Bachelorarbeit
im Studiengang
Medieninformatik (MI-7)

Implementation of an Abstract AI Framework for Basic NPC Behaviour

vorgelegt von Jan-Niklas Keck
Matrikel Nummer: 31046
an der Hochschule der Medien
am 31.07.2018
zur Erlangung des akademischen Grades eines Bachelor of Science

Erstprüfer: Prof. Dr. Stefan Radicke
Hochschule der Medien

Zweitprüfer: Kai Kurfess
91interactive

Abstract

This bachelor thesis describes the implementation of basic tools for the development of video game artificial intelligence in the Unity3D game engine. These tools consist of a behaviour tree implementation plus editor, a humanoid sensor system and an environmental query system. Each tool will be described and defined, with reasoning and implementation details as well as conclusions. After this, an example usage of these tools will be used to show their advantages and disadvantages. An overall conclusion at the end will describe their usefulness when developing artificial intelligence for video games, followed by a future outlook for possible improvements and additional features.

Kurzfassung

Diese Bachelor Thesis beschreibt die Implementierung von grundlegenden Werkzeugen für die Entwicklung von künstlicher Intelligenz in Videospielen anhand der Unity3D Game-Engine. Diese Werkzeuge bestehen aus einer Behaviour Tree Implementierung inklusive Editor, einem humanoiden Sensor System und einem Umgebungsabfrage System. Jedes Werkzeug wird mit Begründung und Implementierungsdetails beschrieben und definiert, gefolgt von einem anschließendem Fazit. Danach werden mit einer Beispielnutzung die Vor- und Nachteile aufgezeigt. Ein abschließendes Fazit beschreibt die Nützlichkeit der Werkzeuge bei der Entwicklung von künstlicher Intelligenz in Videospielen, gefolgt von einem Ausblick auf mögliche Verbesserungen und zusätzliche Features.

Ehrenwörtliche Erklärung

Hiermit versichere ich, Jan-Niklas Keck, ehrenwörtlich, dass ich die vorliegende Bachelorarbeit (bzw. Masterarbeit) mit dem Titel „Implementation of an Abstract AI Framework for Basic NPC Behaviour“ selbständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Die Arbeit ist noch nicht veröffentlicht oder in anderer Form als Prüfungsleistung vorgelegt worden.

Ich habe die Bedeutung der ehrenwörtlichen Versicherung und die prüfungsrechtlichen Folgen (§ 26 Abs. 2 Bachelor-SPO (6 Semester), §24 Abs. 2 Bachelor-SPO (7 Semester), § 23 Abs. 2 Master-SPO (3-Semester) bzw. § 19 Abs. 2 Master-SPO(4 Semester und berufsbegleitend) der HdM) einer unrichtigen oder unvollständigen ehrenwörtlichen Versicherung zur Kenntnis genommen.

Nagold, den 31. Juli 2018

Jan-Niklas Keck

Contents

List of Listings

List of Figures

Abbreviations

AI Artificial Intelligence

API Application-Programming-Interface

FOV Field-Of-View

FSM Finite State Machine

GUI Graphical-User-Interface

IMGUI Immediate Mode GUI

NavMesh Navigation Mesh

NPC Non-Player Character

SQS Spatial Query System

UAIFramework Unity AI Framework

UI User-Interface

Unity3D Unity3D Game Engine

Chapter 1

Introduction

Artificial Intelligence (AI) in video games has changed much since it was first created. From AI for boardgames like checkers or chess to more complex acting AI in games like the famous *Pac-Man* and finally today's impressive examples in games like Ubisoft's *The Division* or Irrational Games' *BioShock Infinite*. AI techniques are becoming more complex over time and they are harder to use without proper tools. These tools are usually provided by the game engine used for development but if they are not, developers have to either find other solutions or develop them them self, which costs time and resources. This thesis looks at the popular Unity3D Game Engine (Unity3D)[1] and its lack of built-in AI tools and tries to evaluate and implement three popular AI techniques for the development of Non-Player Character (NPC) behaviour in a common framework. The three technologies are:

- *Behaviour Tree*
- *Sensory System*
- *Spatial Query System*

In Chapter 2 the framework itself and the components will be described in more detail. The following chapters will each focus on the three mentioned techniques. They will describe them in detail, define their scope and implement it for analysis. After their description all three will be combined in a small example project, designed to show their strengths and weaknesses. In the end their will be a final analysis and future outlook regarding all three topics.

The Unity3D game engine is complex and provides many tools and features for developers in many different areas[2]. Concerning AI, there is only one dedicated tool: the Navigation Mesh (NavMesh) path finding implementation. AI in modern games is becoming more and more complex and thus harder to create and maintain. Proper tools play an important part to minimize the complexity and make AI development easier. To this end an abstract software framework with distinct components in Unity3D was developed. It is supposed to provide useful and easy to use tools for the development of AI in games, with designers, not programmers, as its main users. Three basic technologies are chosen and implemented: A *Behaviour Tree* implementation, including a custom editor Graphical-User-Interface (GUI), a *Sensory System*, to simulate generic senses, and a system for querying the environment for arbitrary information, named Spatial Query System (SQS).

Chapter 2

Unity AI Framework Introduction

The aim of this thesis is to evaluate the design and implementation of an abstract framework for AI technologies with the Unity3D. Each of the components is self contained and not dependent on the others but has greater potential if combined. This framework was simply named Unity AI Framework (UAIFramework), which is used in several places, mainly for folders and namespaces.

2.1 Artificial Intelligence in Video Games

The term AI throughout this thesis refers to the use in video games, rather than the actual meaning of *artificial intelligence*. AI agents in video games are seldom really intelligent creations, their purpose is to enhance the player's experience. Their function can often be achieved through much simpler algorithms and processes, since they do not need exact data but work fine with approximations. They also have access to the data of their whole world, instead of having to rely on sensors which can provide false readings of malfunction.

2.2 Framework Definition

"An object-oriented framework is essentially a set of abstract and concrete classes that collaborate in a precise manner to provide a common framework on which a range of applications can be built"[3]. Its operation is very much based on the Design Pattern of *Template Methods* [4] where a set of abstract classes provide a number of so called *Template Methods*, which are called by the frameworks inner systems. These methods are either *abstract* (must be overridden) or *hooks* (may be overridden) and thus allow extending of the framework by implementing custom derived classes (with its *Template Methods*). Example code for this kind of pattern can be seen in Listing 2.1 for the parent class and Listing 2.2 for the derived class.

```
1  // Abstract class, can not be instantiated and should be derived
2  public abstract class ParentClass {
3    // Abstract, must be overridden.
4    // Protected so its not called from outside classes
5    protected abstract void AbstractMethod();
6
7    // Virtual, may be overridden.
8      // Protected so its not called from outside classes.
9    protected virtual void HookMethod() { }
10
11   // Neither abstract nor virtual, can not be overridden.
12   // Public, will be called from outside classes.
13   public void ConcreteMethod() {
14     AbstractMethod();
15     HookMethod();
16   }
17 }
```

Listing 2.1: Template Methods Parent Class

```
1  public class DerivedClass : ParentClass {
2    protected override void AbstractMethod() {
3      // Do something
4    }
5
6    protected override void HookMethod() {
7      // Do something
8    }
9  }
```

Listing 2.2: Template Methods Derived Class

2.3 Unity3D Game Engine

The Unity3D game engine consists mainly of two different parts: The native backbone, written in C++ and similar, low-level languages, and the scripting environment using the C# language. Its main functionality is realized via a *Component Framework* where almost all functionality is realized via attaching components to otherwise empty *GameObjects*. The "World" in Unity3D is represented by *Scenes* which act as containers for hierarchies of *GameObjects*. The unit of distance is one meter per unit. These *GameObjects* generally have no functionality on their own, which is realized via components. By default, every *GameObject* has some sort of transform component, containing its relative position, rotation and scale. These components can be created by the user by writing a new C# class which inherits from the *MonoBehaviour* class[5]. This class provides many callback methods which are called from the underlying native code and thus provide interaction with the game world. These methods include methods like *Start*, which is called once on the frame when the component is enabled by the engine, *Update*, which is called every frame (every time the engine calculates a new image), *OnCollisionEnter*, which is only called when there is also a *Collider* or *RigidBody* component attached to this *GameObject* and another is colliding with it, and many more[6]. There are several methods with which an action can be executed regularly in the engine and as such, one execution will be referred to as an update tick, disregarding the actual implementation.

Editor The Unity3D editor User-Interface (UI) is obviously a very important part for the user. It provides many tools to aid in the development process. Important for this thesis are two main features:

- *ScriptableObject*: The *ScriptableObject* class provides a way for the user to write custom asset files. These asset files can contain any code from the user but the more important part is that these asset files can be referenced by *MonoBehaviour* components and thus make access very easy.
- *Immediate Mode GUI (IMGUI)*: Most of the GUI of the Unity3D editor is built with the IMGUI. Part of the IMGUI Application-Programming-Interface (API) is open to the user to extend and create editor tools. This can take many forms, from as simple as adding a context menu entry to create *ScriptableObject* assets, to modifying how a type is displayed in the editor or even to creating entirely new editor windows with custom functionality. Many of these features were used in creating the three AI components for this thesis. Most notably are a newly created editor window for behaviour trees and a highly customized inspector editor for the SQS.

2.4 Components

The following chapters describe the three different AI techniques implemented for the UAIFramework. To understand the three different AI techniques and their implementation, it's important to take a look at the problem it's supposed to solve, at the general description of the technique and the design and algorithms used to implement them. After this conclusions can be drawn as to their advantages and disadvantages and possible problems. The three technologies are:

1. *Behaviour Tree*: A data structure to design and define behaviours. A custom editor provides easy access to non-programmers.
2. *Sensory System*: Simulating important humanoid senses, in this case *Sight*, *Hearing* and *Smell*.
3. *Spatial Query System*: A tool designed to ask the environment smart questions.

Chapter 3

Behaviour Tree

Behaviour trees are a collection of "tasks", ordered in a tree-like fashion. They can be seen as an evolution of (hierarchical) Finite State Machine (FSM)'s with a few key differences. States and state transitions in FSM's are represented by "tasks" which contain the actual logic arranged in a tree-like structure (hence the name behaviour tree). These tasks are usually relatively small and for the most part self-contained, which allows them to be re-used in many different applications. They can also be grouped together to represent more complex actions in smaller sub-trees, which in turn can be used in higher level behaviours[7]. This creation of building-blocks allows the user to create complex behaviour by dividing it into smaller and smaller parts, which are combined together into larger ones. A FSM suffers from many dependencies between states and state-transitions. By (re-)moving one component, several other have to be checked for correctness. States tend to be specialized and can not be reused easily. While behaviour tree's are not the answer to every problem, as with FSM's, they are an useful tool to have when developing AI.

3.1 Design and Algorithm

The overall algorithm is very closely related to the one described by Alex J. Champandard and Philip Dunstan in their Paper "The Behavior Tree Starter Kit"[8]. One change is the name of the task base class from *Behaviour* was changed to *Node*. This change was done to better represent the task as a node inside a visual node editor. This editor was created alongside the behaviour tree implementation. Another was the addition of the return status *Aborted*, which will handle a sudden abort, even in the middle of a running task. A few other minor things were also changed, but do not change the overall operation.

The core to each *Node* is the *Update* function as seen in Listing 3.1. The *Update_Internal*

```
1  public Status Update() {
2    if status != Running then
3      initializeNode()
4    fi
5    status = Update_Internal()
6    if status != Running then
7      terminate(status)
8    fi
9    return status
10 }
```

Listing 3.1: Node Update Function

function calls any code overridden by child class implementations, including calling other node's Update functions and responding to their return status. This traverses the tree in a depth-first order, where each node in that path is entered and its *Update* called. Figure 3.1 shows an example behaviour tree. The return status is a very important part in this step.

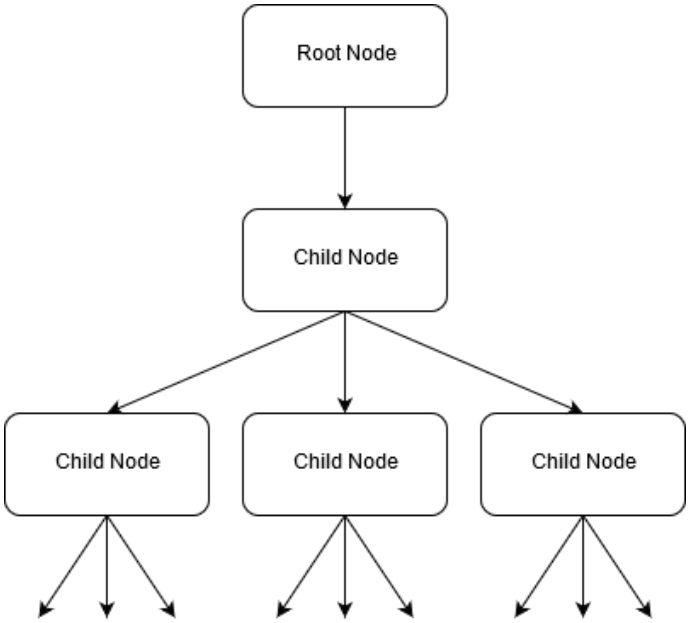

Figure 3.1: Behaviour Tree Structure

It provides information about the node's status. There are four different status a node can return:

- *Success*: Node was executed successfully.
- *Failure*: Node was not executed successfully.
- *Running*: Node has not finished execution.
- *Aborted*: Node execution was aborted.

The status is returned to its parent node, which then acts according to it. This means either passing the status further up, moving to its next child node or modifying the status. If a node returns the *Running* state, it means that this node operates over multiple updates ticks. That is necessary for any action that cannot be finished instantly (like walking towards a target or waiting for some time). If this happens, the status is passed along all the way back to the root node, which, on the next update cycle, takes the same path through the tree and updates the leaf node again, until it returns another status. To have some control over this process, nodes can be aborted to stop their execution. Once a node has been aborted, it will also abort all its children and return the *Aborted* state.

3.1.1 Node Types

Nodes are the core of any behaviour tree and come in a variation of types. There are three main types:

- *Composites*: Responsible for most of the flow of operation. They are the only nodes which can have multiple children, with the exception of the root node.

- *Decorators*: Can only have a single child and usually check some condition before executing their child but are also used to alter the return status of their child.

- *Leafs*: The main worker nodes. They cannot have any children and are located at the bottom end of their part of the tree. They contain the actual behaviour code.

An additional type is a *Service* node, which is a special case of a *Leaf* node. It is always executed within its parent *Update* call and provides a pseudo-parallel operation.

3.1.1.1 Composite

As written above, composites are responsible for directing the flow of the behaviour tree. There exist three main types:

- *Root Node*: The trees root node. Contains only one child and its only responsibility is to execute this child.

- *Selector*: The Selector Node can have multiple children and executes them in order. If any child returns the *Success* status, the Selector will stop iterating and also return the *Success* status. If no child was successful it returns the *Failure*. Selectors behave similar to an *if-elseif* structure, trying each branch until one is successful.

