeAgent** class functions as an interface between the tree and Unity. It needs to handle selecting and storing the tree and a reference to the current actionable leaf, timing the queries to the tree, and changing the leaf to a new one

returned by the query when needed. It inherits from MonoBehaviour, so it can be added to a Unity GameObject as a component. Most functionality is extendable as virtual methods, making it possible to inherit from and customise the agent. The example project does this with the Player class.

As a ScriptableObject the tree is a shared entity (Unity Manual. 2022), it needs to be copied to a run-time instance when the game is played, or any changes made to the graph are permanent and will affect all agents using it. In the implementation, this copying is the responsibility of the agent.

Another important consideration for the agent class is timing. To avoid null reference errors in the execution where some object doesn't exist as it's being accessed, evaluating the tree and changing to a new Leaf must happen within the same call stack of methods - i.e. immediately. This ensures that the reference to the current Leaf is never null at runtime.

4.2.4 TreeResponse

The **TreeResponse** class is a wrapper object that is returned by the Root when the tree is evaluated. Its creation is the responsibility of the Leaf class, as the query ultimately reaches one through the logic branches. It holds a reference to the Leaf that created it, a Result enumeration (with possible values Running, Success and Failure), and references to any Condition nodes it encountered in the branches en route to the Root. The agent then uses this response to first check if any of the Conditions fail, and then to execute code in the origin Leaf.

4.2.5 Context

Context acts as a blackboard in the system. As a class it's very simple, holding key/value pairs of identifiers and objects. Agents have a personal instance of the Context object. It is filled by implementing Leaf nodes that in an unspecified way find an object in the world, store it in the Context property of a Leaf and use their Out Context field to specify an identifier for it. Any following Leafs can then access

a stored Context object by specifying the same In Context identifier.

The Context is a class and not a simple dictionary to enable instances of it to be created outside the TreeAgent class. It is up to the user how to access and modify these instances. Another benefit of implementing it as a class is that it can handle destroyed contexts better. If a context object is destroyed or removed from the game world, it must also be removed from the context dictionary.

4.3 Implementation details

The programming details of any Leaf class inheritor are naturally up to the implementation, but the user must understand the methods they are filling. This chapter describes these methods and their functionality within the system.

In practice the system is an amalgamation of a state machine and a behaviour tree. The Leaf classes are the states, only one of which can be active at any time, but their organisation and fetching is handled with a tree (graph) structure instead of transitions from state to state. As such, it can benefit from features designed for a state machine, such as “enter” and “exit” methods (Nystrom, 2014, 98-99).

The user is responsible for programming the states themselves by inheriting from the Leaf class and filling the required methods. The agent only drives an abstracted interface that the Leaf class implements, and calls three methods on it: **Enter**, **Execute** and **Exit**, which run code common to all Leaf nodes, such as fetching or resetting a context in Enter and Exit respectively. Those methods in turn call their counterparts in the inheritor with an “On” prefix. As such, any class inheriting from Leaf must implement the methods **OnEnter**, **OnExecute** and **OnExit**, among a few other utility methods that correspond to special situations such as initialization or forced failure.

As with any Unity object, Leaf classes can use editor-assignable variables by using the SerializeField attribute (Unity Manual. 2022. Attributes). XNode restricts these variables to be either prefabs or value types, as a Scriptable Object such as the TreeAsset or TreeNode classes cannot reference scene objects.

4.3.1 Implementing the Leaf class

OnEnter, OnExit and OnExecute can be called on the same frame in different nodes. The Result of a node is checked immediately after calling the Execute method on it. When that result is detected as not Running, OnExit is immediately called on that Leaf. The tree is then evaluated, and OnEnter is called on the resulting new Leaf. In the next frame, this new Leaf's OnExecute method is called and the Result checked again.

The OnEnter method is called when a new Leaf is received from the tree. It can handle setting up variables for the OnExecute method or other initialization specific to this particular execution of this node. At this point the possible context is already fetched from the agent by the base Leaf class.

OnExecute drives the performance of an action. It is called on every frame by the agent, and might contain a code block that handles setting the Result (fig. 7). For example, it could be set to Success on a "Move To" node if the agent is close to whichever target it was assigned (fig. 6). If a Max Duration is set on the node, the base class will advance a timer and check if that duration is exceeded and fail the node, but only if the result is still marked as Running. This enables inheritor classes to use the Max Duration field with their own timer to set a success result - the default implementation sets the Result as Failure if Max Duration is exceeded.

OnExit is called when the execution of a node ends. It can be used to reset any variables that were altered during the execution of this node, such as custom timers. The base class will handle resetting any possible context used.

