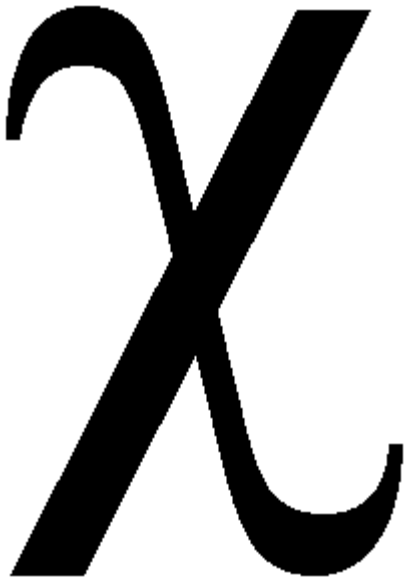


kAI - A Component Based AI Tool

Thomas Kiley - Discrete Mathematics

Project Supervisor: Professor Ranko Lazic

Second Assessor: Professor Nathan Griffiths

Key word list:
Artificial Intelligence, Visual Programming Tools,
Game Development, Behaviour Based AI

2013 - 2014

Table of Contents

Abstract

Develop a tool in C# which takes the idea of a component based objects (as opposed to inheritance hierarchies) that can be used to create AI behaviours for a wide variety of game AI's.

1. Introduction

1.1. The problem

Computer games are a pervasive medium that have eclipsed the film industry in terms of revenue[1]. Modern video games are a serious technical achievement, involving some of the most sophisticated rendering algorithms to achieve extremely high performance. Large game developers can employ hundreds of artists[2] to produce 100k polygon models.

In games, artificial intelligence (AI) techniques are used to control the behaviour of non-player entities. There is a huge range of scope for these AI: from path-finding algorithms to get the entities around the world, to highly sophisticated strategic AI that play opponents in strategy games, like chess.

With the rise of multi-processor devices – for example, the PlayStation 4, the latest games console from Sony, has 8 parallel processors[3] – AI agents in games are increasingly able to simulate complex behaviours[4].

In a game development studio, there are three main departments: art, design and code[5]. Art is responsible for producing the visual assets – 3D models, textures, interface images etc. The code team, consisting of programmers, is responsible for implementing the game. This ranges from low level engine work such as making the renderer, to more high level “gameplay code”, such as implementing the weapons system. The design team, which will often be made of non-technical staff, are responsible for producing gameplay content; this includes the different levels in the game and how the game's systems interact with each other.

The creation of the AI agents in the game is typically the responsibility of the code team[6] instead of the design team. This is because AIs often contain complex technical elements such as the A* path-finding algorithm (an efficient algorithm for working out the shortest viable path between two points[7]). The implementation of these elements would be outside the skill set of most designers.

However, this can create a serious problem for designers. AI's are increasingly central to the players experience[8]. As a result, when designing levels, designers will need to adjust the AI to get it to do precisely what is required. However, if the designer wishes to tweak the AI, they would have to find a coder to implement it for them. This can create a bottleneck in development and limits how much iteration the designers can perform on the AI.

1.2. The Solution

To try and address this problem, I have developed kAI.

kAI is a library and visual tool for developing AI's specifically for games. It is aimed at designers who do not have the technical experience to develop and modify game based AI's. The tool allows for the code team to provide the primitive operations that the AI's can perform, hiding complexities from the designer such as path-finding. The editor can be used to produce kAI behaviours – a network of interconnected nodes. Nodes themselves can be other kAI behaviours or nodes implemented by the coders at the development team.

I have also developed two example games to demonstrate what behaviours can be created within kAI. The first is a simple sword fighting game. The AI controls the behaviour of the opponent. It will move towards the player and try to hit the player when it gets near. If it is on low health, it will run away from the player.

The second example is considerably more complex. It is a strategy game in which the player controls squads of ships in a battle with another army. Each ship is running its own AI developed in kAI. This behaviour follows orders, but also has other competing motives to give rise to more subtle operation. For example, if it is low on health, it will choose to run away.

1.3. Current Technology

The current technology for developing AI in games is limited[9] and normally highly specialised towards a specific type of AI. For example, the React[10] Unity plugin allows only for the creation of behaviour trees. While these are quite versatile, there are certain AI design patterns, such as steering behaviours, which they are not suitable for.

More general tools, such as the visual programming tool included in the Unreal engine, Blueprint[11], come with the same problems that normal programming has – bugs, infinite loops and crashes. Also, as it is a general purpose tool, it is not streamlined to support central AI ideas such as behaviours.

These limitations have informed the design of kAI. The primitive operations – written by coders – are put in to kAI networks by designers. The form of these networks is very flexible, allowing for great freedom in what kinds of AI can be created. kAI allows, but is not limited to, the creation of behaviour trees, finite state machines and steering behaviours. Moreover, as it is more abstract than visual programming (for example, it lacks programming primitives such as loops), the tool is intuitive to designers and limits the introduction of complicated code bugs.

2. Current State of AI Development in Games

2.1. AI Archetypes

In games, artificial intelligence agents are used to drive entities in the game that are not controlled by the player. The complexity of these AIs varies hugely depending on the context of the game. Below are some archetypal examples of game AI, but this is by no means an exhaustive case study.

2.1.1. Boss Fights

Probably the simplest type of game AI is the one that controls bosses. Bosses are an enemy that are usually used to mark the end of a levels[12]. These enemies will have a set of pre-determined states. One of these will be a state where they are attacking the player. This is then typically followed by a state of vulnerability where the player is supposed to attack the AI.

This kind of behaviour would be implemented as a finite state machine[13], with each state representing one of the modes the boss can be in. Switching between these will then be controlled by some other mechanism such as a timer.

For this kind of AI to be developed, a designer would work out what different states they want the boss to have and how to transition between them. A coder will then develop this and return it to the designer. In most modern development studios, certain elements such as the timings, would be exposed to the designer in a tool. They can then tweak the values, but structural changes – such as the order of the states – will often involve the coder.

2.1.2. Steering Behaviours from Strike Suit Zero

In Strike Suit Zero[14] – the game I worked on at Born Ready Games – AI was written to fly all the non-player ships. This was done via a technique called steering behaviours[15]. These work by taking a weighted average of a set of vectors. The vectors come from individual steering behaviour such as seek (which moves the AI towards a target) or flee (which moves the AI away from a target). The AI can then be made to perform differently by adjusting the weightings at runtime. For example, one may make the flee behaviour carry a heavier weight when the AI is on low health.