- *Sequence*: The Sequence behaves exactly like the Selector but instead of stopping when a child returns the *Success* status, it stops when the *Failure* status is returned. If every child is successful it returns the *Success* status. It is very useful for modelling actions as a sequence of leafs.

3.1.1.2 Decorator

A Decorator is a specialization of a composite node. It can only have a single child node and can have one of several different functions. It either changes the status that its child returns, stops the child's execution or does some form of repeating its child. Examples would be a "Cooldown" node which only allows entering after a specified delay or an "Inverter" node which inverts the status returned by its child (success to failure and vice versa).

3.1.1.3 Leaf

Leaf nodes by them self are very basic nodes. They cannot have any children and implement the actual functionality. This could be a "Walk" leaf node, which makes a character walk towards a given destination.

3.1.1.4 Service

Services are a special kind of leaf node. They are meant to run in "parallel" to its given parent node and execute itself in a given time interval. Parallel in this case means always executing them when the parent node is entered. Depending on their position in the tree, they can be global services or confined to a sub-tree. They can be used to periodically check or change something.

3.1.2 Blackboard

A blackboard in reference to behaviour trees is a data structure containing (arbitrary) information which is accessible to the whole behaviour tree and also can be shared between multiple behaviour tree instances. This is an easy way to share data between two or more nodes. For example, one node could gather information, save them in the blackboard and the next node could use this information to fulfil its task.

3.2 Implementation

Overall the implementation of the previously described algorithm with the Unity3D API was rather straightforward and produced no major problems. The creation of the visual editor and the integration of the behaviour tree was quite more complicated. Almost all classes and sub-systems went through a number of iterations to provide a pleasant user experience. The following sections describe how each part was implemented and how problems were solved.

3.2.1 Nodes

All nodes inherit from one abstract base class, *Node*. This class contains variables and logic common to all node types Listing 3.2 shows the important parts of this class. The parent-child relation between nodes is modelled via a *Parent* field and a *Children* list. While the whole tree could be constructed by just having the *Parent* field, the *Children* list exists for convenience and performance. Listing 3.3 shows the code for handling parent-child hierarchy changes. While every node has these methods available, the method *CanAddChild(child)* can be overridden to implement special behaviour. This is used to block leaf/service nodes from having any child nodes and to only allow a single child nodes for decorators and the root node. Following are some example nodes which have been implemented.

3.2.1.1 Composites

Composites are responsible for directing the flow of the behaviour tree. They are the only nodes which can have child nodes (with the exception of decorators). There exist three main types:

- *Root*: The root composite is, as the name implies, the root of the tree. It can only have one child node and its only purpose is to run its child and restart when it is returned to.

- *Selector*: The selector composite can have multiple children and its purpose is to iterate through them until one of them returns the success status. This effectively simulates a *if-else if* construct, which in this case is useful for creating a sort of priority list.

- *Sequence*: As the selector, the sequence can have multiple children and it iterates over them, with the difference that it will stop as soon as a child returns the failure status. This allows to execute a chain of nodes in a row, important for actions that have to be executed in sequence.

```
public abstract class Node : ScriptableObject {
  ...
  // Update node, handles initialization, abortion and update
  public NodeStatus UpdateNode(BehaviourTree parent, GameObject owner) {
  // If node was aborted set status to Failure\
    // and return Aborted status
    if (Status == NodeStatus.Aborted) {
      Status = NodeStatus.Failure;
      return NodeStatus.Aborted;
    }
    if (Status != NodeStatus.Running) {
      OnEnter(parent, owner); // Initialize node
    }
    // Run overridden update behaviour
    Status = Update_Internal();
    if (Status != NodeStatus.Running) {
      OnLeave(Status); // Potential clean up method
    }
    return Status;
  }
  // Called only once per node execution, initializes state
  protected virtual void OnEnter(BehaviourTree parent, GameObject owner) {
    this.parentTree = parent;
    this.owner = owner;
    Status = NodeStatus.Running;
    currentChildIndex = 0;
  }
  // Overridden by all nodes,
  // Implements specific node behaviour
  protected virtual NodeStatus Update_Internal() {
    return NodeStatus.Failure;
  }
  // Can be overridden by children
  protected virtual void OnLeave(NodeStatus status) { }
  // Abort this node and every child node
  // Abort happens on next update
  public void AbortNode() {
    if (Status == NodeStatus.Running) {
      Status = NodeStatus.Aborted;
    }
    OnAbort();
    int childCount = Children.Count;
    for (int index = 0; index < childCount; index++) {
      Children[index].AbortNode();
    }
  }
  // Can be overridden by children
  protected virtual void OnAbort() { }
  ...
}
```

Listing 3.2: Node Base Class

```
public class Node : ScriptableObject {
  ...
  // Check if a child can be added.
  // Checks for services, the root node and node connection cycles
  protected virtual bool CanAddChild(Node newChild) {
    // Services are only allowed on composite nodes
    if (newChild is Service) {
      return GetType() != typeof(Composite);
    } else if (newChild is RootComposite || newChild == Parent) {
      // The root node must not be a child node
      return false;
    }
    // Check for cycles
    return !IsChildAlreadyInHierarchy(newChild);
  }
  // Recursively iterates over every child to check for cycles
  private bool IsNodeAlreadyInHierarchy(Node node) {
    ...
    // Returns true if node already is anywhere in this
    // node's child hierarchy.
  }
  // Correctly adds child to this node.
  // Also removes child from previous parent if present.
  public bool AddChild(Node newChild) {
    if (CanAddChild(newChild)) {
      if (newChild.Parent) {
        newChild.Parent.RemoveChild(newChild);
      }
      newChild.Parent = this;
      Children.Add(newChild);
      return true;
    }
    return false;
  }
  // Removes child from this node
  public Node RemoveChild(Node child) {
    // If child could be removed, also reset parent field
    if (Children.Remove(child)) {
      child.Parent = null;
    }
    return child;
  }
  // Removes child from this node, based on index
  public Node RemoveChildAt(int index) {
    if (Children.Count > index && index >= 0) {
      Node removedChild = Children[index];
      removedChild.Parent = null;
      Children.RemoveAt(index);
      return removedChild;
    }
    return null;
  }
  ...
}
```

Listing 3.3: Node Base Code Regarding Children

3.2.1.2 Decorators

Decorators are used to model conditions in the behaviour tree. Each can only have one child and it will only enter this child if a certain condition exists. Example decorators would be:

- *Cooldown*: The child of the Cooldown decorator can only be entered every X seconds, which can be useful for a multitude of reasons.
- *Inverter*: The Inverter inverts the result status from its child, i.e. success to failure and failure to success.
- *BlackboardValue*: It checks if a given value on the blackboard is either set or not. This can be used to only set values once or to check if one is available.

3.2.1.3 Leafs

Leaf nodes contain the main functionality of the behaviour tree. They cannot have children are responsible for executing the actual logic. Example leaf nodes are:

- *Constant Status*: Simple leaf which returns always a pre-set status.
- *Check For Visible Player*: Hooks into the owners SightSensor OnNotify method to check if a player is visible. If a player is visible it saves the position in the blackboard and returns Success, otherwise Failure.
- *Walk To Blackboard Position*: This node will try to walk its owner to a position saved in the blackboard by another node.

3.2.1.4 Services

Services are special nodes. They cannot have children and are themself a special child to their parent. They will always be executed when their parent node is executed (independent from selector or sequence order)and thus are similar to a parallel operation. They can be configured with a cooldown, including a random deviation which will be added to the cooldown. Only a single service was implemented, the *RunSpatialQueryService*. It regularly runs the given Spatial Query and saves the best position in the blackboard.

3.2.2 Editor

Creating behaviour trees via code can be a complex job if the desired tree is also complex and basic to advanced programming knowledge is necessary. To enable non-programmers to create behaviour trees, they need an tool that is easy to use. The behaviour tree's structure already lends itself well to have a clear display of its parts. As seen in Figure 3.1, a simple display of boxes representing the different nodes and lines representing the parent-child relationships gives a clear understanding of flow and function. To make working with the tool as seamless as possible the Unity3D Editor was extended with a custom behaviour tree editor window and a few auxiliary classes. The Editor exposes many functions to allow users to build extensions to the editor[9].

```
1  public abstract class NodeAttribute : Attribute {
2    // Internally used string id.
3    public string NodeID { get; private set; }
4    // Name displayed in the editor.
5    public string ContextMenuText { get; private set; }
6    // If set to true, this will hide the node from the editor.
7    // Used for the root composite node.
8    public bool HideNode { get; private set; }
9
10   public NodeAttribute(string nodeID, string contextText, bool hideNode) {
11     NodeID = nodeID;
12     ContextMenuText = contextText;
13     HideNode = hideNode;
14   }
15 }
```

Listing 3.4: NodeAttribute Class

3.2.2.1 Class Attributes

To effectively use the behaviour tree editor, it needs a complete list of all available nodes to build a tree. Class attributes, built from the *System.Attribute* class, and Reflection was used to accomplish this. The base attribute class can be seen in Listing 3.4. It is used for identifying the different node types and providing some meta information to display in the editor. Several child classes of it are used to identify the different node types:

- *CompositeAttribute*
- *DecoratorAttribute*
- *LeafAttribute*
- *ServiceAttribute*

3.2.2.2 Serialization

Since the behaviour trees are not created in code, they have to be saved otherwise. For this reason all important parts of the behaviour tree inherit from the built-in class *ScriptableObject*. *ScriptableObject* instances are supported by the Unity3D serialization system and can be saved as **.asset* files. An important advantage of this method in comparison to other methods (i.e. saving as text/binary files) is that these **.asset* files can be referenced by *MonoBehaviour* instances. This makes working with them easier. They can be switched and reused without touching code or typing in file names. A potential problem however is the handling of which asset is currently edited. *ScriptableObject* asset file are serializing every change made in the editor automatically, without the need for a manual save action.

3.2.2.3 Editor Window

To make creating and editing of behaviour trees as easy and intuitive as possible, a GUI was implemented. All nodes, node connections and node properties are displayed inside a custom Unity3D editor window (see Figure 3.2 for example). Its design is similar to the default Unity3D editor, with a toolbar on top and the editing canvas below.

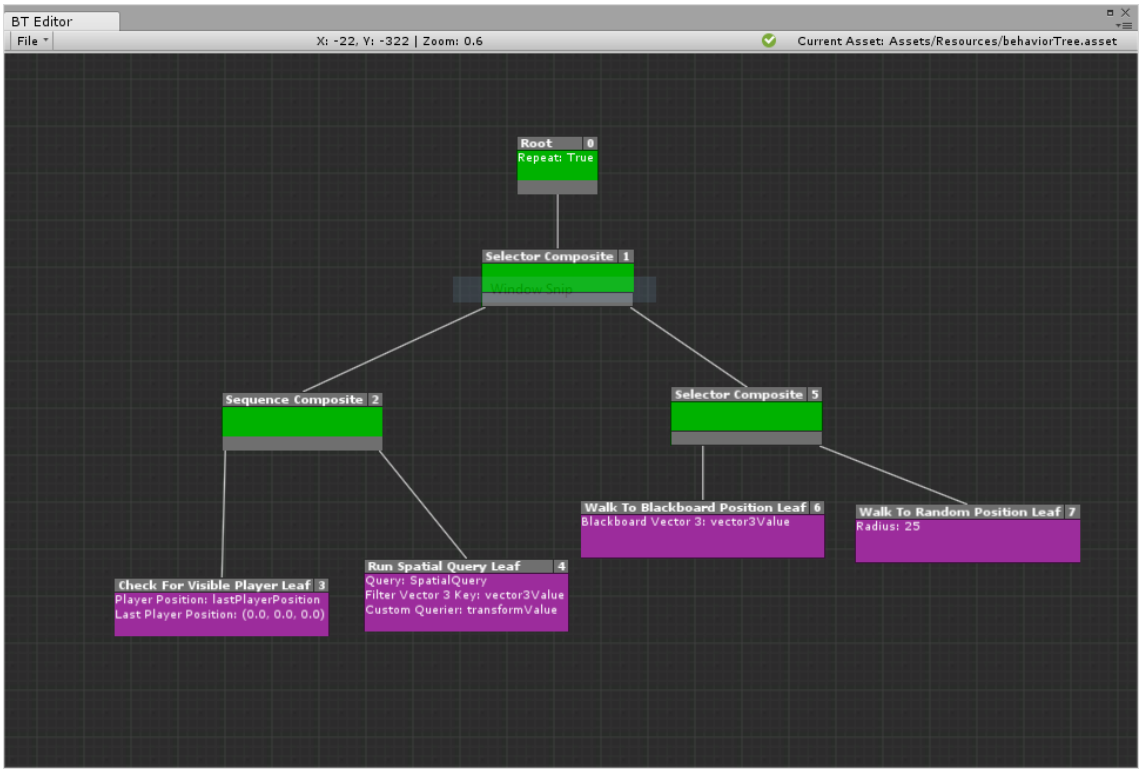

Figure 3.2: Behaviour Tree Editor Window With Example Tree

Toolbar The toolbar contains a single dropdown button which contains further buttons for saving, loading and resetting the current behaviour tree. It also displays the current position that is shown on the canvas, as well as the zoom factor, for orientation. And lastly it shows the file path of the currently loaded behaviour tree and a small icon which indicates if the behaviour tree is editable.

Canvas The canvas is a simple box area displaying a repeated grid pattern. The user can move around via dragging the mouse with the right mouse button held down and zoom in and out with the mouse wheel. Its coordinate system is based on floats, which allows for more than enough space. Its main purpose is to be a background with a lot of contrast, aiding in the creation of the behaviour trees.

Context Menu To add any kind of node to the tree inside the editor, the user has to use the context menu, available via a right mouse click. A small additional window allows for selecting and searching all available nodes (see Figure 3.3).