Changes to the result of a node are done through a property of the TreeResponse class named Response, which in turn has a property of a Result enumerator with the possible values of Running, Success and Failure, as described previously.

```
protected override void OnExecute()
{
    if ((Agent.transform.position - target).sqrMagnitude < 1f)
    {
        Response.Result = Result.Success;
    }
}
```

FIGURE 7. A code example of setting the result of a node inside the OnExecute method. The "target" variable was assigned, and the movement initiated inside this node's OnEnter method.

4.3.2 Creating a TreeAsset

The TreeAsset is the template of the tree used by a TreeAgent, copied to an instanced version at runtime. Users can create new TreeAssets from Unity's project window via the Create menu. Cloning existing trees is handled like any other asset cloning inside Unity.

Once a TreeAsset is created, it can be edited in XNode's editor window by opening the asset. In the editor window, new nodes can be added via right clicking the window. Connections between nodes can be added by dragging from any Output port to another Input port. All paths of connections should eventually lead to the Root node.

4.3.3 Node Logic Flow

The user is responsible for ensuring a tree's logic flow is viable for execution. This requires understanding the functions of the Branch nodes. Theoretical basis for the Branches is described earlier in chapter 3.2., but is reiterated here in the context of this particular implementation.

Condition signifies a true/false condition that can be connected to either the Condition connectors on Branch nodes, or as a Leaf at the end of a logic path. In either case, an abstract method must be implemented to perform that check. The implementation has a single built-in condition called HasContext that compares a text field and returns a success or true result if an object with that name exists within the agent's context.

Interrupt can be called from outside the agent with the provided identification. Once called, any Interrupt will start executing the logic attached to it until no more Running results are found from its children, at which point the tree will return to normal execution - either continuing from where it left off or resetting the tree entirely, depending on the Force Reset field on the node.

Inverter is the simplest logic node, turning a Success result into a Failure and vice versa. It will not affect any Running result.

Repeater resets its children and restarts their execution on receiving a Success a set amount of times or indefinitely. It can be modified to act in the same way in the case of a Failure result by setting the Ignore Failures field on the node.

Selector chooses a single child node that is able to run and passes on the result received from it. If no children are able to run, or the chosen child returns a Failure, it will return a Failure upwards.

Sequence will try to run each of its children, moving to the next child if the previous returned a Success. A Failure result can be configured to halt execution and pass the result on, or simply alter that result to Running and move on to the next child by setting the Halt On Failure field on the node.

Selector and Sequence are both priority-enabled nodes. This means they will start examining their children in order from top to bottom, enabling a user to prioritise some actions over others if conditions remain the same.

Sub Tree is a convenience node that represents another TreeAsset. The node

will in practical terms act as another tree's Root node. This feature can be used to recycle behaviours and to reduce the visual complexity of any tree.

4.3.4 Using a Context

A Context container is used in the TreeAgent to store the internal context objects of that tree. The container is cleared every time the tree is reset.

Leaf is a generic class that can take the type of the context object it uses as the generic type. Any class intended for use as a context object must implement ITreeContext interface, but the ITreeContext interface can also be used as the generic type for the Leaf class if the user does not want to specify a type. The interface is very simple, requiring only a GameObject property named gameObject, so it will work without further setup on any MonoBehaviour script.

To add a context object to a tree, the user must implement a Leaf node that in an unspecified way finds the context object from the game world, and assigns it to the Context property of the Leaf class. When the node is exited, the base class adds that object to the agent's Context container with the identifier provided on the Out Context field in the node (fig. 8). An example of this functionality is provided in the example's *Find Target* node.

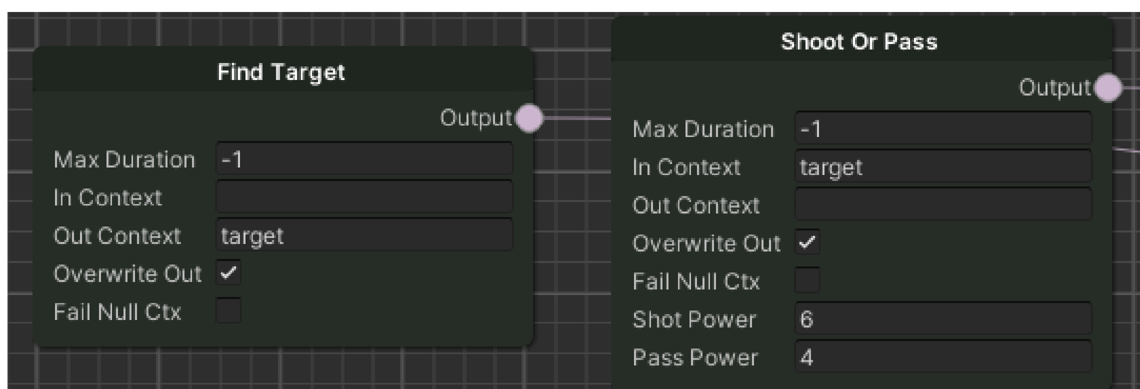

FIGURE 8. Two nodes in the example project utilising the Context fields. The left node sets a context with the identifier "target", and the right node uses that context.