Steering behaviours rely on the locomotion model to be parametrised by a single vector. This means that when the AI wants to move along a vector $\{1, 0.7, 0.2\}$, the next frame , the ship would move in that direction. However, this wasn't the case in Strike Suit Zero as we had a realistic inertia flight

model which required the ships to turn. As a result, the connection between the behaviours and the locomotion of the object was relatively complicated.

In addition to this lower level, moment-to-moment control, the ships had a more long term AI to control things like bombing runs as well as being responsible for the overall strategy to control all the ships in a cohesive manner.

All of these AI's were written by the coders and tested by the designers, who could then request changes. This meant that designers weren't able to have specific AI for scenarios in their levels.

2.1.3. AI from a strategy game such as Age of Empires

Probably the most sophisticated type of AI in games is the strategic AI. Games such as Go are the subject of much academic AI research, but even these represent only one section of the creation of AI for computer strategy games such as Age of Empires[16].

In addition to the non-trivial task of working out an optimal strategy in the grand strategy, these games often have many more interactions that must be handled in both a strategically sound and a believable way.

For complex AIs of this variety, a goal based system is often used[17]. In this, the AI has multiple goals which have their own set of tasks that must be completed to achieve it along with a priority rating for that goal.

This kind of AI will normally require dedicated personnel who will have both design and programming experience. It does, however, point to another challenge when creating game AI – debugging.

2.2. Debugging AI in games

Debugging AI in games is a serious problem. Due to the real time nature of games, traditional debugging methods cease to be suitable[18].

For example, consider print statements. If an AI is controlled by a finite state machine, one may use a print statement to print out the current state of the AI. However, since the game is running in real time, the number of print statements will quickly become unmanageable. This problem is only magnified if there are lots of different AIs in the game.

Breakpoints are also difficult to use in a real time setting. The game will often become unplayable if the game has to pause at every frame. Often, bugs in AI will be caused by a rare combination of stimuli. These will become hard to reproduce if a breakpoint is hit every frame.

2.3. Current Tools for AI in Games

There are some existing tools for developing AIs in games.

2.3.1. Blueprint (formerly Kismet)

The Unreal Engine is a high profile game engine, most notably used for Gears of War[19] but is common in many large games. Included within the Unreal editor, there is a tool called Blueprint[11]. Blueprint is a node based editor that allows designers to create routines and behaviours in a visual way. It can be used for AI but is also used for scripting the flow of levels (e.g. when do enemies appear, making automatic doors open etc.).

As Blueprint is a visual programming tool, rather than a specialist AI tool, it includes the ability to create loops and branching statements.[20] These have the potential to cause the game to hang. Since designers do not necessarily have the technical experience to debug a hang, this can involve the code team.

2.3.2. React

React is a plug-in for developing behaviour trees in Unity[21]. Behaviour trees are a powerful AI tool and a key paradigm in modern AI development[22].

However, the key limitation of this tool is it generates code for simulating behaviour trees. Having generated this code, you can no longer modify the trees directly, making it difficult for designers to adjust the trees based on testing.

Further, while behaviour trees have a wide variety of applications, things such as steering behaviours and other, more goal oriented behaviours are not supported.

2.4. Questionnaire

I asked game developers from both the code and design disciplines to answer a series of questions about current AI tools and what tools they would like to use. I sent the questionnaire to people I know

from my time at Born Ready Games and on Twitter. I received 13 responses in total. The full questionnaire and responses are provided in appendix I on the CD.

One question was regarding existing tools for AI development. 92% of respondents who had worked with AI said there were no available AI tools when they were working on games.

One respondent provided an example of the kind of existing tools they were using:

> "On one project we had a tool that allowed us to to modify a behaviour tree. We were able to add, remove and edit the nodes on the tree.
> Another was a visual scripting system with the AI commands exposed by scriptable nodes. (Similar to Kismet)"

I also asked respondents what kind of AI tools they thought would be most useful. There were a number of comments but common themes were:

- A visual editor for designers
  - "If there were a framework for writing AI behaviours/states for any game (more or less), but that also supports a visual editor of some sort for designers to stitch them all together then that could be useful."
- The ability to debug behaviours
  - "An ability to 'debug' these behaviours (seeing live updates of what was happening while the game was running) would also be a great help in the tool."
- Flexible as the nature of AI's varies hugely from game to game
  - "It really depends on the type of AI being developed."

This, coupled with the lack of tools available, clearly demonstrates a need for AI tools. I then asked the respondents to rate (out of 5) how potentially useful the tool I planned to create appeared.

> Rate how useful the following tool sounds:
> Coders develop code behaviours that represent primitive operations for AIs. Designers then combine in a visual tool, these behaviours in to a network that controls how the AI behaves on a high level

Most people (92%) rated the tool I designed as either Useful or Very Useful.