Nodes All nodes are displayed as rectangle boxes with a title and body. Composites and decorators also have an additional child area at the bottom. Figure 3.4 shows the different node types as they are usually displayed in the editor. The title displays the name of the node (which can be customized) and the current order of execution. This order is based on the horizontal position of the nodes in the tree and configured to increase from left to right. The body of the nodes displays all configurable variables of the node. These can be edited via selecting the node with a simple mouse click and then accessing the settings via a Unity3D Inspector window. If the node is a composite or decorator type node, there is

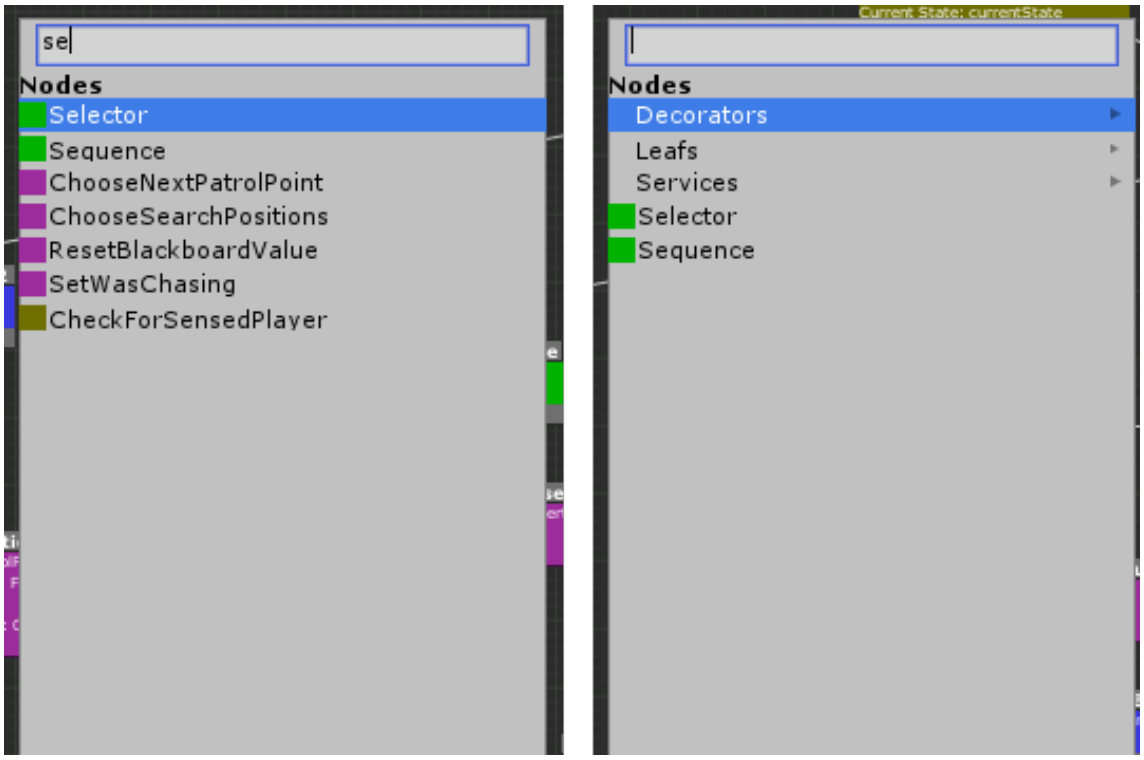

Figure 3.3: Behaviour Tree Editor Context Menu: Left in search mode, right without search mode

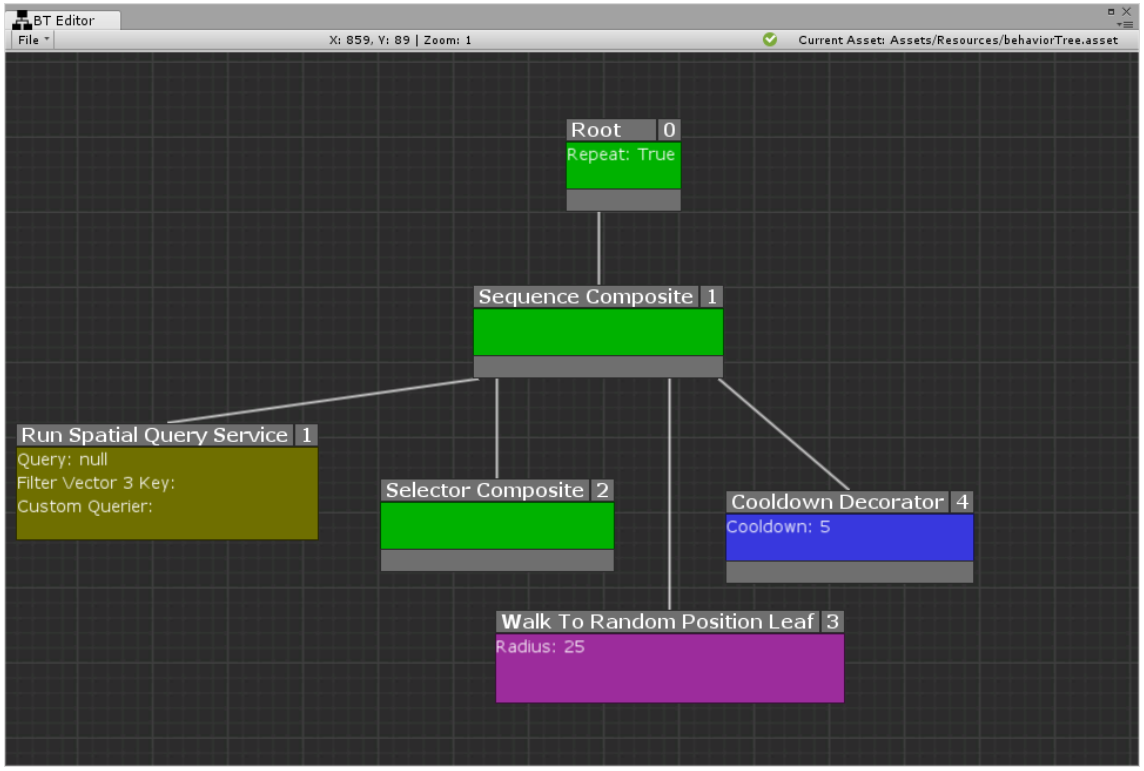

Figure 3.4: Behaviour Tree Node Examples

a third area at the bottom, the child area. This area is responsible to allow connecting nodes via a simple drag and drop operation from the child area to the desired child node. All nodes can be moved around on the canvas via a drag and drop operation where the mouse is hovering the node, but not the child area.

Node Connection Node connections are displayed via simple white lines, originating at the bottom of the parent nodes to the top of the child nodes. To edit these connections, lines can be deleted via hovering over them with the mouse (hovered lines have a red color) and pressing the delete key.

3.2.3 Blackboard

The blackboard implementation consists of two main parts: The blackboard base class (*BlackboardBase*) and the blackboard property class (*BlackboardProperty*).

BlackboardBase The *BlackboardBase* class is the parent class of all blackboards. It itself inherits from the *ScriptableObject* class, which makes it easier to share blackboard instance between multiple behaviour trees since they are saved as asset files. Its main component is an instance of a Dictionary (C# equivalent to a map data structure) with the keys being strings and the values being a custom *BlackboardValue* class, which is a wrapper class around the actual value. The wrapper adds a few important functions: It keeps track if the value was set to a value or not, for reference and value types. It also provides a *onValueChange* delegate, which is called every time the value changes. This also makes it necessary to initialize the blackboard, which is done lazily and uses reflection to retrieve all fields. Since the blackboard fields are fixed and known at compile time, their names are used as string identifiers for their values. This makes misspelling errors practically impossible and also allows for a easier debugging process. Listing 3.5 shows the initialization process. The values itself can be accessed with a number of getter and setters which simplify the access. Listing 3.6 shows the signatures of these methods.

BlackboardProperty The *BlackboardProperty* class's main purpose is to hold an identifier for blackboard field, which is selected while creating the behaviour tree. Every node that wants to access a blackboard field needs one or more *BlackboardProperty* fields. These fields then are filled in the editor window, where a drop down list shows all available fields (see Figure 3.5). They are not linked to a specific type of value, they simply hold the string name of the blackboard field to ensure a safe access to the values on the blackboard.

3.2.4 MonoBehaviour Component

To use a behaviour tree in the game world, a simple *MonoBehaviour* component was created. All it does is holding a reference to a behaviour tree asset file. When the *GameObject*, the component is attached to, is activated, the behaviour tree will be copied as a transient copy. If this is not done, all agents using this behaviour tree would be using the exact same instance, which obviously would cause problems. While the game is running, the copied behaviour tree's *Update* method will be called in the components *Update* method. This will traverse the tree and execute its nodes.

```
public abstract class BlackboardBase : ScriptableObject {
  private Dictionary<string, BlackboardValue> fieldValues;
  ...
  // Called every time the blackboard is queried
  private void InitBlackboardFields() {
    // Stop initialization if already done
    if (fieldValues.Count > 0)
      return;
    // Get public, non-static field meta data
    FieldInfo[] fieldInfos =
      GetType().GetFields(BindingFlags.Public | BindingFlags.Instance);
    foreach (FieldInfo info in fieldInfos) {
      // Add new entries to the blackboard according to field meta data.
      BlackboardValue value =
        new BlackboardValue(info.FieldType, info.GetValue(this));
      // The field name is used as the key, the value is initialized
      // as an empty wrapper around a plain Object instance.
      fieldValues.Add(info.Name, value);
    }
  }
}
```

Listing 3.5: Blackboard Base Class Initialization

```
public abstract class BlackboardBase : ScriptableObject {
  ...
  // Get value directly via string identifier
  public object GetValue(string valueName){...}
  // Use BlackboardProperty instance to identify value
  public object GetValue(BlackboardProperty property){...}
  // Same as above but also cast value to given type
  public T GetValue<T>(BlackboardProperty property){...}
  // Check if the value is valid(not null or default)
  public bool IsValueSet(BlackboardProperty property){...}

  // Only one setter,
  // Type information is necessary to check for valid assignments
  public void SetValue<T>(BlackboardProperty property, T value){...}
  // Set value to null or default value
  public void ResetValue(BlackboardProperty property){...}
  // Add listener action that is called every time the value is changed
  public void AddOnValueChangeListener(BlackboardProperty property,
                Action<object> onValueChangeAction){...}
...
}
```

Listing 3.6: Blackboard Base Class Getter And Setter

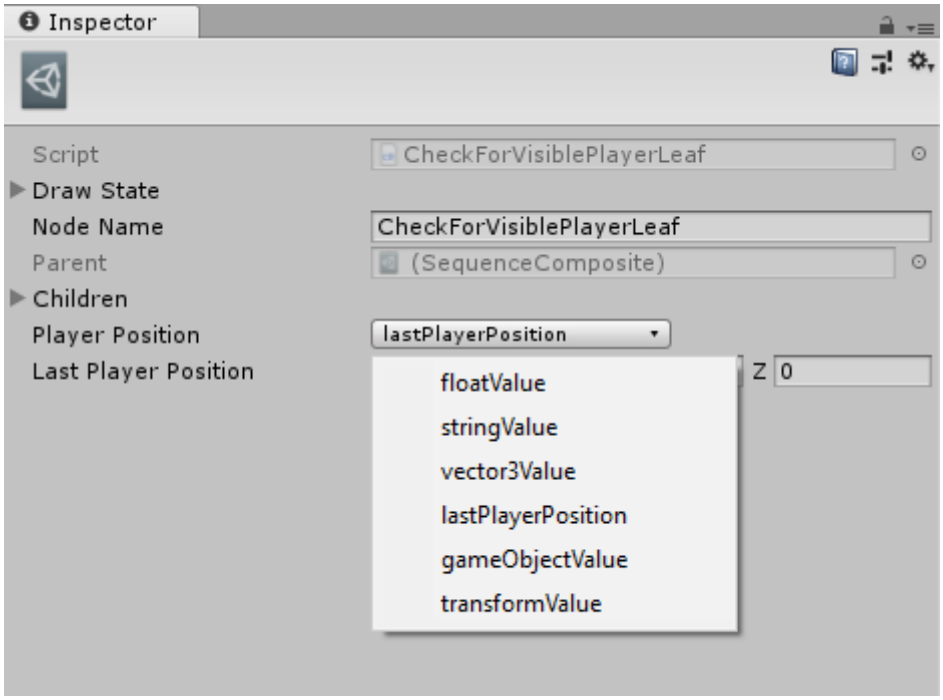

Figure 3.5: Blackboard Property PropertyDrawer

3.3 Conclusion

Behaviour trees are a powerful tool when developing AI behaviours. But to make them this powerful, they need to be paired with powerful tools. The implementation of the behaviour tree algorithm as described in 3.1 was for the most part quite straightforward, with no major problems. As for the visual editor however, the development was rather complex. The IMGUI proved difficult to use in the beginning and the sometimes, at best, mediocre documentation did not help. Once the editor was functional, however, it made creating and editing of behaviour trees very easy. The context menu, providing a searchable list of all node types, makes it extremely easy to find the wanted nodes and adding them to the current tree. Configuring nodes is also quite simple since it's done via the in-built inspector window, which is used for practically any other configuration in the Unity3D editor. Because of the design of the node methods, creating new node types can be very easy, depending on their features. At a minimum, only the *Update_Internal* method has to be implemented and the class needs the appropriate Attribute for its type.

Blackboard Regarding the blackboard implementation, while it is a rather strict system, having to declare every field manually, it provides a more stable operation. Using some sort of map data structure (*Dictionary* in this case) is a more flexible system but also prone to more errors, mainly misspelling of fields. It also makes debugging harder, because the values are hidden beneath another layer.

Performance The behaviour tree system described here performs quite well for smaller instances. The base code has no heap allocations except for initialization and the tree traversal is very fast. There was no test for a large application, with many and/or large behaviour tree instances running.

Improvements and Additional Features While the current implementation already provides the user with many features, there are still some improvements or additions possible. One of these would be the addition of more in-built node types, like a random *Sequence*, iterating randomly over its children. Especially useful would be a behaviour tree node, which, when updated, will run another behaviour tree instance inside a node. This would make compositing of behaviour trees even easier and allow re-usability of subtrees. To improve performance of traversing a behaviour tree, an event based system could be implemented. Instead of traversing the whole tree every update tick, it would save the currently executed node and directly jump to it again during the next update. If the Unity3D API would be thread safe, execution of behaviour trees in other threads could also improve its performance.

Chapter 4

Sensory System

To interact with the environment the AI has to have a way to collect information about it. Since it is purely virtual it could just take whatever information it wants and act based on that. This is not realistic behaviour however and can easily feel that way to the viewer. Another possibility is to simulate their input similar to how humans collect information about their environment: senses. Even though humans have several senses, it does not make sense to simulate all of them as well as to the extent that humans have them. Ian Millington and John Funge describe an relatively simple approach to a sensory system which can easily simulate three important human senses: sight, hearing and smell[10]. Their approach was implemented with minor adjustments and is described in the following section. There are a lot of different senses apart from the three named above, but most of them are usually not relevant to AI simulation or can be modelled with a simpler system. Touch for example can be simulated by hooking into a physics system which handles collisions already, taste on the other hand is usually much too complex to simulate correctly and can be replaced by a simpler system (if implemented at all).

4.1 Algorithm

The algorithm described is event-driven system where signals are sent and sensors are receiving them. A *SenseManager* has knowledge about all sensors currently active and receives any type of signals sent from other agents or the player. These signals are then evaluated and sent to the appropriate sensors.

Signal Signals are simple data structures and contain only four data fields.

- *Transform*: The position of the signal in the world.
- *Signal Source*: The *GameObject* which produced the signal.
- *Modality*: The type of signal (Sight, Hearing etc.) including values for the signal transmission speed, signal attenuation and signal maximum range.
- *Signal Strength*: A custom arbitrary value which defines the initial "strength" of the signal.

Sensor Sensors are responsible for receiving signals and providing an interface for other systems to receive the signals informations. Each sensor is specialized for a single type of signal. They can also hold additional configuration depending on the needs. This is especially important for the sense of sight.

SenseManager The original idea of the *SenseManager* was to have multiple instances, each responsible for some sort of region (room, area etc.). Here only one per *Scene* is used since there is no assumptions about the world structure, but the system can be adjusted to have several managers for different parts of the world. Its responsibilities are to receive any signals, process them and then route them to the appropriate sensors.