Once a context object has been added, it can be used in other nodes later in the execution order by specifying the same identifier to the In Context field on the

node. The Leaf class will try to fetch the object from the Context container, and set it to the Context property of the class before calling OnEnter on the node.

While the user must take care in creating logically sound trees, the context system has a built in safety feature that Resets the tree when an agent tries to use a context object that is null. This feature works through a special Exception case and its handling within a Try/Catch block.

5 DISCUSSION

5.1 Results

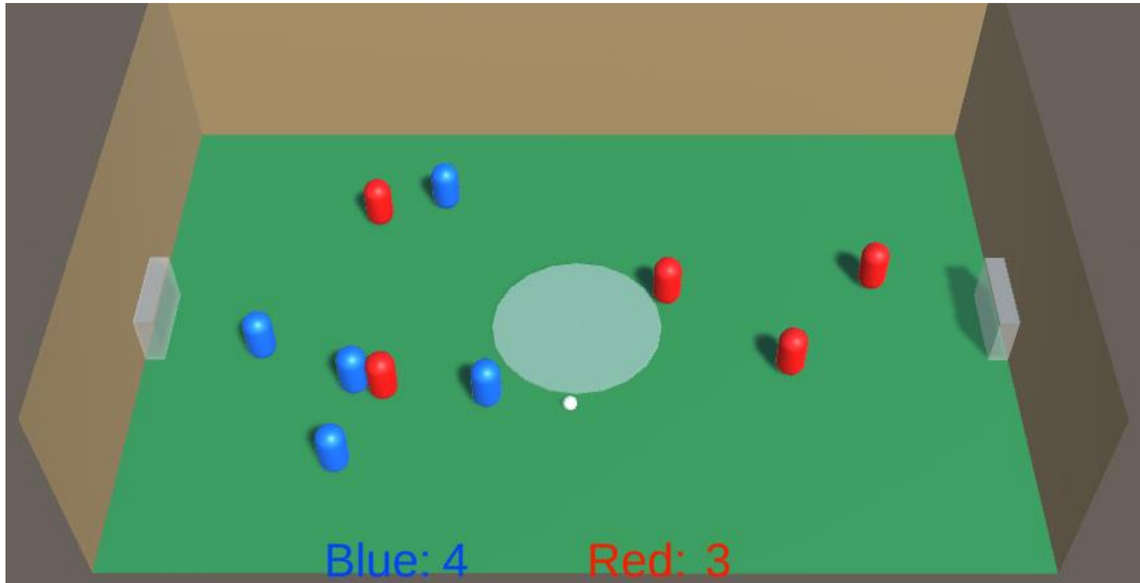

FIGURE 9. The example project being "played" by the agents in Unity.

While the author had prior experience in implementing a behaviour tree system, the task proved to be more complex and time-consuming than anticipated. Library design decisions and architecture had to be re-evaluated multiple times to account for use cases in- and outside the example project. A significant part of this redesign related to halting execution of an ongoing action. Without such functionality, the AI entities are not very reactive and lack dynamicity.

All Leaf classes in the example project are implemented naively, meaning they are unoptimized and written in a simple way to demonstrate the functionality of the system. The project still runs at a consistently high frame rate, despite having 10 AI entities running behaviour trees in real time. Doubling the amount to 20 entities did not affect frame times in the Unity editor in a significant way.

The greatest challenge turned out to be debugging and finding errors. As XNode doesn't offer runtime editor capabilities, a developer can't visually see the query path in the editor and must rely on console text to ascertain where any bugs or logical errors might propagate from

5.2 Development possibilities

The library works well for a real time simulation. There are some aspects of event-driven design incorporated in the library, but overall functionality does not support running the Leaf classes on demand – i.e. the system runs within Unity's game loop (Nystrom. 2014. Chapter 9). Implementing on demand behaviour would grow the pool of possible use cases significantly, such as turn-based game mechanics or a system where the Leaf classes are decoupled from character actions.

Another important feature would be improved debugging, as mentioned in the previous chapter. The current visual representation only enables easy editing of the trees, but a run-time visual tree showing which Leaf is currently running and the path the most recent query took, would make it significantly easier to build and debug more complex trees (fig. 10). Implementing this would require extending the XNode editor classes.