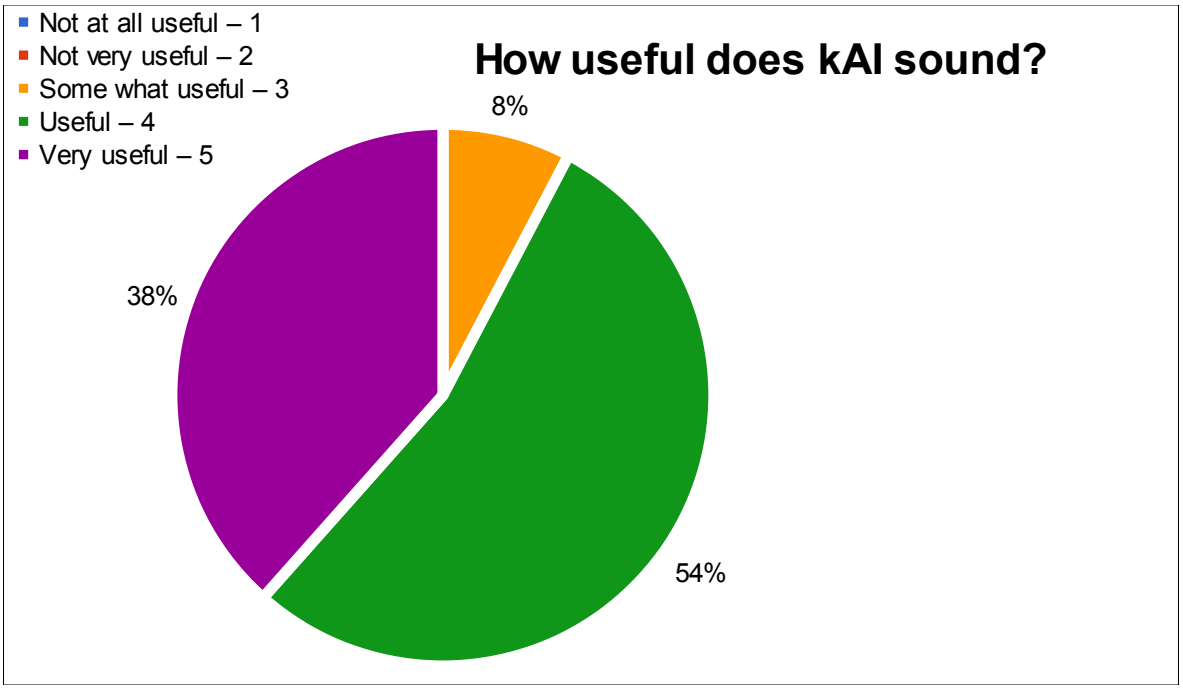


Fig 2.1Respondents to how useful would the proposed tool would be

2.4.1. Conclusion

The survey clearly demonstrates before a desire for and a type of tool for AI game development. While some tools do exist, they are normally highly limited or specialised. Further, the positive reaction to the notation of the tool encouraged me to peruse the development of kAI.

3. Project Development

3.1. Overview

My project consisted of 4 main elements, the engine, the editor, the debugger and some examples.

- kAI Core is a library that stores and simulates the kAI behaviours
- kAI Editor is a visual tool for creating kAI Behaviours
- kAI Debugger is the debugging interface for seeing what the behaviours do in real time
- The first example game is a simple fighting game which demonstrates kAI in a simple manner
- The second example is a strategy game which features different kAI behaviours simulating more sophisticated AIs

3.1.1. Terminology

Below are the terms for things within kAI.

3.1.1.1 kAI Behaviours

A kAI Behaviour is the central object of kAI. They are a network of nodes connected together via ports. Game designers can create kAI behaviours in the editor.

Global kAI Behaviour
A kAI behaviour that is used to simulate the AI of an entity. It is at the root of the tree of behaviours.

Embedded kAI Behaviour
A kAI behaviour that is used as a node in another kAI behaviour.

3.1.1.2 Nodes

Nodes are elements of the network within a kAI Behaviour. They represent an instance of some class that inherits from kAIINodeObject. The primary inheritors of this class are as follows:

Code Behaviours
These are behaviours written by the coders at the development team. They represent the primitive actions of the AI.

kAI Behaviours
Behaviours created by the tool can be embedded as nodes in other kAI behaviours. Code Behaviours and kAI Behaviours are collectively referred to as behaviours.

Functions
Coders can expose functions to designers to provide utilities such as mathematical operations to the designer.

Constants
Configuring parameters like numbers can be embedded into the network in the form of constant nodes which just store a particular value.

3.1.1.3 Ports

A port is a gateway for two nodes to be connected via. They come in two types:

Trigger Ports
These correspond to events occurring and can be triggered on a certain frame. They can be used to activate and deactivate behaviours.

Data Ports
These correspond to a specific data type and transmit a continuous stream of data of the specified type. These can be used to monitor the value of something within the game.

Connexions
Ports are connected together to form the kAI network. The connexions must be between matching types and differing directions. i.e. trigger ports must be connected to trigger ports, data ports must connect to a data port of the same type and outbound ports (ports that send data and trigger) are connected to inbound ports (receive data and get triggered).

kAI behaviours can define what ports they have. This involves the creation of two ports:

Internal Ports
These are on the inside of the kAI behaviour and can be connected to/from nodes that are inside the kAI behaviour.

External Ports
These are of the opposite direction to the internal port. If the kAI behaviour is embedded as a node in another kAI behaviour, then these are the ports the node has that can be connected to/from. If it is a global kAI behaviour, the game communicates with the kAI behaviours through these ports.

3.2. kAI Core

The core of kAI is a library which stores and simulates the behaviours. It was developed in C#[23]. The deliverable is a single DLL[24] that is used by both the editor and the games to create and run the behaviours.

3.2.1. Key Classes

Class	Responsibility
`kAIBehaviour`	Represents a behaviour that can be either active or inactive.
`kAIXmlBehaviour`	Represents a kAI behaviour created by the editor, manages what nodes are within it
`kAICodeBehaviour`	Represents a code behaviour which can be developed by coders inheriting from this class.
`kAINode`	Represents a node within a kAI behaviour; contains a `NodeObject`.
`kAINodeObject`	Represents something that can go within a node; inherited by `kAIBehaviour`, `kAIFunctionNode`, `kAIConstantNode`. Details how it can be serialised and what ports can be connected to from other ports.
`kAIPort`	Represents a port on a node that can be connected to or from.
`kAITriggerPort`	Represents a trigger port.
`kAIDataPort<T>`	Represents a data port of type T.
`kAIFunctionNode`	Represents a function that can be used in a kAI Behaviour.

3.2.2. Key Technical Areas

3.2.2.1 Behaviour Update

To use kAI, one creates a kAI behaviour in the editor and then attaches this to an entity in the game. Then, once a frame, the update method is called on this kAI behaviour – simulating the AI. This section will detail how the update method was developed to avoid issues relating to non-determinism.

When designing the update system, I wanted some elements of it to be deterministic (i.e. running the behaviour twice on the same input would produce a consistent output). This is important for debugging as inconsistent bugs are very hard to fix[25].

What determinism means in the context of a kAI behaviour must be carefully considered and limited. Since a kAI behaviour is a network of nodes, of which more than one can be active, there is no intuitive order in which to evaluate the update of each of the nodes. Further, guaranteeing an order of these nodes is not desirable as it would restrict how much parallelism can be achieved when simulating the behaviour.

Instead, the determinism that kAI guarantees is more restrictive:

Definition: Given a kAI behaviour with some behaviour nodes within it. Before updating any behaviour, the behaviours that are active is deterministic. That is, whatever order the behaviours and ports are updated, the set of behaviours that are updated in the following frame is deterministic. This is said to be a deterministic kAI behaviour.