4.2 Chosen Senses

Three senses were selected for implementation (not including touch): Sight, Hearing and Smell.

- *Sight*: Sight is the primary human sense for recognizing and identifying objects at greater distances. In reality it is based on light hitting receptors in the back of the eyeballs which then relay that information to the brain that turns it into understandable information. Most of this process is too hard or not practical to model it completely but there are a few important parts. First, a pair of human eyes are positioned on the front of our heads and have a limited Field-Of-View (FOV), about 190 degrees horizontally and around 80 degrees vertically with both eyes combined [11]. Second, for all intents and purposes, with light travelling at the speed of light at around 300 000 000 meters per second (ignoring other effects), sight signals can be considered to arrive instantly, unless very large distances are simulated (i.e. solar system or galaxies). Third, while human eye sight can vary wildly with different physical conditions, the here described system only focuses on changes to the FOV and overall range of recognition.

- *Hearing*: Human hearing is based on sound waves reaching the ears which then are interpreted by the brain. In this context, the sound waves always travel through air, at a speed of around 340 meters per second. Humans have two ears, positioned on the side of the head, and together they almost allow for omnidirectional hearing. This system simulates such a configuration and does not take into account the differences in hearing with two separate ears.

- *Smell*: Smelling works by particles entering the nose, which then allows interpretation by the brain. The sense of smell is quite similar to hearing in function, but instead of sound waves travelling to the ear, it is particles travelling to the nose. These particles usually move at a very small speed, unless factors like wind are taken into account. The system treats smells as a stationary object, emitting signals regularly in a small range.

4.3 Implementation

Implementation of the described algorithm was very straightforward but there are a few things which had to be done to make it work with the Unity3D engine and a few additions, especially for the sense of sight.

4.3.1 Sense Manager

The *Sense Manager* is the central part of the algorithm, being responsible for receiving signals and distributing them to the appropriate sensors. Sensors have to register themself

```
1  public abstract class AbstractSensor : MonoBehaviour {
2    // Received signal strength must be higher than this value
3    // to be perceived by this sensor.
4    public float threshold = 0.1f;
5    // Alternative transform as origin of the sensor (i.e. head).
6    public Transform altTransform;
7    // Action for other systems to hook into this sensor.
8    public System.Action<Signal> OnNotify;
9
10   // Register this sensor with the SenseManager on game start
11   private void Start() {
12     SenseManager.Instance.AddSensor(this);
13   }
14   // De register when being destroyed
15   private void OnDestroy() {
16     SenseManager.Instance.RemoveSensor(this);
17   }
18   // Abstract method for each specified sense
19   public abstract bool DetectsModality(Modality modality);
20   // Called when signal reaches this sensor.
21   // Calls the OnNotify Action
22   public virtual void Notify(Signal signal) {
23     OnNotify?.Invoke(signal);
24   }
25 }
```

Listing 4.1: Sensor Class

with the manager to receive any signals. Signals can be added to the world via a *AddSignal(signal)* method. This method runs several checks on the signal and creates notifications for the sensors. These checks include:

- *Identity*: Check if signal source and sensor are the same.
- *Sensor Type*: Check if the sensor can even sense the signal.
- *Distance*: Is the sensor within the range of the signal.
- *Intensity*: Is the signal strength higher than the sensors threshold after travelling the distance.
- *Custom Checks*: Any additional, custom checks based on the signals modality.

If each of these tests are passed, a *SenseNotification* will be constructed and added to an ordered queue. These notifications consist of the original signal, the sensor that has to be notified and the time at which the sensor will be notified. This time is calculated by the distance the signal has to travel and its transmission speed.

4.3.2 Sensors

All sensor classes inherit from an abstract *AbstractSensor* class (see Listing 4.1), which itself inherits from the *MonoBehaviour* class so they can be attached to a Unity3D *GameObject*: All child classes (*SightSensor*, *HearingSensor*, *SmellSensor*) only implement the *DetectModality(modality)* method and, in case of the *SightSensor*, additional configuration data.

```
1  public struct SightConeConfig {
2    // Either of type Ray or Cast
3    // Defines which type will be used for checking
4    // if line of sight exists
5    public LineOfSightType lineOfSightType;
6    // Angle of the cone, in degrees
7    public float angle;
8    // Custom attenuation of the light signal
9    // Models the peripheral vision of human eyes
10   public float signalAttenuation;
11   // Decides which sight cone has priority over another
12   public int priority;
13 }
```

Listing 4.2: SightConeConfig Class

4.3.2.1 Sight

The *SightSensor* class contains additional configuration data in a *SightConfig* class which inherits the *ScriptableObject* class. Each *SightConfig* asset stores a list of *SightConeConfig* (Listing 4.2) instances. This allows to configure multiple cones with different signal attenuation rates and line of sight modes. These configurations are sorted by priority to properly handle overlapping of cones. Since the *SightConfig* class is a *ScriptableObject*, one can easily configure multiple types of configuration, save them individually and use when needed.

SightModality The most complex part of the sense of sight is the SightModality, or rather the custom checks to test if a signal (or rather its source) can be seen. Listing 4.3 shows the default configuration and high-level code of the extra checks. Important to note is, while the transmission speed is only set to the value of 100 000 instead of the much more accurate 300 000 000, that is of no practical concern since most games never simulate the world even that far.

4.3.2.2 Hearing

The hearing sensor is very simple since there is no need for additional configuration. Hearing is assumed to be always omnidirectional in this implementation, which makes the signal the only limiting factor.

HearingModality The hearing modality in Listing 4.4, like the *SightModality* in Listing 4.3, also has some extra checks in place to determine if the signal will be received. Since sound can travel around corners or through walls, the extra check will try to estimate the additional attenuation of the obstacle and check the resulting signal strength with the sensors threshold again.

```
public class SightModality : Modality {
  ...
  public SightModality() {
    // Fake maximum range
    maximumRange = 30.0f;
    // Signal never degrades by default
    attenuation = 1.0f;
    // Very fast speed
    transmisssionSpeed = 100000.0f;
  }
  // Additional signal checks
  public override bool ExtraChecks(signal, sensor) {
    foreach (SightConeConfig config in configurations) {
      if (IsInAngle(signal, sensor, config.angle)) {
        bool isVisible = IsVisible(config.lineOfSightType, signal, sensor);
        float signalStrength = signal.strength - config.signalAttenuation;
        bool overTreshold = signalStrength >= sensor.threshold;
        if (isVisible && overTreshold) {
          return true;
        }
      }
    }
    // Signal not visible
    return false;
  }
  // Checks if the signal is in one of the sensor's configured cones
  private bool IsInAngle(Signal signal, Sensor sensor, float angle) {
    ...
  }
  // Checks if the given signal is visible.
  // Can use a RayCast or a SphereCast.
  private bool IsVisible(LoSType type, Signal signal, Sensor sensor) {
    ...
  }
  ...
}
```

Listing 4.3: SightModality Class

```
public class HearingModality : Modality {
  ...
  // Constructor setting defaults values
  public HearingModality() {
    // Fake maximum range
    maximumRange = 20.0f;
    // Signal gets weaker over distance
    attenuation = 0.75f;
    // Default speed of sound
    transmisssionSpeed = 340.0f;
  }
  // Additional signal checks
  public override bool ExtraChecks(signal, sensor) {
    // Check if there is an obstacle between sensor and signal
    if (HasObstacle(signal.transform, sensor.transform)) {
      // Check if dampened signal still is strong enough
      if (signal.strength - GetObstacleAttenuation() < sensor.threshold) {
        // Signal not audible
        return false;
      }
    }
    // Signal audible
    return true;
  }

  // Raycast check for obstacle check
  private bool HasObstacle(signal, sensor) {
    Vector3 direction = (signal.position - sensor.position).normalized;
    int hitCount = Physics.RaycastNonAlloc(sensor.position, direction,
                          obstacleHits, maximumRange);
    return hitCount > 1;
  }

  // Method for getting additional obstacle attenuation
  // Only returns a token value, but can be expanded
  // to check for things like wall thickness or similar
  private float GetObstacleAttenuation() {
    return 0.1f;
  }
}
```

Listing 4.4: HearingModality Class

```
1  public class SmellModality : Modality {
2    public SmellModality() {
3      // Small range
4      maximumRange = 5.0f;
5      // No degradation over the short distance
6      attenuation = 1.0f;
7      // No transmission speed necessary with small distances
8      transmisssionSpeed = 0.0f;
9    }
10 }
```

Listing 4.5: SmellModality Class

4.3.2.3 Smell

As described in 4.2, the sense of smell is functionality wise very similar to the sense of hearing, with the main difference that a smell signal has no transmission speed (without any kind of wind or similar effects). To enable smell signals to reach a sensor, they have to be repeatedly be send over a period of time so the sensor can pick them up when they are close by. This behaviour is not implemented by the system itself but it is trivial to do so. The *SmellSensor* required no special code and the *SmellModality* in Listing 4.5 only defines the default values.

4.4 Conclusion

The Algorithm described by Ian Millington and John Funge[10] was easy to understand and implement, with only minimal changes. The structure of sensor manager, sensors, modalities and signals also translates very well into the workflow of the *Unity3D* engine. Each modality and sensor can be customized easily to suit the user's need. A drawback of this system, however, is that signals, especially sight signals, have to be sent regularly. The upside of that is, that every signal could, in theory, be customized to suit the moment. Another drawback is the way how sound is portrayed in this system, especially for fast moving sensors. In the scenario where a fast moving sensor is moving past a hearing signal, the sensor will receive the signal after the calculated time, with the problem being that it will move very quickly away from the signal source. It could move past doors or corners before it receives the signal. Also the attenuation of sound through solid objects is not modelled at all, the small addition in Listing 4.4 was implemented as an approximation. Overall this system gives the user great flexibility in configuring and using the sensor system to simulate humanoid senses.

Chapter 5

Spatial Query System

AI agents in all kinds of games or dynamic simulations need to gather information about their environment. With this information they can make decisions about what to do, where to move towards or away from or which target is currently more important and many more. While some of this information can be generated manually or embedded into the environment itself, at some point it becomes too cumbersome to manually control and maintain. If the environment changes, much of the information could be invalidated and have to be placed again. With a dynamic runtime system generating points of interests of any kind many of those problems can be avoided, although not without cost. Such a system needs additional computing time and a robust architecture as well as an easy to use editor. In the paper "Asking the Environment Smart Questions"[12] Mieszko Zielinski describes the general algorithms and design ideas. The general idea is to answer some specific question, such as *Where is the best place to hide?*, *Which enemy should I attack?* or maybe *Where do I not want to go?*.

5.1 Design

Each query consists of one generator, generating the points of interest, and one or more tests, which filter and/or score these points. Each of them is configurable and easy to understand. Zielinski and his team defined five basic questions they want to answer with their implementation[13]:

- *What to generate?*: What is the overall purpose of the query, i.e. find cover, enemy locations etc..
- *Who's asking?*: What object is asking the question or what is the query's context. This can be an AI agent but also any location in the world.
- *Where to look?*: Where are the generated points supposed to be? Around the context, on the NavMesh, in a specific area etc..
- *Which items are good enough?*: Which points are invalid and are not interesting at all? When searching for cover, any visible points can be discarded.
- *Which items are better?*: What would be the best possible point for our needs? Points need to be scored via given criteria to find these points.

5.1.1 Context

The context of the query is important, since it defines the "subjective world view"[14] for the query. It can be pretty much anything in the world, it is not limited to AI agents. If a world position is used, the query could find out if this position is visible to another position (i.e. for hiding) or, if another actor is used, one could find a suitable flanking position (i.e. for attacking).

5.1.2 Generator

Generators are the initial stage for any spatial query. Their purpose is to generate points in the world to be used later for subsequent filtering and scoring. These points can be arbitrarily chosen, although certain patterns are useful. A few examples of these patterns could be a ring of points around the context, all visible enemy positions or points arranged in a grid on the NavMesh.

5.1.3 Test

Tests operate on the points generated by the query's generator, filtering out any points which are not good enough while also scoring them based on their criteria. To improve usability, all tests are capable of filtering and scoring and can be configured to either one or the other, or both. Since the tests can vary in their computational cost, each test is given a fixed, arbitrary sorting value, which is supposed to represent its cost relative to the other tests. This allows running cheap tests first to minimize execution of more expensive tests. Their operation consists of two stages:

1. *Filtering*: Order all tests by their computational cost and then run all, which are configured as filtering tests. Each test is run sequentially on all remaining valid points until all are done.

2. *Scoring* After all invalid positions are filtered out, they are scored by all tests configured to score them. As with the filters, these tests are also sorted by their computational costs and they are run sequentially. After a test has scored all points, the resulting scores are then normalized to a value between 0 and 1 (both inclusive). This is necessary to remove any differences which occur by using different scoring methods. After normalization each score can be further changed with a configured score type and score multiplier. This allows to prioritize certain tests over others.

Once these two stages are finished, the query itself is done and its results can be accessed. The query provides a list for all valid points sorted by their score, a list for all invalid points and the highest scoring point.

5.2 Implementation

The SQS consists of three main classes: *SpatialQuery*, *BaseGenerator* and *BaseTest*. All three inherit from *ScriptableObject* so they are supported by Unity's serialization system. Several classes inheriting from *BaseGenerator* or *BaseTest* were implemented to serve as the base set of functionality. Furthermore, a semi-custom editor for the *SpatialQuery* assets was written to improve usability.

5.2.1 Spatial Query

The *SpatialQuery* class is the central class to the SQS where the selected generator and tests are combined and executed. Once finished, the results are available for use. Depending on the configuration of the query and its tests, especially the amount of generated points, the query can take relatively long amounts of time since every position is filtered and possibly scored well. To counteract this, the query's test execution is setup to be split over multiple update ticks. This done by only processing a certain number of positions per update tick, controlled via the *processedPointsPerTick* variable.