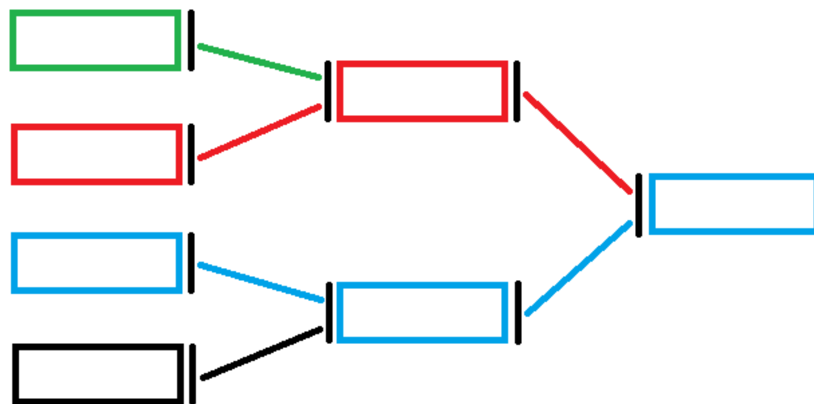

FIGURE 10. An early mock-up of improved debugging with visual feedback at runtime. A green colour would indicate a Success. Red nodes and connections have returned a Failure. Blue would mark the currently running query. All nodes and connections yet untouched would be marked in black.

The Context system shows promise but doesn't share data between agents. Having some kind of a "world context" that is shared between some or all agents could improve functionality significantly. Picking a use case from the ex-

ample project, this would enable switching offensive players to a more aggressive approach if the team is behind in score - the scoreboard being the shared context between all agents. In the current implementation this is possible by creating specific Leafs that fetch a context from the world, but the process could be streamlined with the shared context objects.

As it is, the project is not a .unitypackage file, but a simple code repository. Disseminating it as .unitypackage would make importing it to other projects significantly easier. This would require understanding how dependencies on external libraries are handled within .unitypackage files (such as with XNode in the implementation).

5.3 In closing

Behaviour trees are a good solution for medium-complexity games. As a game’s complexity grows, so grows the complexity of any AI solution it utilises. While a behaviour tree handles this complexity better than a state machine, it can become overwhelmingly difficult to understand what is happening inside a complex tree with multiple paths and sub trees – especially without visual feedback in the graph at runtime.

While designing the architecture of any library before implementation is important, it’s very difficult to account for all possibilities before creating a working prototype. Trying to conform too much to a predesigned template can make later changes to some crucial part of it a very time-consuming task, and the same is true for overcomplicating the design. This is evidenced by the author’s difficulties in adding the Condition functionality to all logic nodes late in development.

REFERENCES

Orkin, J. 2006. Three States and a Plan: The A.I. of F.E.A.R. Game Developers Conference 2006.

Rabin, S. (ed.) 2017. Game AI Pro 3. Boca Raton, Florida: CRC Press.

Gamma, E. 1995. Design patterns: Elements of reusable object-oriented software. Reading, Massachusetts: Addison-Wesley.

Nystrom, R. 2014. Game Programming Patterns. Genever Benning.

Unity User Manual. 2023. Referenced on 1.11.2022. https://docs.unity3d.com/Manual/index.html

Lewis, M. 2017. Chapter 13: Choosing Effective Utility-Based Considerations. In Rabin, S. (ed.) Game AI Pro 3. CRC Press.

Rasmussen, J. 2016. Are Behavior Trees a Thing of the Past? Referenced on 5.10.2022. https://www.gamedeveloper.com/programming/are-behavior-trees-a-thing-of-the-past-

No One Lives Forever 2. 2002. Developer: Monolith Productions. Publisher: Vivendi Universal Games, Sierra Entertainment.

Simpson, C. 2014. Behavior Trees for AI: How They Work. Referenced on 13.10.2022. https://www.gamedeveloper.com/programming/behavior-trees-for-ai-how-they-work

Francis, A. 2017. Chapter 9: Overcoming Pitfalls in Behavior Tree Design. In Rabin, S. (ed.) Game AI Pro 3. CRC Press.

AI Tree. 2022. Developer: Renowned Games. Publisher: Unity Asset Store. https://assetstore.unity.com/packages/tools/ai/ai-tree-229578

Dawe, M., Gargolinski, S., Dicken, L., Humphreys, T., Mark, D. 2013. Chapter 4: Behavior Selection Algorithms. In Rabin, S. (ed.) Game AI Pro. CRC Press.

Champandard, A., Dunstan, P. 2013. Chapter 6: The Behavior Tree Starter Kit. In Rabin, S. (ed.) Game AI Pro. CRC Press.

FIFA World Cup. 2022. Referenced on 30.11.2022. https://www.fifa.com/fifaplus/en/tournaments/mens/worldcup/qatar2022

Brigsted, T. 2021. XNode wiki. Referenced on 2.12. 2022. https://github.com/Siccity/xNode/wiki

APPENDICES

Appendix 1. The example project.

This is the self-contained example project created by the author as described in chapter 4. It is a link to a repository in the GitHub service.

https://github.com/EppuSyyrakki/BTree