To achieve this, I had to work through a couple of different update algorithms. For each algorithm there is a trigger method and an update method. The trigger method is called whenever a trigger port is triggered. The update method is called once per frame on each behaviour.

Naïve

This algorithm is the simplest way to implement these two functions. When a trigger port is triggered, the action – such as activating a node – is performed immediately. Then, all connected ports are triggered.

The behaviour update iterates through all the nodes in the kAI behaviour. If they are active it calls their update function.

```
Trigger(p : trigger port)
        Perform action associated with p
        ∀ p' where (p , p') ∈ {connexions}
                Trigger(p')
        end loop
end method
```

```
Update(B : χ − behaviour)
        if B is active
                ∀ B' : behaviour ∈ B
                        if B' : code behaviour ∧ active
                                Code Update(B')
                        else if B' : χ − behaviour
                                Update(B')
                        end if
                end loop
        end if
end method
```

Algorithm 3.1: Naïve algorithm for triggering a port and updating a kAI behaviour

However, this does not preserve determinism, consider the following counter example.

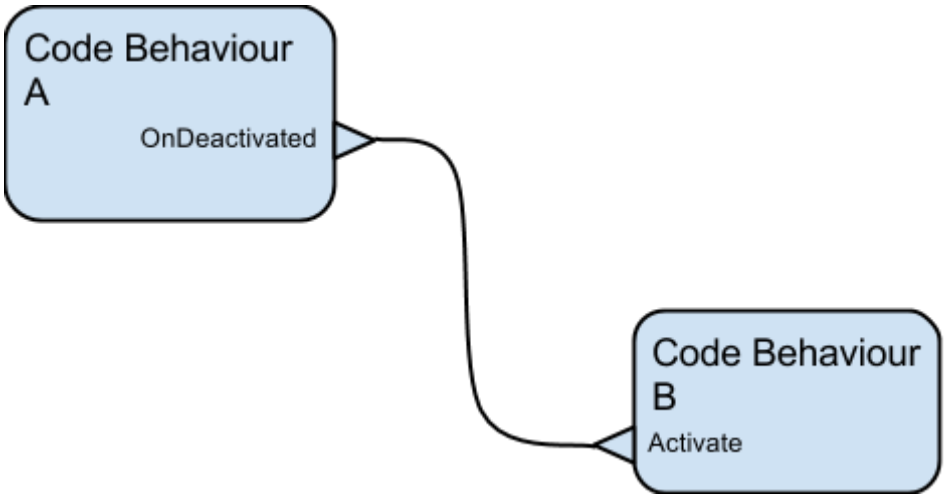


Fig 3.1: Sample Behaviour exhibiting problems with the updates

In this sample, the first code behaviours OnDeactivated trigger port is connected to the second code behaviours Activate port. Let the update of A causes it to be deactivated, causing the OnDeactivated port to be triggered.

Suppose A is updated first in the update of this kAI behaviour. This triggers the OnDeactivated Port. According to the algorithm, this immediately notifies the Activate port of B. Again, according the trigger

algorithm, this will immediately activate B. Now, when the update behaviour checks to see if B is active, it will find it is and update it.

However, were the update to first attempt to update B, it would find it inactive and leave it. It would then move on to A, causing the Activate port to be triggered. However, it wouldn't be until next frame that B would be updated.

This can be seen in the following two diagrams, in the first, the update of B is performed, in the second, it is not.

Node A updated first

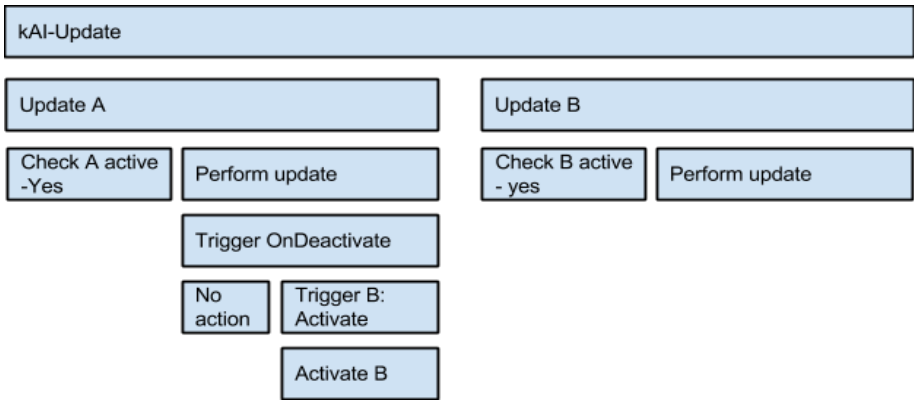


Fig 3.2: Update breakdown under the naïve algorithm

Node B updated first

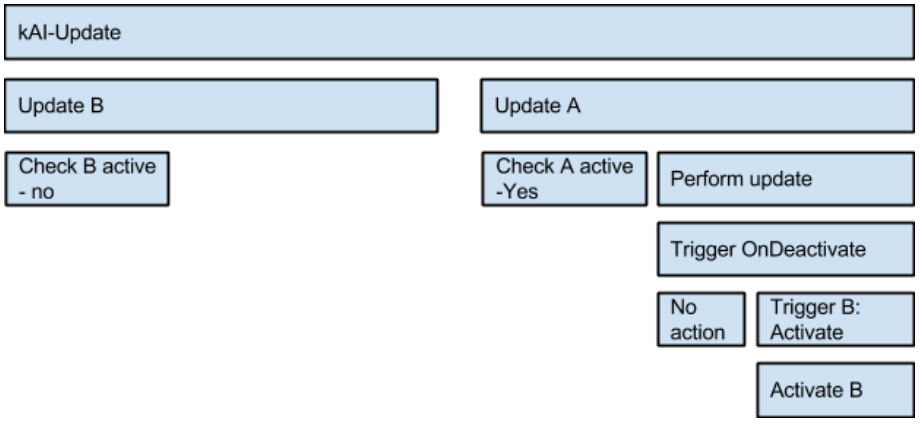


Fig 3.3: Update breakdown of the naïve algorithm with the order reversed

This violates the determinism required as it is not deterministic which frame B will first be updated in.