5.2.1.1 Editor

The SQS is designed to be used mostly by non-programmers and thus needs an easy to use and understand UI. To satisfy this need, a GUI based on the *Inspector* editor window from the *Unity3D* engine was developed. Figure 5.1 shows an example query in the editor: On the top is a dropdown menu for selecting the query's generator type. Below that, there are all custom variables of that generator type, followed by the BaseGenerator variables. On the bottom, divided by a line, is the list of tests. Each test was displayed in a fold-out control, with their name and test type as title. The name can be customized but defaults to the test class name. Each entry in that list has a small button with an "X" in it, which allows the user to delete single entries. This button also opens a simple dialogue window to prevent accidental deletions. If the test is folded out, as seen in the image, all variables of the test are displayed. Finally, always at the end of the list, is a button labelled "Add". This button opens a small context menu, listing all available test types. By clicking on of these types, it will be added into the list to the query. This editor was realized by

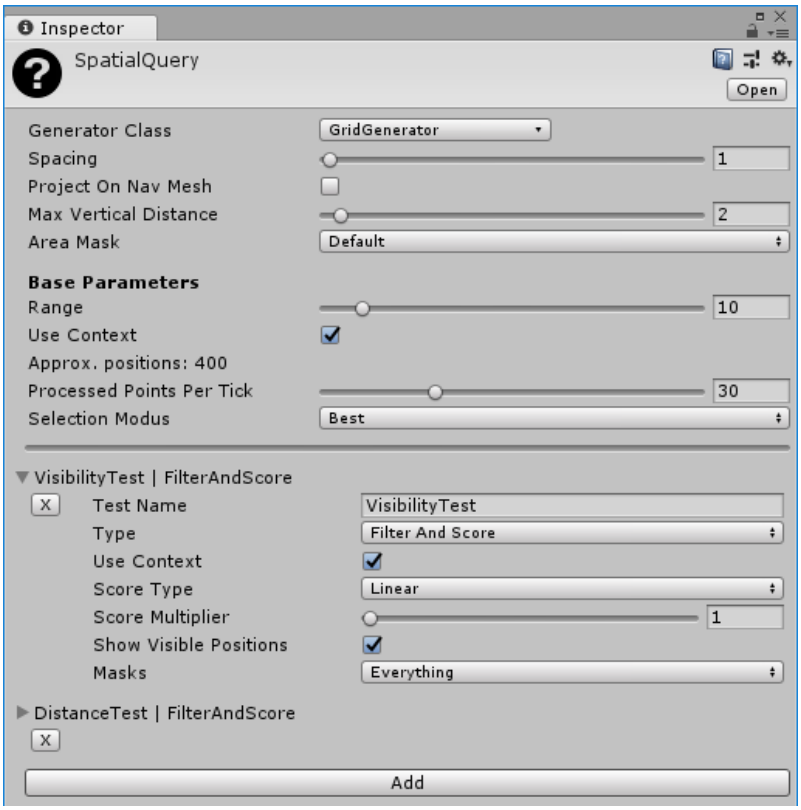

Figure 5.1: Example Spatial Query Instance With Test

```
1  [CustomEditor(typeof(SpatialQuery))]
2  public class SpatialQueryEditor : Editor {
3    ...
4    public override void OnInspectorGUI() {
5      // Target contains the actual object instance
6      SpatialQuery query = (SpatialQuery)target;
7      // Update the serialzed object to display the current state
8      serializedObject.Update();
9      // Display all parameters of the generator
10     DisplayGeneratorConfig(query);
11     // Simple horizontal line to separate generator and tests
12     EditorGUILayout.TextArea("", GUI.skin.horizontalSlider);
13     // Display the list of tests including additional buttons
14     DisplayTestList(query);
15     // Apply any changes made to the object instance
16     serializedObject.ApplyModifiedProperties();
17   }
18   ...
19 }
```

Listing 5.1: SpatialQuery Class Custom Editor

implementing a custom editor class for the *SpatialQuery* class, as seen as in Listing 5.1.

5.2.1.2 Debug Utilities

While creating a query it is very important to see it in action easily. Although the creation process is quite simple, it is hard to imagine how the result actually will be. This is also very important when creating and testing new generators and tests. To this end, a simple class *QueryDebugDisplay*, inheriting from *MonoBehaviour*, was created. When given a *SpatialQuery* instance as a reference, it will execute and display the results in the game world, including scores, if available. This script also updates the result whenever the object it is attached to, is moved. This allows for easy configuring, testing and bug searching. In Figure 5.2 an example query is displayed in this way. The query is a simple *Grid* generator with a *VisibilityTest* and a *DistanceText* as tests. All non-visible positions are invalid and every position is score based on its distance to the context, the green capsule in the middle. Each position generated is generally displayed via a wire sphere, with the colour of each sphere is dependent on its state:

- *Valid*: The colour of valid positions is interpolated between red and green, with the alpha being the score of the position divided by the best positions score.
- *Invalid*: Invalid positions have a blue colour.

Additionally, all valid positions have a small number next to them displaying their current score. All these colours can be configured to match the need of the user and scene. This display gives immediate feedback to the user and is easy to understand.

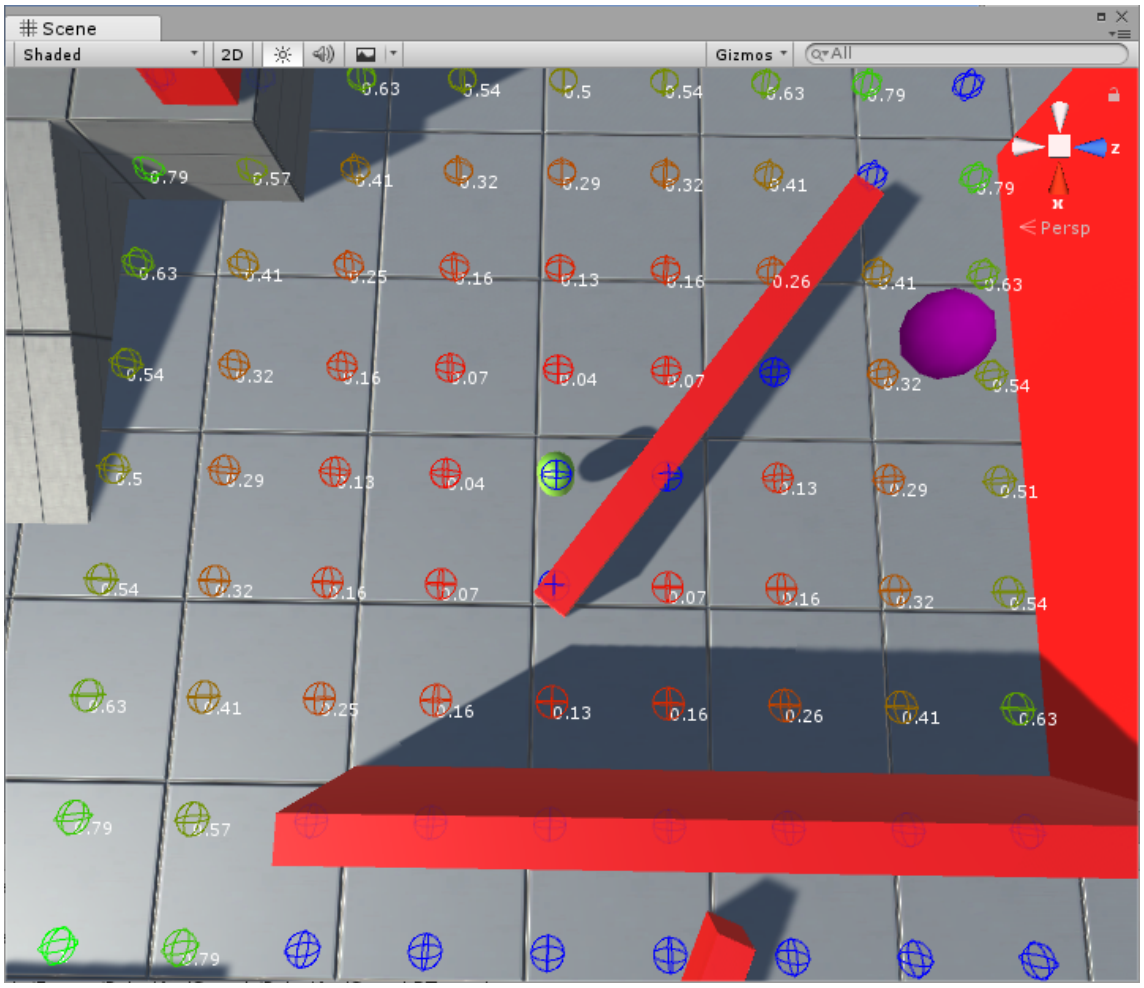

Figure 5.2: SpatialQuery Debug Display

```
1  public abstract class BaseGenerator : ScriptableObject {
2    ...
3    // Filled by "GeneratePositions_Internal".
4    protected List<QueryPosition> latestGeneratedPositions = new
5    // Generates new set of points.
6    public void GeneratePositions() {
7      latestGeneratedPositions.Clear();
8      GeneratePositions_Internal();
9    }
10   // Must be overridden by child classes.
11   protected abstract void GeneratePositions_Internal();
12   // Returns approximated amount of positions generated by this generator.
13   public abstract int GetGeneratedAmount();
14 }
```

Listing 5.2: BaseGenerator Class

5.2.2 Generator

The purpose of a SQS generator is to arbitrary points according to a given pattern. The base class (see Listing 5.2) contains all fields and methods common to all generator implementations. Child classes must overwrite the *GeneratePositions_Internal* method which is responsible for actually generating the specific points and filling the *latestGeneratedPositions* list with them. Additionally the *GetGeneratedAmount* is also marked abstract and its purpose is to approximate the amount of points which will be calculated. This is only used for display in the editor, as an indicator of computational cost. The following list contains a few examples of generators which were implemented, including a description and possible usage scenario.

- *Tag*: Generates positions for all game objects within range and having a certain tag. This can be used to create points for existing static objects in the world, such as patrol points or cover locations, but also for dynamic, such as all enemies/friendlies in range. See Figure 5.3 for an example image.
- *Grid*: Generates points in a grid, based on the range of the generator and a given spacing between each point. These points can then be attempted to be placed on the NavMesh, if the NavMesh is in a certain radius of the points but, if not, they will be discarded. This can be used for many queries where information around an object is desired. This could be nearby cover, flanking positions etc.. See Figure 5.4 for an example image.

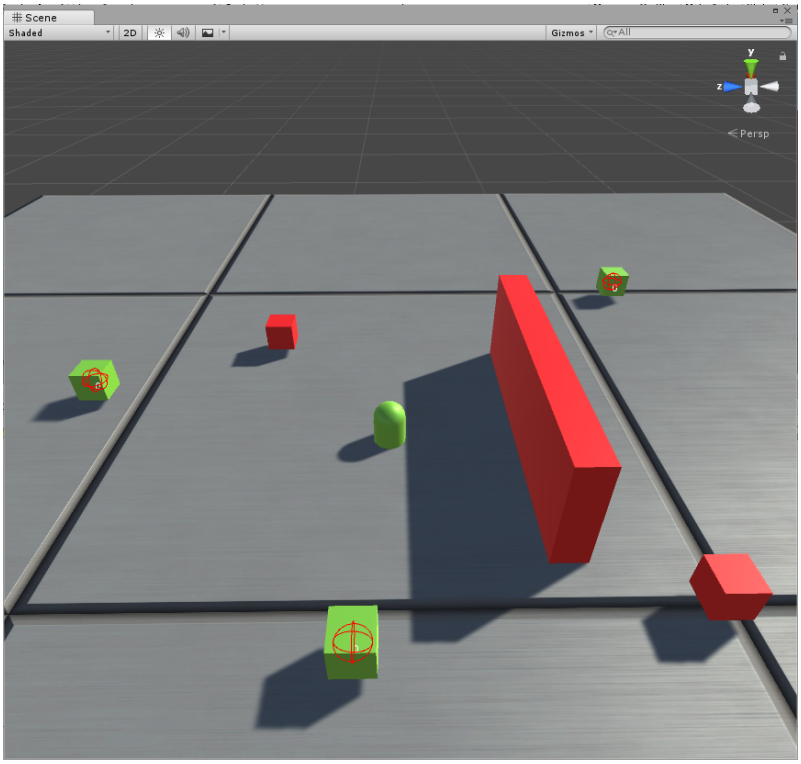

Figure 5.3: SpatialQuery Tag Generator

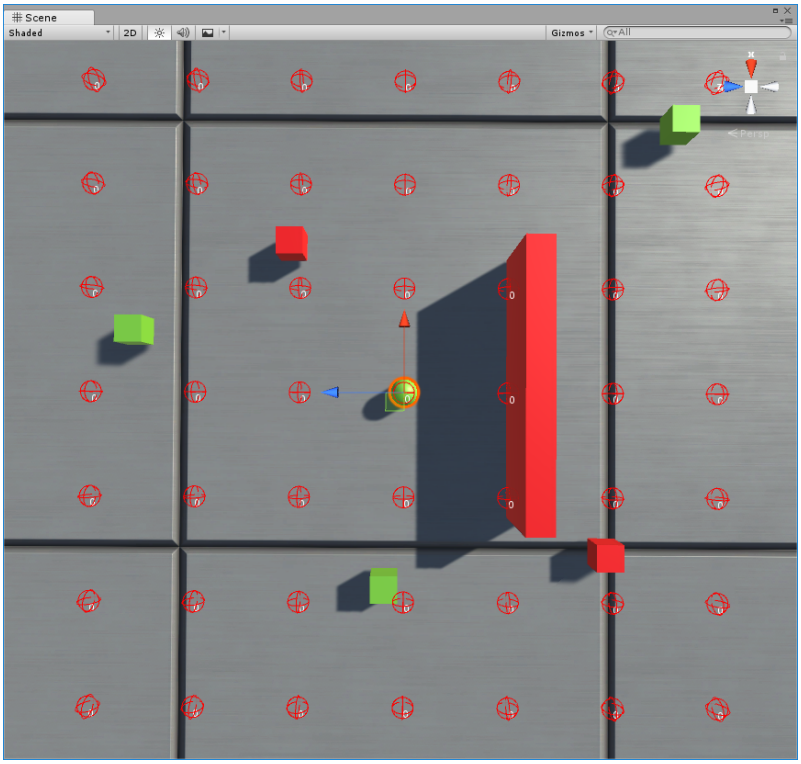

Figure 5.4: SpatialQuery Grid Generator

- *Ring*: Generates points in a configurable ring. This ring has a inner and outer radius, a angle spacing as well as a radius spacing. Additionally these points can also be placed on the NavMesh, just as the *Grid* generator. Possible usage scenarios could be locating flanking positions or follow a target secretly at a distance. See Figure 5.5 for an example image.

5.2.3 Test

All test classes have to inherit the *BaseTest* class which contains common functionality and provides a number of abstract and virtual methods for overriding. It takes care of iterating over each position and normalizing scores, if necessary. To create a new test, the developer only has to implement three methods:

- *TestPosition*: Filters a single position and decides if it is valid or invalid.
- *ScorePosition*: Calculates the score of a single position with the value being not normalized.
- *GetSortPriority*: Returns a fixed value which is used for sorting the tests before running a Spatial Query. Their purpose is to order the tests based on computational cost, so cheaper tests run before more expensive ones.

Additionally there are two virtual methods for initialization:

- *PrepareTest*: Run once before this test filtered any positions.
- *PrepareScoring*: Same as the *PrepareTest* method, this method is run once before this test starts scoring positions.