Hold and Release (H&R)

This algorithm attempts to resolve the issues of the naïve implementation by delaying the actions of the port until the beginning of the subsequent frame. In this algorithm, when a trigger port is triggered, nothing happens until all nodes have been updated. At the beginning of each frame, the ports are iterated through and released, causing their action to be performed.

```
Trigger( p : trigger port )
        Mark p has triggered
end method
```

```
Release( p : trigger port )
        if p marked as triggered
                Perform action associated with p
                ∀ p' where ( p , p' ) ∈ { connexions }
                        Trigger( p' )
                end loop
        end if
end method
```

```
Update( B : χ − behaviour )
        if B is active
                ∀ p : trigger port ∈ { node ports } ∪ { internal ports }
                        Release( p )
                end loop

                ∀ B' : behaviour ∈ B
                        if B' : code behaviour ∧ active
                                Code Update( B' )
                        else if B' : χ − behaviour
                                Update( B' )
                        end if
                end loop
        end if
end method
```

Algorithm 3.2: Hold and Release algorithm for triggering a port and updating a kAI behaviour. This uses an auxiliary algorithm called Release for releasing a port

This resolves the issue of the naive implementation: in whichever order the nodes are updated, the activate node won't be triggered until the beginning of the next frame.

However, there is no clear order in which to release these port. Using the same counter example and considering the release phase, the problem with the H&R algorithm can be seen.

Suppose the OnDeactivated port is released first. Then, by the algorithm, this tells the Activate port it has been triggered (marking it for release). Now the Activate port is released, which has been marked as triggered, so B will be activated.

However, if the Activate port was released first, it will not have been marked and hence does not activate B. The OnDeactivated port is the released, marking the Activate port, but this will not be released until next frame. This can be seen in the following two diagrams; in the first, B is activated, in the second it is not.

OnDeactivated Released First

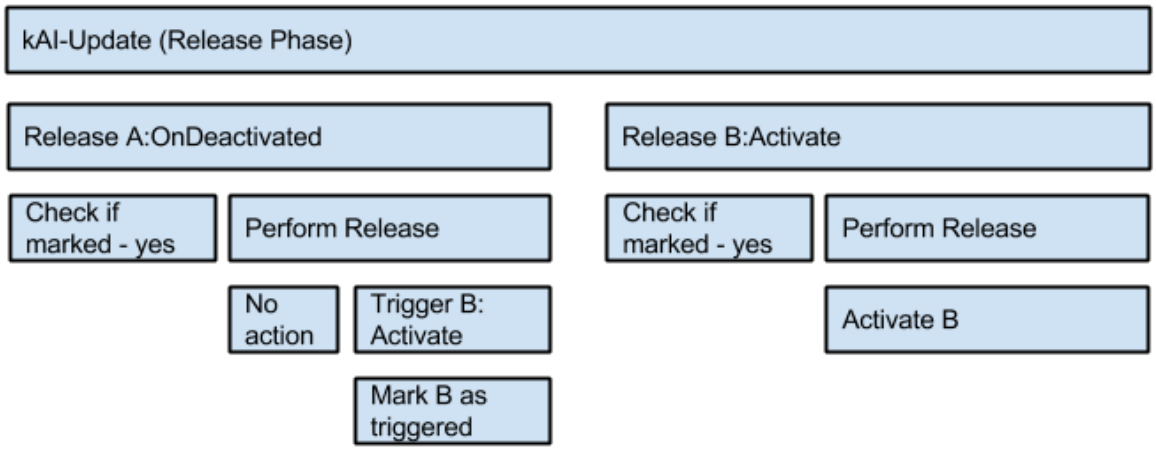


Fig 3.4: Update breakdown of the H&R algorithm

Activate Released First

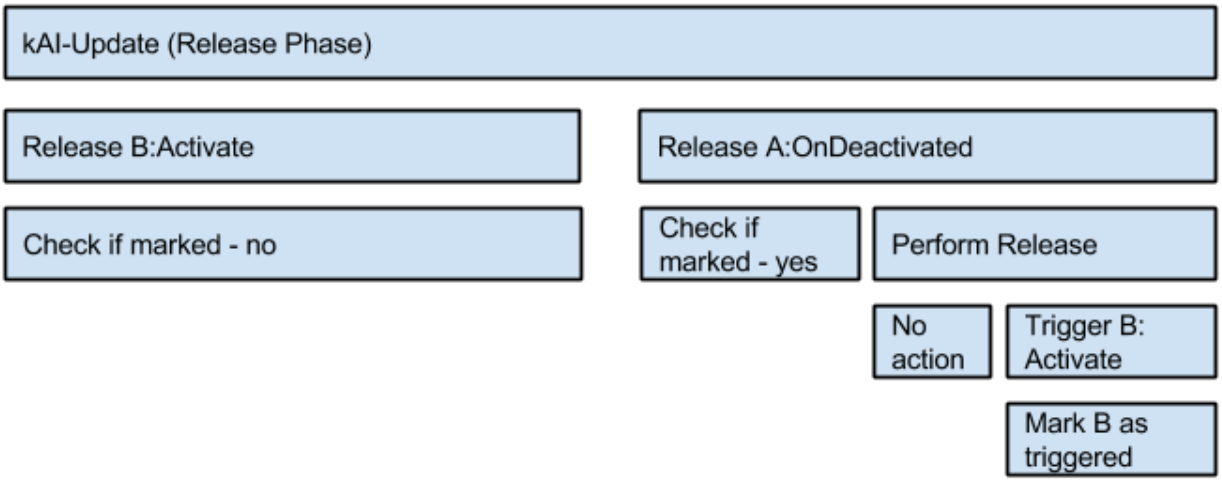


Fig 3.5: Update breakdown of the H&R algorithm with the order reversed

This violates determinism because it is not deterministic if B will be active after the release phase.

Drill Down

Like H&R, the action associated with the trigger port is not taken until the beginning of the next frame. This resolves the issue created in the first solution (Naïve). The difference between this and H&R algorithm is that all connected ports are informed immediately that they have been triggered. These still don't perform any action. However, this modification ensures that the same nodes are active in the same frame.

This algorithm also addresses a more fundamental issue with trigger ports. When a code behaviour has a trigger port, the coders will write event handlers. These methods will describe what actions should happen when the port is triggered. These event handlers can be considered as the actions of the port and hence are performed during the release phase of the update.

However if these event handlers trigger ports themselves, the same non-determinism that could be seen in the H&R algorithm is introduced. The port that is triggered in the event handler may or may not have yet been released (as the order in which the ports are released is non-deterministic). As a result, it is non-deterministic if its action (such as activating a node) will be performed this frame or the next. The Drill Down update resolves this - an error is thrown if a trigger port's event handler attempts to trigger a port.

```
Trigger( p : trigger port )
        if release phase
                throw new error : cannot trigger port during release phase
        end if
        ∀ p' where ( p , p' ) ∈ { connexions }
                Trigger( p' )
        end loop
end method
```

```
Release( p : trigger port )
        if p marked as triggered
                Perform action associated with p
        end if
end method
```

```
Update( B : χ − behaviour )
        if B is active
                release phase = true
                        ∀ p : trigger port ∈ { node ports } ∪ { internal ports }
                                Release( p )
                        end loop
                release phase = false
                ∀ B' : behaviour ∈ B
                        if B' : code behaviour ∧ active
                                Code Update( B' )
                        else if B' : χ − behaviour
                                Update( B' )
                        end if
                end loop
        end if
end method
```

Algorithm 3.3: Drill down algorithms for triggering a port and updating a kAI behaviour

This resolves all issues of non-determinism with the update, ensuring that whatever order the nodes and ports are considered, the same set of behaviours will be active in each frame.

Theorem: The Drill Down Update produces deterministic behaviours.

Proof:

1. The areas of non-determinism in the update come from the order of the nodes being updated and the order of the ports being released, the rest of the update is deterministic
2. First consider the node order:
    - 2.1. At the start of a given frame, suppose there is an active node and an inactive node.
    - 2.2. Since the nodes can only change active-state through one of their trigger ports being triggered and all of them are released at the start of the frame their state cannot change this frame.
    - 2.3. Suppose during this frame, the active node triggers a port connected to the activate port.
        - 2.3.1. The activate port will be marked this frame since the drill down algorithm ensures all connected ports are marked as having been triggered
        - 2.3.2. The node won't become active until the subsequent frame
        - 2.3.3. On the subsequent frame, since the activate port is marked, the other node will always be activated in time for the update
    - 2.4. Suppose at some point during that frame the node deactivates itself
        - 2.4.1. Since the node was already being updated, it will always have had its update called that frame
        - 2.4.2. Since deactivating triggers the OnDeactivated port, any effects of this are covered by 2.3
    - 2.5. Suppose some trigger port is connected to either a activate or deactivate port of a node is triggered before the update
        - 2.5.1. Since is before the kAI update the effect will happen always in the first kAI Update following it as the action will be enacted by the release phase
    - 2.6. Suppose a deactivate port is triggered during the update
        - 2.6.1. As with the activate, it will be marked so nothing will happen this frame
        - 2.6.2. Therefore, irrespective of whether the node has been updated it will be updated this frame and deactivated before next frame
3. Consider the order of release of port orders

3.1. The actions of ports are to activate and deactivate nodes and code behaviours
    3.1.1. All ports are released before considering the nodes, therefore the activate and deactivate actions will all be taken before they affect the behaviour
    3.1.2. If a trigger port is connected to some code, this can break determinism - if another trigger port is triggered it is not known if this will have been released already been released
    3.1.3. However, this throws a wrong phase error, preventing other triggers being triggered
    3.1.4. Therefore any other externally visible action will be delayed until the node is updated
3.2. Since all connected ports are notified in the drill down, the order doesn't affect this

4. These operations represent all the ways nodes can influence other nodes (for example, a code behaviour cannot access another code behaviour except via ports)
5. Therefore all non-determinism in the update cannot affect which nodes are active in which frame

QED

Conclusion

This update gives a very useful property to kAI: it is impossible to structure a kAI network to enter an infinite loop that would hang the game. This is because even if two behaviours constantly activate each other, a frame will have to pass for this to be released. This means that even if the AI doesn't do anything useful, the game's update and render passes can still be done. Other comparable tools, such as Kismet, do not do ensure hanging loops cannot be created. This is a unique feature that makes kAI a robust tools for designers to use.

3.2.2.2 Ports

Ports are how nodes communicate with each other and how behaviours (both kAI and code) are activated and deactivated. They come in two flavours: trigger and data.

Trigger ports are the type discussed in the Update section (3.2.2.1) and are used primarily to activate and deactivate nodes. Their interface is fairly simple, they have a function to trigger them and an event which is called when their action is to be taken (in the release phase).

Data Ports have an associated type. They are used to transmit data from the game to code behaviours. They are also used to pass parameters of function nodes to the functions.

```
1. public class kAIDataPort<T> : kAIPort
```

Code Sample 3.1: Data Port class definition

It has a public property, `Data` , which allows the getting and setting of the data that is stored in this data port and hence all ports it is connected to.

One challenge with implementing the ports is, when a port is added to a kAI behaviour, two ports need to be created: an outbound one on the inside of the behaviour for designers to connect from, and an inbound one on the outside of the behaviour. The external port is the one that coders interface with or, if the kAI behaviour is embedded into another kAI behaviour, designers of the parent kAI behaviour interface with.

With trigger ports, an event handler is added to the external trigger port to listen for it being triggered. This event handler has a reference to the internal trigger port and triggers this.

For data ports, because it is not known at compile time what the type of the port are, the two ports must be bound together. This means when the data is updated in one of the port, it knows what it is bound to and is able to update the value in the other port.

3.2.2.3 Function Nodes

Function nodes use reflection (a process of inspecting compiled code at runtime) to represent functions as nodes. A function in C# can be represented via the `MethodInfo` [26] type which contains all the relevant information about the method such as its signature and the ability to invoke it.

This information can then be used to construct a node. For each of the parameters, a data port of the same type is created. Further, an out-going data port of the return type is created. The node can then store a reference to the function which can be invoked - passing in the parameters from the data ports and setting the value of the outbound data port based on what is returned – when it is updated.

There were some interesting challenges in dealing with more complex functions.

Generic Functions

The first is dealing with generic methods[27]. For example, kAI includes some primitive functions for developers to use such as `IfEquals`:

```
1. public static bool IfEquals<T>(T entry1, T entry2)
```

Code Sample 3.2: Example function node function deceleration

Which is a simple function to compare whether two data ports contain the same value. However, the user will specify, on a node by node basis, what type T will actually represent.