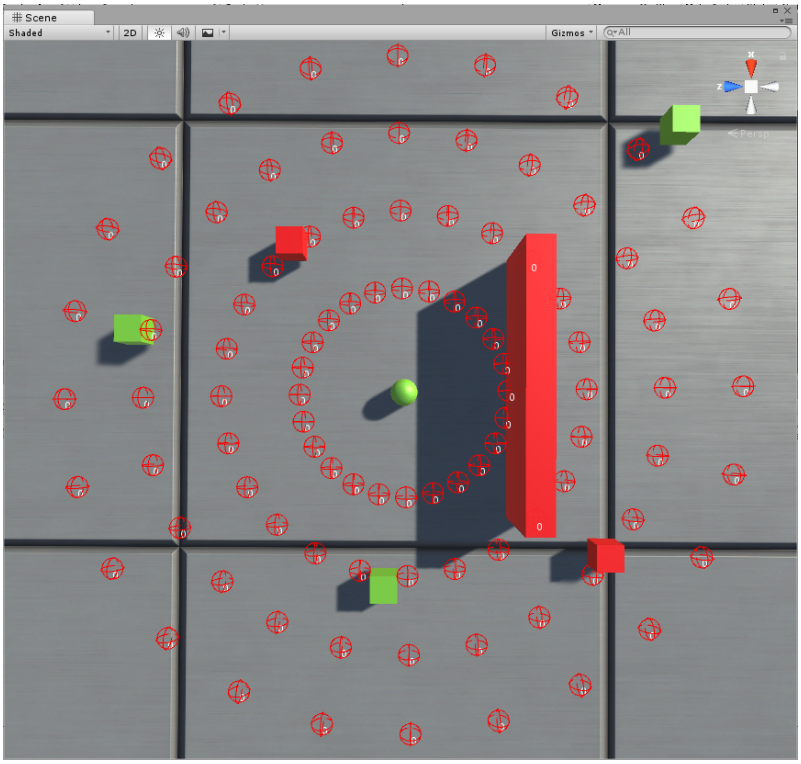

Figure 5.5: SpatialQuery Ring Generator

And lastly a *NormalizePositions* method, which normalizes the previously calculated scores by using the highest score as reference. This will normalize all scores into a range of zero to one after which a score multiplier is applied, which allows prioritization of certain test scores. Listing 5.3 shows these methods and their signature in more detail.

```
public abstract class BaseTest : ScriptableObject {
  ...
  // Initialize testing
  protected virtual void PrepareTest() { }
  // Iterate over positions from startIndex to endIndex
  public void TestPositions(int startIndex, int endIndex) {
    PrepareTest();
    // Iterate over positions and test with "TestPosition"
    ...
  }
  // Tests a single position and marks it invalid
  // if it fails
  protected abstract void TestPosition(QueryPosition position, int index);

  protected virtual void PrepareScoring() { }
  // Iterate over positions from startIndex to endIndex
  public void ScorePositions(int startIndex, int endIndex) {
    PrepareScoring();
    // Iterate over positions and score with "ScorePosition"
    ...
    // Normalize scores after all are calculated
    NormalizeScores();
  }
  // Calculates the not normalized score for a single position
  protected abstract void ScorePosition(QueryPosition position, int index);
  // Normalizes all calculated scores
  private void NormalizeScores() {
  ...
  }
  // Returns manually entered value used for sorting
  public abstract float GetSortPriority();
}
```

Listing 5.3: BaseTest Class

There were four tests implemented as built-in tests:

- *Visibility*: The visibility test checks if a position is visible by using a raycast from the context query to the generated position. A boolean switch is available to filter out either visible or not visible positions. Since the filter operators only on a boolean basis (visible and not visible), scoring is practically not possible and as such does not change the score of the position. See Figure 5.6 for an example image.

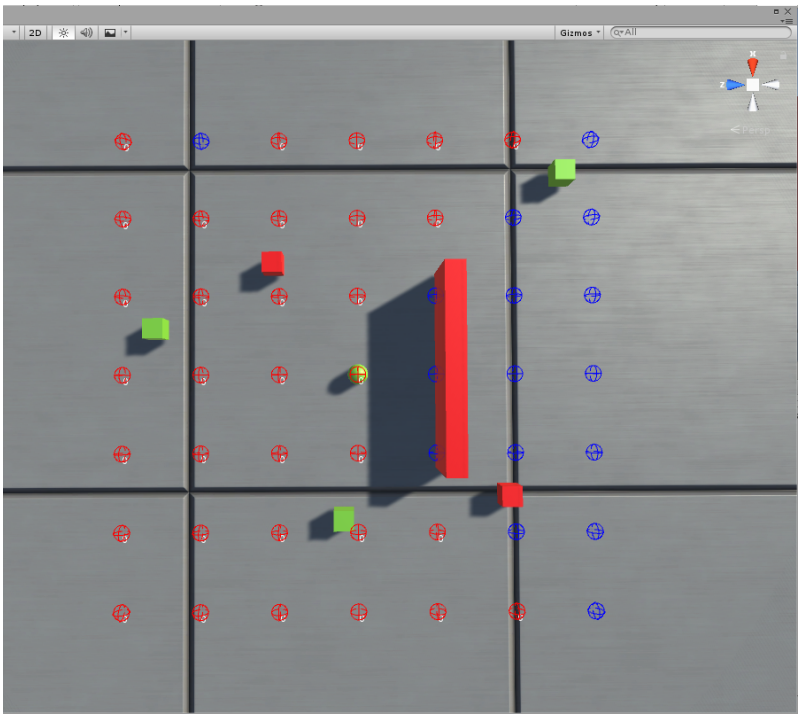

Figure 5.6: SpatialQuery Visibility Test

- *Path*: The Path filter checks if there is a path on the NavMesh from the query context to the generated position. As with the *Visibility* filter, it can filter out reachable or not reachable paths. For scoring it calculates the overall length of the calculated path and divides it by the maximum range of the generator. This can be configured to prefer shorter or longer paths via a float multiplier. See Figure 5.7 for an example image.

- *Distance*: Filters positions based on their distance to the context query. This can be configured with a minimum and maximum distance. Scoring is similar to the Path filter, it considers the direct distance from position to query context and divides it by length given by minimum and maximum distance. This can also be configured to prefer closer or further positions. See Figure 5.8 for an example image.

- *NavMesh Border Distance*: Filters positions based on their distance to an edge in the NavMesh. It functions exactly as the *Distance* filter but instead of using the query context to compare to, it uses the function *NavMesh.FindClosestEdge(...)* and checks the distance based on this. See Figure 5.9 for an example image.

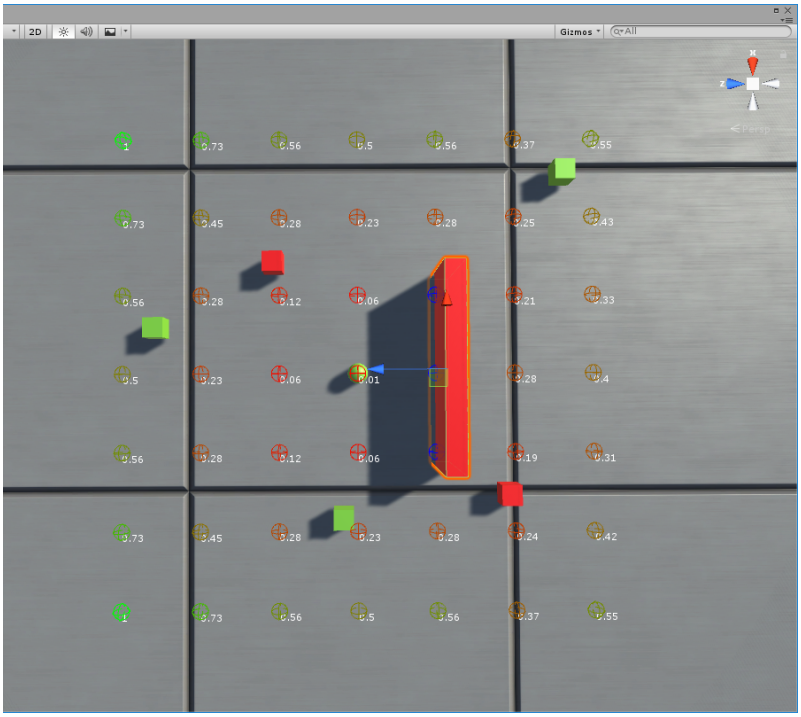

Figure 5.7: SpatialQuery Path Test

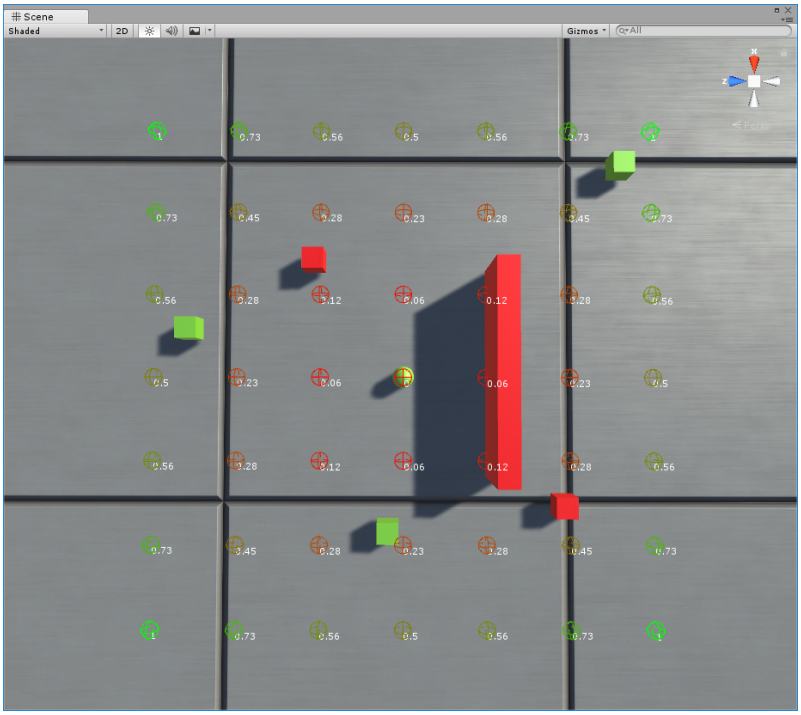

Figure 5.8: SpatialQuery Distance Test

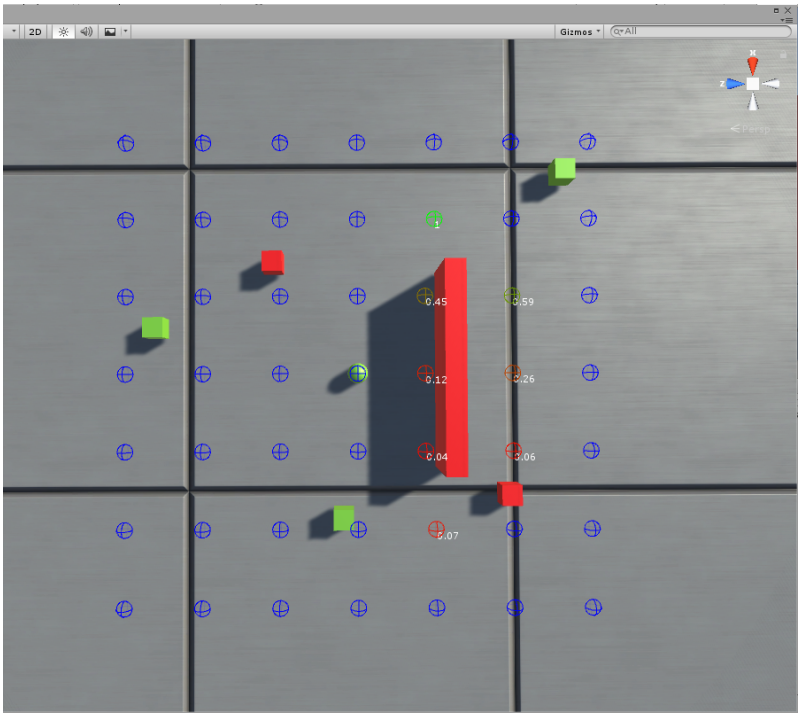

Figure 5.9: SpatialQuery NavMesh Border Distance Test

5.3 Conclusion

The SQS is a powerful system for generating information about the environment and can be used in a multitude of situations. Given the proper generators and tests, new queries can be created relatively quickly and, more importantly, without the need to write additional code. Embedding the whole SQS into *ScriptableObjects* also improves the whole usability since the queries can be referenced by other *ScriptableObjects* or *MonoBehaviours* without requiring extra code. Creating new generators and tests is relatively simple due to their clear base classes and not having to override a lot of methods.

Editor Through the custom editor all options are visible in one single window and since all of them are realized via *Unity3D* in-built controls, they are also easy to operate. A problem that sometimes occurred, was that, since the parameters are all looking very similar and there can be a lot of them, it can be easy to miss or mix up certain parameters. This could be alleviated by creating custom controls, separators and more colours but was not implemented due to time constraints.

Performance Computational cost is a key factor when creating a *SpatialQuery*. Ideally this work would be offloaded to another thread but cannot since most of the Unity3D API is not thread safe and cannot be accessed from another thread[15]. Through splitting the execution to run over multiple update ticks instead of all at once, the performance impact can be kept to a minimum. Although this takes more time, it will not be very noticeable if the query is configured properly.

Chapter 6

Example Configuration

To evaluate the previously described AI technologies and their implementations an example configuration was developed. The idea is to combine the three technologies (*Behaviour Tree*, *Sensory System*, *Spatial Query System*) to test their effectiveness and usability. To achieve this, a guard-like AI agent was implemented, with its task being to patrol around in an example scene, chase the player if he is found and search for him if the player was lost in a chase.

6.1 Design and Implementation

The overall design goal is to create a patrolling guard of some sort, which, if it senses the player, chases him until the player is caught. If the player can hide during a chase, the guard will search for the player in nearby possible hiding places and if no one is found, it will continue the patrol. Figure 6.1 shows the overall state diagram of this design. The following sections describe the role of each of the used components and what was necessary to develop the described behaviour.

6.1.1 Behaviour Tree

The behaviour tree combines all other parts into the guard's behaviour. Its structure is kept simple, there are three sub-trees, each describing one state of the guard. As described above these three are:

- *Patrol*
- *Chase*
- *Search*

The state changes are controlled via several *BlackboardValue* decorators, all checking the *currentState* value on the blackboard. Additionally the placement of the Sense service node is of importance. This and each state will be described in more detail in the following sections.