This is resolved using an auxiliary class, `kAIFunctionConfiguration`, which is passed in when creating the function node for a specific method. This has some properties which are used for configuring the function.

```
1. Type[] mGenericTypes;
2. List<int>[] mGenericMappings;
```

Code Sample 3.3: Function Configuration properties for configuring generic parameters

The `mGenericTypes` array stores a concrete type for each of the generic parameters (so for the `IfEquals` method, this would be an array of length 1 which contains the concrete type the user has selected T to be). The `mGenericMappings` has a list of indices for each generic parameter.

```
1. mGenericTypes = { actual type of T }
2. mGenericMappings = { {0, 1} } //i.e. the type of T corresponds to the 1st and 2nd parameter
```

Code Sample 3.4: Example configuration for the first generic function

For a more sophisticated example, such as:

```
1. public static U AOrBTransformer<T, U>(bool value, T trueValue, T falseValue)
```

Code Sample 3.5: More complex example generic function

Which is a contrived example which takes two data ports of type T and based on a boolean value, transforms them in to a type U. The result of this would be:

```
1. mGenericTypes = { actual type of T, actual type of U }
2. mGenericMappings = { {1, 2}, {-1} }
```

Code Sample 3.6: Example configuration for the second generic function

Where here -1 indicates the generic parameter relates to the return parameter.

This system has a limitation: it does not support functions whose parameters or returns have a type that have a nested generic type, for example `IEnumerable<T>` since the generic mapping does not know

where the concrete type goes within the parameter. While it would be possible to resolve this for types that have only a single generic parameter, if the type a pair of generic parameters, then it is not clear in this structure which one you are trying to specify.

An alternative implementation
To resolve this issue, an alternative implementation would be to store a map from generic parameter names to concrete types. Then, when actually filling in the types, one could check what the name of the type is (e.g. "T") and look up the corresponding type (e.g. float). This would resolve nested types as you could just recursively perform this on all parameters that aren't fully defined.

Function Node Generic Parameter Constraints

In C#, generic parameters can be constrained via an interface[28] (i.e. the type must implement a specified interface). However, this has a limitation – it cannot require that a type implements operators such as addition (+) or subtraction (-)[29]. Since I planned to use the functions as mathematical operations, it was important that coders were able to specify what operators they were going to use in the function. For this, I created a static constraint system which allowed methods to specify that a type must implement certain operators.

To do this, the target function would be decorated with an attribute[30] I created called a `StaticConstraint`. Here is the minus function decorated with the requirement that T must implement the subtract operator.

```
1. [StaticConstraint(StaticConstraint.StaticConstraintType.eConstraint_Minus)]
2. public static T Subtract<T>(T entry1, T entry2)
3. {
4.     return Operator.Subtract<T>(entry1, entry2);
5. }
```

Code Sample 3.7: Example function with a static constraint

The `Operator` class is a part of an open source library – MiscUtil[31] – that handles calling operators on unknown types. Unlike C#'s interface constraint system, these constraints do not work at compile time. However, the purpose was to allow the editor to know which types the function supports. This means, when designers are creating function nodes based off this function, they will see a list of types that are supported by the function.

This decorator provides a function for checking whether a given type is suitable for this function. This is done through reflection and knowing the names of the functions. For each constraint type (listed in `StaticConstraint.StaticConstraintType`) there is a function which takes a Type and returns a

description of the method the type needs to have. For example, here is the function for the addition operator:

```
1. public static MethodDescription[] PlusOperator(Type Type)
2. {
3.     MethodDescription PlusOp = new MethodDescription();
4.     PlusOp.mMethodName = "op_Addition";
5.     PlusOp.mParameters = new List<Type>(new Type[] { Type, Type });
6.     PlusOp.mReturnType = Type;
7.
8.     PlusOp.mSpecialCases = new List<Type>();
9.     PlusOp.mSpecialCases.AddRange(GetPrimitiveTypes());
10.    return new[] { PlusOp };
11.}
```

Code Sample 3.8: Code for the plus operator constraint check

Here it is specified that the type must have a static method called “op_Addition” (the special name for the operator method) which takes two parameters of the relevant type and returns a variable of the same type.

Further, primitives in C# - such as integers - do not work in the same way and have their operators built into the compiler. To resolve this, I hard coded these types as supporting specific operators. These are added to the special cases list for the method description.

Using this description, kAI is able to check if a type is a suitable parameter for this function.

```
1. public bool DoesTypeHaveMethodMatch(Type lType)
2. {
3.     if (mSpecialCases.Contains(lType))
4.     {
5.         return true;
6.     }
7.
8.     MethodInfo lMethod = lType.GetMethod(mMethodName);
9.     if (lMethod == null)
10.    {
11.        return false;
12.    }
13.    if (!lMethod.ReturnType.Equals(mReturnType))
14.    {
15.        return false;
16.    }
17.
18.    ParameterInfo[] lParams = lMethod.GetParameters();
19.
20.    if (!lParams.DoMatch(mParameters))
21.    {
22.        return false;
23.    }
24.    return true;
25.}
```

Code Sample 3.9: Code for checking if a specific type is suitable for a static constraint

First, the type is checked against one of the special case types (e.g. is it an integer?) on line 3. If not, the type must have a method that matches the name – checked on lines 8 and 9; the return type – check on line 13; and the parameters – checked on line 20 – of the method description. Assuming it passes all these tests then the type provided matches the MethodDescription and it is a suitable candidate for this method.

Smart Return Types

The other main issue when creating function nodes is dealing with the return type. I wanted functions that return a boolean value to be able to trigger a port as opposed to just having a data port of type boolean.

However, I wanted to implement this in a generic manner to allow for more complex configuration of specific return types. This could be used by developers who wanted to provide bespoke actions for their own return types. For example, the developer could define a trigger port for the return type vector which is triggered when the vector is of unit length. Then for all vector returning function nodes, there will be the option of enabling this trigger port.

This is again part of the `kAIFunctionConfiguration`, specifically the `kAIReturnConfiguration`.

To provide custom functionality for a type, the developer must provide a `kAIReturnConfigurationDictionary<T>` corresponding to the type for which the developer wants to provide custom functionality. For example, there is built in functionality for boolean returns where it is specified that the function node can optionally have a port that is triggered when the function evaluates to true.

A property for a return type has 4 components:

- The name of the property
- Whether it is enabled by default
- What port does it need
- What action needs to be taken when the function is evaluated

The action knows the current and previous values of the function and can interact with the ports of the node. If the developer provides these dictionaries for a specific type, then the properties will be loaded in to the editor and displayed as check boxes for the designer when creating the function node.