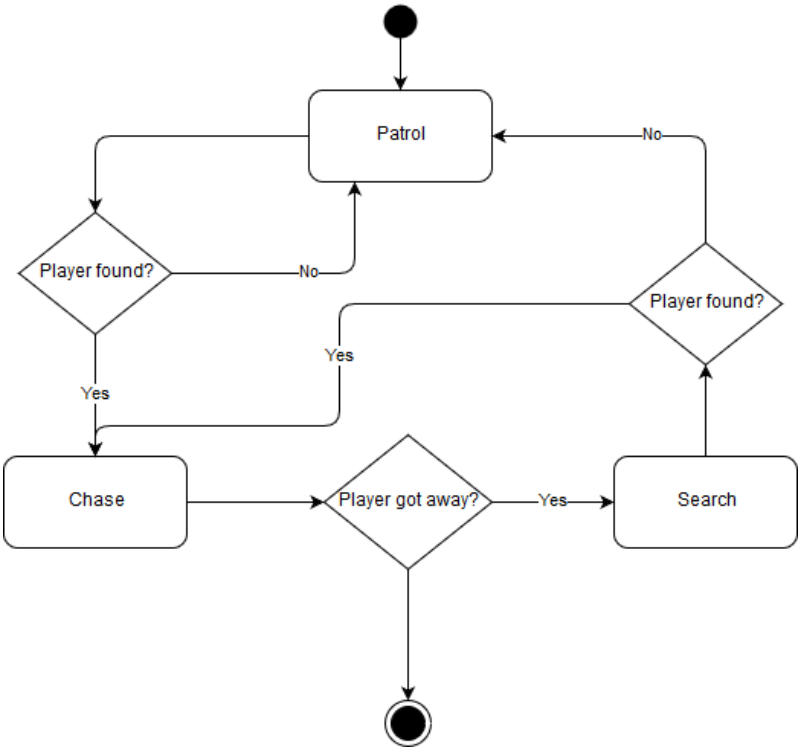

Figure 6.1: Example Configuration Design

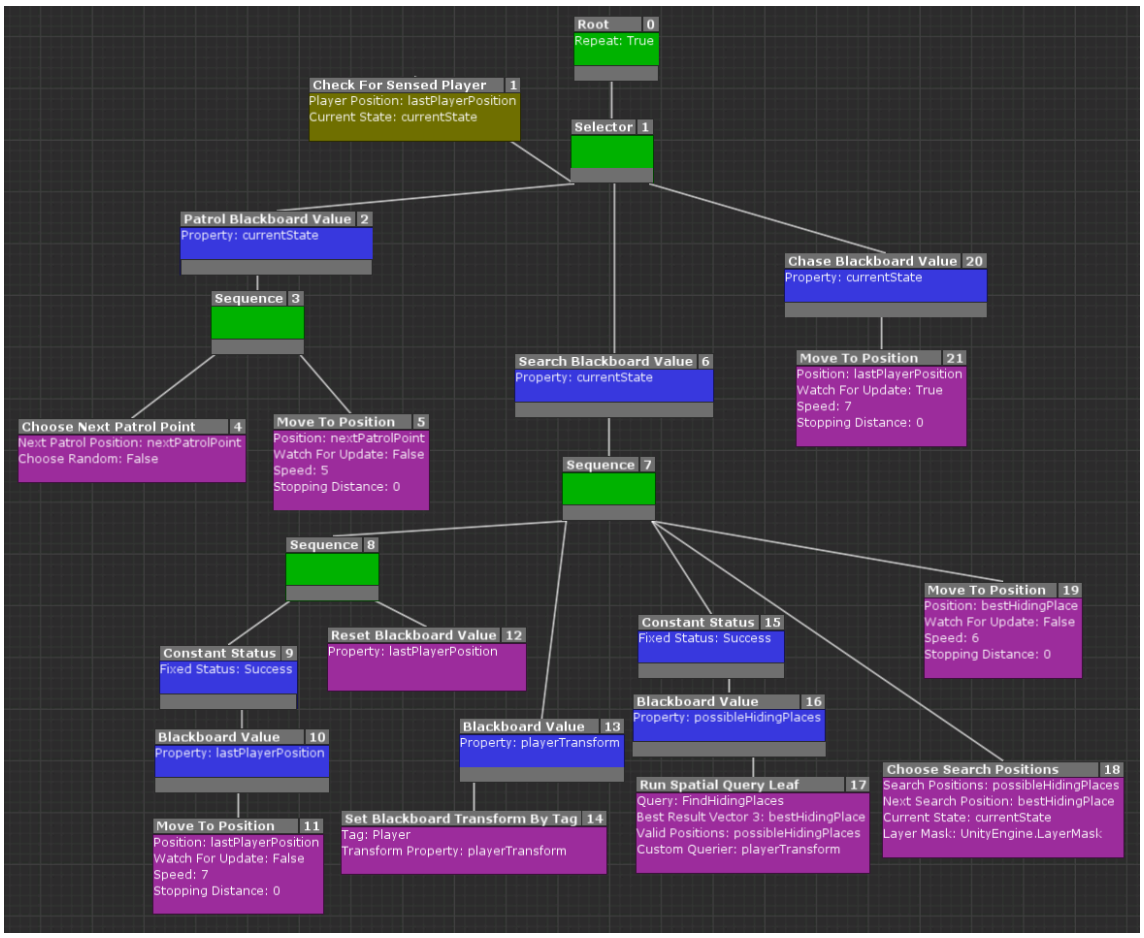

Figure 6.2: Example Configuration Complete Behaviour Tree

6.1.1.1 Patrol

The guards default behaviour is to patrol in the level until it finds the player. This is accomplished by defining multiple "patrol points" in the level which the guard uses to patrol between them. The actual patrol behaviour consists of a *Sequence* with a *Choose Next Patrol Point* and a *Move To Position* node, see Figure 6.3. First the next patrol

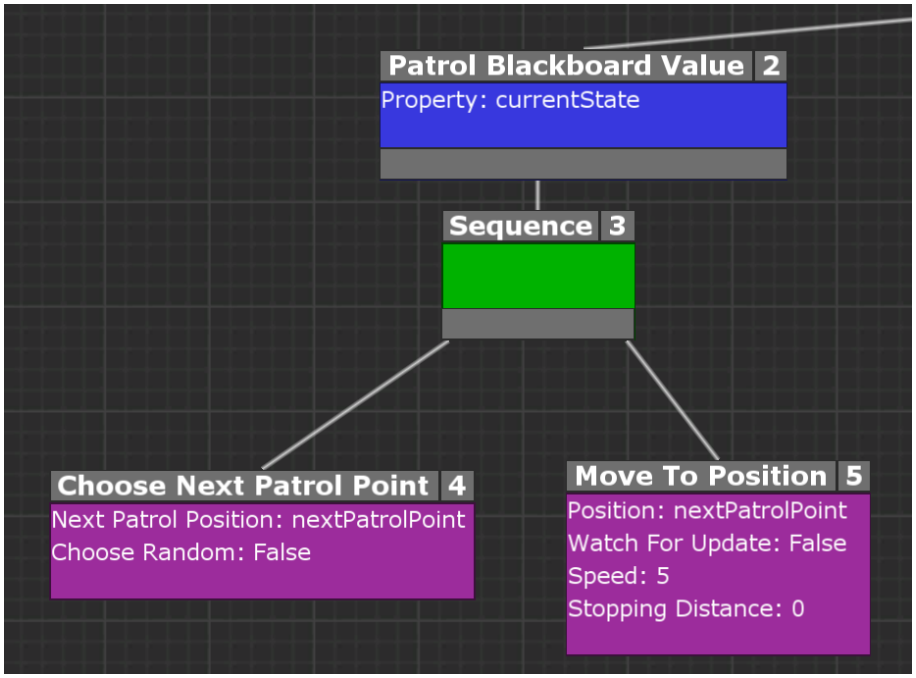

Figure 6.3: Example Configuration Behaviour Tree Patrol Sub Tree

point is chosen. This can happen in two different ways:

- *Random*: Choose a random patrol point to move towards. This can produce strange movement and is more useful if used in smaller areas, as larger ones allow for very inefficient movement (i.e. move from one corner to another an back again).
- *Range Based Queue*: The patrol points are divided into visited and unvisited points. The guard now always chooses the closest, unvisited patrol point. Once all are visited, their status is just reset so that all are unvisited again. This algorithm produces movement which is comparable to patrol route, since all points are all visited once before repetition starts. If the end point is not the start point, the route can change when the guard is finished moving.

The default modus is the range based queue and also the used modus here to imitate a guard moving around an area with defined checkpoints. The chosen patrol point position is saved on the blackboard with the key *nextPatrolPoint* which is used in the next node to move towards it.

6.1.1.2 Sensory System Component

The guard has the *Sight* and *Hearing* senses which are constantly running. Their information is accessed via a *Check For Sensed Player Service* node which is attached to the *Selector* directly under the root node, as seen in the Figure 6.4. This means it will be updated

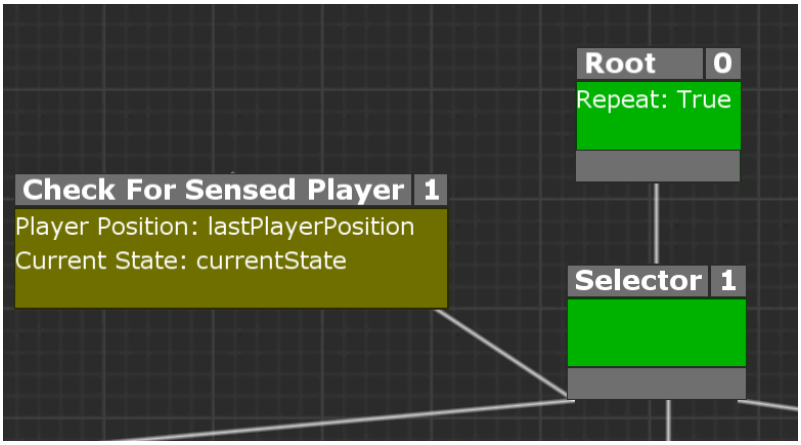

Figure 6.4: Example Configuration Behaviour Tree Agent Sense Service

every time when the *Selector* is updated, which in this case is every update tick. The service checks if the *Sight* and *Hearing* sensors detected anything and, if so, test whether the signal originates from the player. If that is also true it sets the blackboard value for the *lastPlayerPosition* key to the position of the signal but also changes the value of the blackboard *currentState* key to the appropriate value: *Chase*, if the player is detected, or *Search* if the player was detected earlier but now is not any more.

6.1.1.3 Chase

Chasing the player is accomplished through a simple *Move To Position* node which moves the guard to the position provided by the blackboard value with the key *lastPlayerPosition* (see Figure 6.5). This position is provided by the guard's senses (see 6.1.1.2). Since

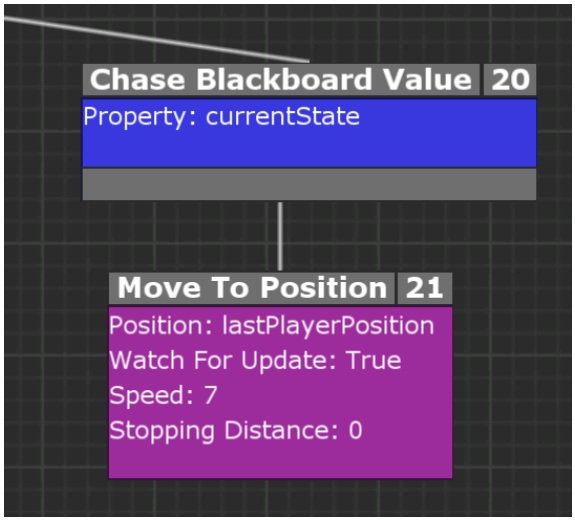

Figure 6.5: Example Configuration Behaviour Tree Chase Sub Tree

the position of the player can change while the guard is moving, the *Watch For Update* parameter is set to true, so the node updates the target position if the blackboard value changes. Finally the *Speed* parameter is set to a slightly higher value than the patrol or search speed to give the guard a chance to catch up with the player.

6.1.1.4 Search

The *Search* subtree (see Figure 6.6) is the most complex. It consists of three separate behaviours:

- *Move to last known player position*: This is supposed to move the guard to the last known position of the player. The assumption here is that the player will be close to this position which makes this is the first, easy to choose search position.
- *Run Spatial Query to find possible hiding positions*: Runs a pre-configured *Spatial Query* (see 6.1.3) to estimate possible hiding positions of the player. These positions are saved on the blackboard.
- *Choose and move to all hiding positions*: First all remaining possible hiding positions are checked if they are visible and, if so, they are removed from the list. If there are any positions left, the agent will move to the next one on the list and repeat until all positions are removed from the list.

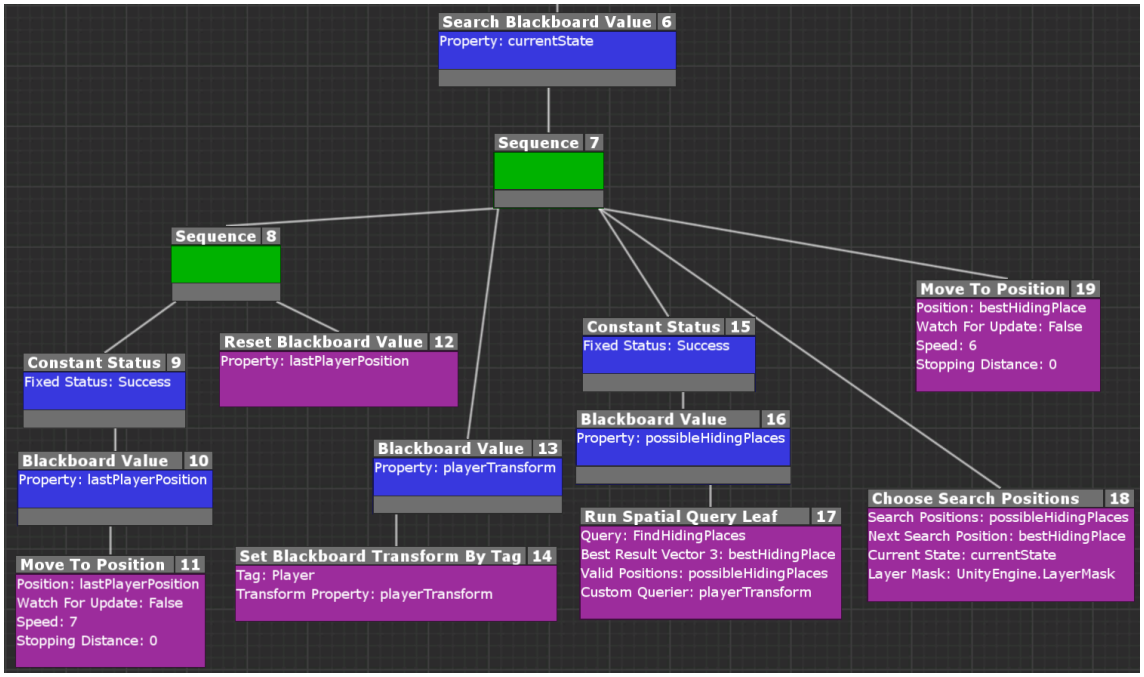

Figure 6.6: Example Configuration Behaviour Tree Search Sub Tree

6.1.2 Senses

The guard agent has the *Sight* and *Hearing* senses to perceive any signals sent by the player. The player will constantly send out *Sight* signals and also *Hearing* signals, but only when he moves.

6.1.2.1 Sight

The *Sight* sensor configuration consists of three different cone configurations. The first has a wide angle of 180 degrees in front of the agent with a signal attenuation of 0.8 to simulate

the peripheral vision. The second has a smaller 90 degree cone with an attenuation of 0.2 to simulate object close to the focus of the eye but not within it. The last has a small 30 degree cone with zero attenuation. This is the currently focused area of the sensor where every signal is perceived. The signals' modality configuration is based on having clear and sunny weather. Maximum range is high, the attenuation is one, which means no signal degradation, and the transmission speed is set very high.

6.1.2.2 Hearing

The *Hearing* sensor's configuration is only the threshold which is set to a value of 0.1. The modality configuration for sending signals consists of two separate files, one for walking and one for sprinting. Their parameters are the same as with the *Sight* modality, just with different values. Both configurations have the same transmission speed but the maximum range and attenuation differ. The sprinting configuration has double the range and a slightly higher attenuation to simulate a clearer and louder sound.

6.1.3 Spatial Query

The SQS in this example is responsible for providing a list of possible hiding locations of the player after the agent has lost him in a chase. Figure 6.7 shows the configured parameters of the query, especially the generator. A *Cone Generator* is used to generate points in front

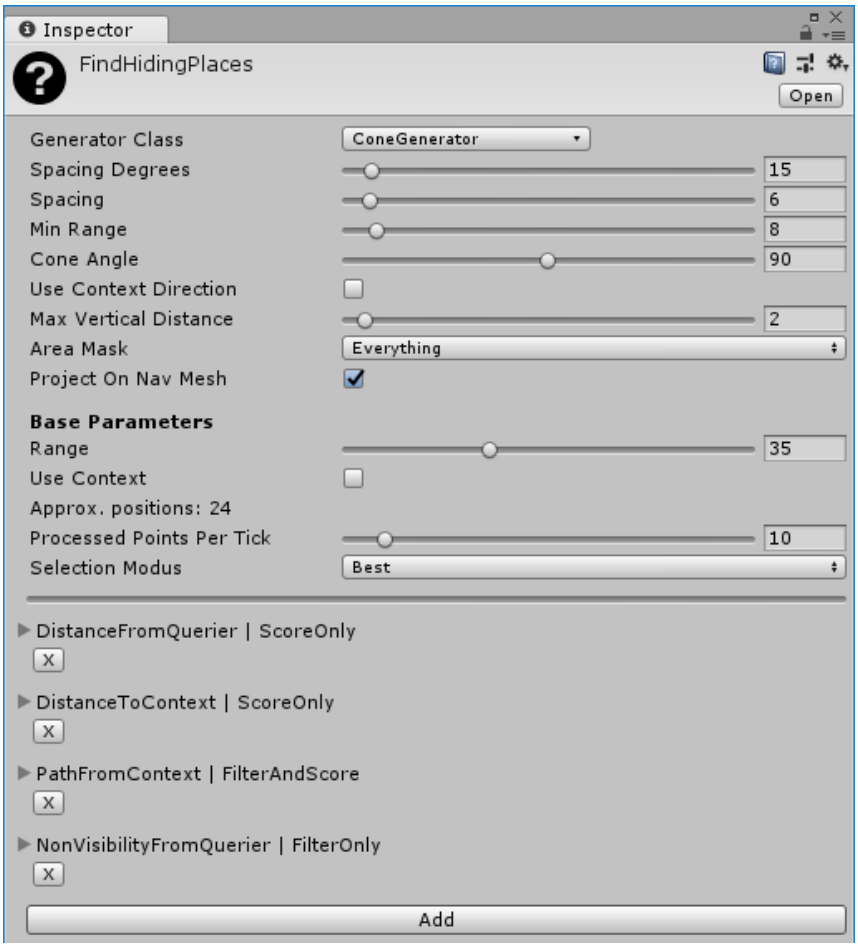

Figure 6.7: Example Configuration Spatial Query Configuration

of the agent. This is supposed to rule out any unlikely or impossible positions. Filtering and scoring of these positions is handled by four tests:

- *DistanceFromQuerier*: This is a *DistanceTest* which is supposed to make distant points more attractive for searching, with the assumption that the player is trying to move away from the guard. This is a score only test, no points are discarded here.
- *DistanceToContext*: Another *DistanceTest* but this test scores higher on points close to the context of the query, the player. This is supposed to start the guard near the position of the player. As above, this test only scores points.
- *PathFromContext*: This is a *PathTest* which filters out any points which are inaccessible for the player. It also scores the points based on their distance to the player, in this case, the closer the higher. This test filters and scores points.
- *NonVisibilityFromQuerier*: This is a *VisibilityTest* which simply filters out any points which are visible from the guard. Since the player is hiding from the guard, any visible points are not necessary for searching. This test only filters out points.

6.2 Example Scene

An example scene (see Figure 6.8 was built to test and evaluate the previously described behaviour in a "real" scenario. The scene is a simple rectangular box, filled with walls and obstacles. One of its design intents was to give the player several different types of obstacles for hiding and test the AI in different environments.

6.3 Walkthrough

To properly describe how this example works in practice, this section will provide a simple walk-through of it. This can be divided in three sections: *Patrol*, *Chase* and *Search*.

Patrol In the beginning both player and guard are placed in the level. While the player can move around freely, the guard immediately starts moving towards the nearest patrol point and continues to the next one, once it arrives. This will continue indefinitely until the guard senses the player, either through its hearing or sight sensor. The player will send constant sight signals and also sound signals, but only when moving. The sound signals also vary if the player is sprinting or not. Once the player is sensed, the guard will abort the patrol and start chasing the player.

Chase When chasing the player, the guard will simply always walk to the last position sensed by its sensors. While the player is in sensor range, this always be the players current position. The guard also increases its speed to make it harder for the player to escape. If the player manages to break line of sight and be far enough away that the guard can not hear him, the guard will abort chasing and start searching for the player.

Search If a player signal is picked up while searching, the guard will immediately start chasing again. Searching starts by moving to the last known position of the player. If no one is found, it will run a *SpatialQuery* to calculate possible hiding positions of the player. Once this is finished, the guard will move to best hiding position and try to look at all remaining hiding positions. If any of them are visible, the are removed from the

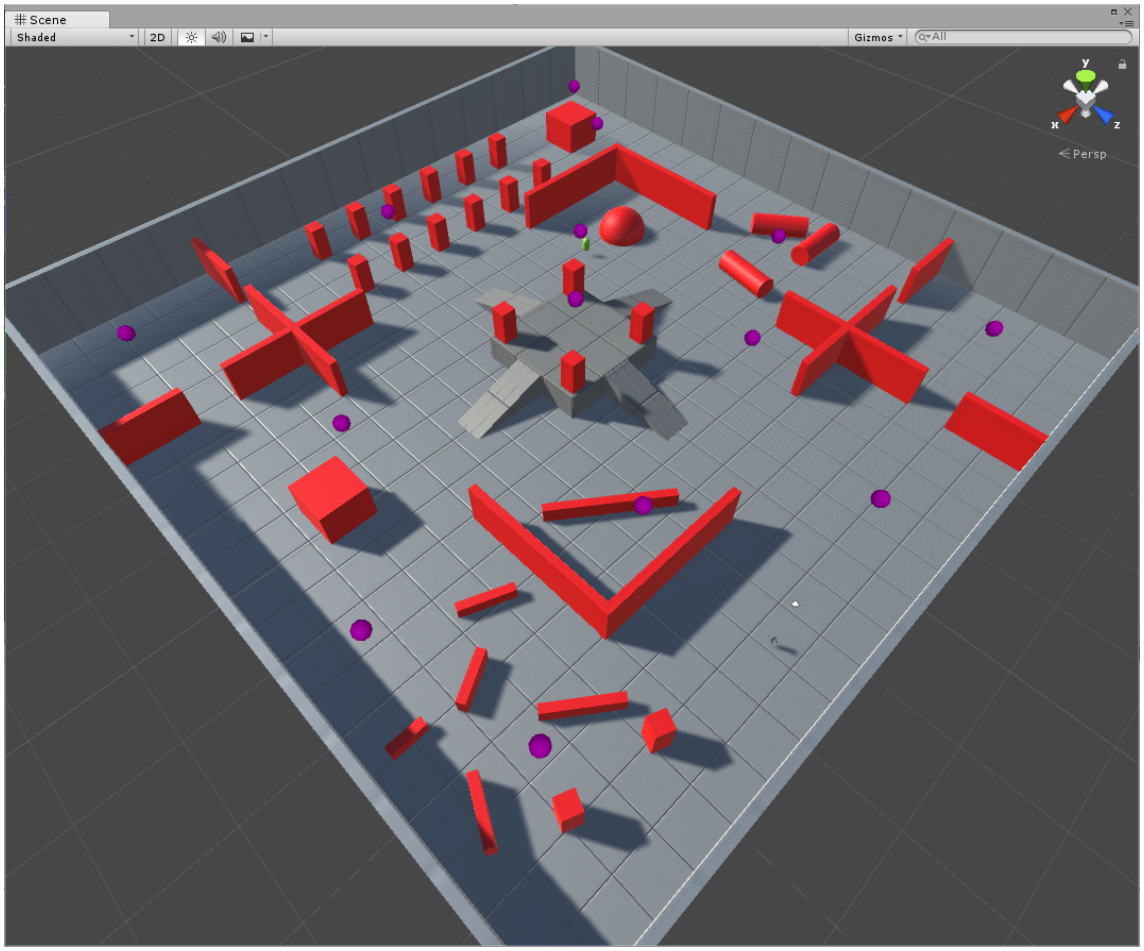

Figure 6.8: Example Scene Overview. Patrol points are indicated by magenta spheres.

list of remaining hiding positions, they are considered searched. If any remain after the first position, the process is repeated with the next position until none are left. Once this happens, the guard will start patrolling again.

6.4 Conclusion

While this example is a bit contrived and not a very sophisticated behaviour, the actual execution overall was a success. The amount of actual work on the implementation was quite low, most of the time was spent fixing bugs in the systems that appeared while creating each component. Apart from that, the behaviour tree took most of the time to create. A few new node types had to be created (*Choose Next Patrol Point*, *Choose Search Position* and *Check For Sensed Player*) but these were pretty straightforward to implement. The configuration of the used *SpatialQuery* was made very fast and easy through the debug display. Every change in parameters could be seen immediately in the world. Setting up the sensor system was the taking the least amount of time, since all there was to do, was adding the components to the guard *GameObject* and creating the few configuration assets.

In terms of the overall effectiveness and actual behaviour of the guard, it was a success for the most part. The behaviour tree worked just as intended, changing the guards behaviour at the right times and cancelling the previous actions. The *Sensory System*

worked great, visual and auditory signals were received without any problem and the service node picked them up as well. The only smaller problem occurred with the *SpatialQuery*. While the technical site worked fine without any problems, the generated points often did not represent probable hiding locations of the player, they often were in the wrong area. This lead to the search almost never being successful. No formal profiling was done to evaluate the actual computational cost but there were no spikes in resource usage and no slowdowns could be noticed in the guards behaviour.

Chapter 7

Conclusion And Future Outlook

The development of the UAIFramework to provide AI development tools for the Unity3D engine was successful, overall. The three techniques that were chosen provide a good baseline of tools and empower non-programmers to create new AI agent behaviours on their own. The most prominent part was the behaviour tree implementation plus its custom editor. This combination really makes it easy to create complex behaviours in a manageable and maintainable way. This, however, required a relative deep familiarization of how Unity3D's IMGUI system is used, which was the largest hurdle during development. As for the other components, the Sensory System was the easiest and simplest part of all three. The algorithm that was used, is very simple which has the advantage of making it easy to use. The disadvantage is that it lacks a more complex simulation, especially for the hearing sense. The SQS turned out to be a powerful, easy to use system. Generating useful data at arbitrary locations in the world can be important to create believable AI. This process can be expensive to run but is manageable when spread over a larger timespan.

For the future of the UAIFramework there are multiple options available. More AI techniques could be implemented, like Decision Trees, Neural Networks and a custom FSM implementation (custom because Unity3D already provides one, but it is not as user friendly as it could be) and many more. With a lot of independent features to pick from, the developers are empowered to create their desired AI designs. Improvements to the currently implemented techniques fall into two categories:

- *Feature Additions*: All three systems benefit from additional parts or features. The behaviour tree's usability is dependent on the available node types and is improved by their quantity and quality. The SQS is similarly dependent on the available generators and tests and its capability would increase with more options to choose from. In case of the *Sensory System* there is not much to add in types of useful humanoid senses but depending on the use case, some might be practical. One important sense would be the improvement of how sounds are simulated.

- *Performance Improvements*: With performance meaning computational cost, all three systems can be improved. While *Sensory System* is fairly lightweight, the *Behaviour Tree* could be improved through an alternate way of traversing the nodes, which does not traverse the every node but instead saves the currently active node and immediately executes it. The SQS's performance would improve greatly through a parallel execution of its parts, but since the Unity3D API is not thread-safe it would be difficult to implement. The *Unity Job System*[16] would be a possible system to use, but as of the date of this thesis, not all necessary API functions are accessible, mainly the NavMesh and physics API.

Bibliography

[1] *Unity3D*. https://unity3d.com/. [Online; accessed 23.05.2018].

[2] *Unity3D Feature Overview*. https://docs.unity3d.com/Manual/. [Online; accessed 26.06.2018].

[3] Neelam Soundarajan Jason Hallstrom. "Incremental Development Using Object Oriented Frameworks: A Case Study". In: *Journal Of Object Technology* (2002).

[4] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1st Edition. Prentice Hall, 1994. Chap. 5: Template Pattern. ISBN: 978-0201485370.

[5] *Unity3D MonoBehaviour class*. https://docs.unity3d.com/ScriptReference/MonoBehaviour.html. [Online; accessed 13.07.2018].

[6] *Unity3D Execution Order*. https://docs.unity3d.com/Manual/ExecutionOrder.html. [Online; accessed 13.07.2018].

[7] John Funge Ian Millington. *Artificial Intelligence for Games*. 2nd Edition. Taylor and Francis Ltd., 2009. Chap. 5.4. ISBN: 978-0123747310.

[8] Steve Rabin et al. *Game AI Pro*. 1st Edition. A K Peters/CRC Press, 2013. Chap. 6.3. ISBN: 978-1466565968.

[9] *Unity3D Editor Extension Manual*. https://docs.unity3d.com/Manual/ExtendingTheEditor.html. [Online; accessed 26.06.2018].

[10] John Funge Ian Millington. *Artificial Intelligence for Games*. 2nd Edition. Taylor and Francis Ltd., 2009. Chap. 10.5. ISBN: 978-0123747310.

[11] H. M. Traquair. *An Introduction To Clinical Perimetry*. 3rd Edition. London: Henry Kimpton, 1938. Chap. 1.

[12] Steve Rabin et al. *Game AI Pro*. 1st Edition. A K Peters/CRC Press, 2013. Chap. 33. ISBN: 978-1466565968.

[13] Steve Rabin et al. *Game AI Pro*. 1st Edition. A K Peters/CRC Press, 2013. Chap. 33.4. ISBN: 978-1466565968.

[14] Steve Rabin et al. *Game AI Pro*. 1st Edition. A K Peters/CRC Press, 2013. Chap. 33.5.1. ISBN: 978-1466565968.

[15] *Unity3D Thread Safety*. https://docs.unity3d.com/Manual/script-Serialization-Errors.html. [Online; accessed 13.07.2018].

[16] *Unity3D Job System*. https://docs.unity3d.com/Manual/JobSystem.html. [Online; accessed 20.07.2018].