Conclusion
Function nodes combined with code behaviours give a huge amount of flexibility to developers. The development team can choose to use kAI in a way that best suited to the AIs in their game.

3.2.2.4 Save Format

kAI behaviours are saved into an XML format. To save a kAI behaviour, all nodes and the connexions between them need to be saved.

The connexions are stored as a list of connected ports. I chose this representation as opposed to a connexion matrix as the graph is typically sparse.[32]

To save the nodes, the save format needs to be independent of the type of node, so that it can be loaded dynamically. Each `kAINodeObject` (the base class for all items that can be used as a node) implements two methods:

```
1. abstract kAIINodeSerialObject GetDataContractClass(kAIXmlBehaviour lOwningBehaviour);
2. abstract Type GetDataContractType();
```

Code Sample 3.10: Interface methods for node objects

The first method returns an object that can be serialised into XML using C#'s built-in XML serialiser[33]. The second returns the type that will be returned from this class. These are both stored when storing

the node. For each node in the kAI behaviour, an XML node is generated. For example, a code behaviour of type SwordSwingAction would generate the following XML:

```
1. <kAIXmlBehaviour.InternalXml.SerialNode>
2.   <NodeContents i:type="kAICodeBehaviour.SerialObject">
3.     <BehaviourAssembly>Assembly-CSharp</BehaviourAssembly>
4.     <BehaviourID>SwordSwingAction</BehaviourID>
5.     <BehaviourType>SwordSwingAction</BehaviourType>
6.   </NodeContents>
7.   <NodeID>
8.     <NodeID>SwordSwingAction</NodeID>
9.   </NodeID>
10.  <NodeSerialAssembly>kAICore</NodeSerialAssembly>
11.  <NodeSerialType>kAI.Core.kAICodeBehaviour+SerialObject</NodeSerialType>
12.  <NodeType>SwordSwingAction</NodeType>
13.  <NodeTypeAssembly>Assembly-CSharp</NodeTypeAssembly>
14.</kAIXmlBehaviour.InternalXml.SerialNode>
```

Code Sample 3.11: Example node XML tag from a kAI Behaviour

On line 13, the type – `kAICodeBehaviour.SerialObject` – is specified, allowing the node contents tag (lines 4 – 8) to be de-serialised using this class.

Another challenge in saving was creating the serialisation class for kAI behaviours nested within kAI behaviours. An obvious answer would be to serialise nested kAI behaviours in the same way – that is – recursively serialise the child kAI behaviours as part of the parent XML. However, this has two main problems:

- **File Size** - If a kAI behaviour contains 5 kAI behaviours which in turn uses another 5 kAI behaviours, the file could become very large even if it is the same kAI behaviour being used 5 times
- **File Rigidity** - If one of the behaviours is changed, the editor would have to update all the behaviours that use it. This would take time when saving and would cause difficulties with source control.

Instead, the Serialisation Object for a kAI behaviour is just a path to the behaviour. Then, when the behaviour is de-serialised, the kAI behaviour locates the XML file referenced and de-serialises that in turn. This decouples the XML behaviours allowing for them to be modified independently without the likelihood of creating issues like merge conflicts.

However, this comes with another problem: relative paths. Since games need to be path independent (developers do not know where the user will install the game), the behaviours must use a non-absolute path. To do this, the paths are stored relative to the root behaviour they belong to. This means the game

doesn't need any knowledge of the project and behaviours can easily be reused in other games as they are self contained.

```
1. <NodeContents i:type="kAIXmlBehaviour.SerialObject">
2.   <BehaviourID>
3.     <BehaviourID>AIIdleBehaviour</BehaviourID>
4.   </BehaviourID>
5.   <XmlBehaviourFile>
6.     <Path>\AIIdleBehaviour.xml</Path>
7.     <RootID>Behaviour:SimpleAIBase</RootID>
8.   </XmlBehaviourFile>
9. </NodeContents>
```

Code Sample 3.12: Example kAI node XML tag demonstrating that only the path is stored

When a node is storing a kAI behaviour, it can be seen that the serialised version stores only a path to the XML files (and what the path is relative to).

3.3. Editor

The editor is a C# visual application developed used WinForms[34] for the user controls and SlimDX[35] (a managed wrapper of DirectX) to render the behaviours. I also used SpriteTextRenderer [36]to assist in rendering text and graphics. The editor allows for the creation of kAI projects and kAI behaviours.

3.3.1. Key Classes

Class	Responsibility
`Editor`	The main program.
`kAIInteractionTerminal`	Provides a simple command line tool for executing operations within the program.
`kAIProject`	Represents a project and stores associated files.
`kAIInputManagerDX`	The input manager for the behaviour editor.
`kAIBehaviourEditorWindowDX`	The DirectX (DX) controller for the behaviour editor.
`PropertiesWindow`	The property grid that displays information about the selected node.

3.3.2. Key Technical Areas

3.3.2.1 Loading User Types

The designers assemble kAI behaviours through composition of code behaviours, kAI behaviours and function nodes. Code behaviours and functions are loaded via reflection[37][38] from a DLL. For this to be done, the designers selects DLLs to be loaded.

The tool then parses this DLL for relevant types for code behaviours:

```
1. foreach (Type lType in lAssembly.GetExportedTypes())
2. {
3.   if (lType.DoesInherit(typeof(kAIBehaviour)))
4.   {
5.        kAIINodeSerialObject SerialObject =
              kAICodeBehaviour.CreateSerialObjectFromType(lType);
6.   }
7. }
```

Code Sample 3.13: Loading relevant types from an assembly

Types and functions are imported when configuring the project. The designer is able to browse through a list and select relevant entries.

3.3.2.2 Interaction Terminal

I implemented a terminal for executing simple commands within the editor. I wanted a quick way of exposing methods to the terminal; a useful tool for testing functionality. To do this, I created an attribute that could be used to decorate specific methods. The editor then generates regular expressions using these methods.

```
1. // we need the method name and an opening bracket to start with
2. StringBuilder Regex = new StringBuilder(Method.Name + @"\s*\(\s*");
3. foreach (ParameterInfo Param in Method.GetParameters())
4. {
5.     // if parameter is a string, we need quotes followed by letters then another quote
6.     if (Param.ParameterType == typeof(string))
7.     {
8.         Regex.Append("\"(.+)\"");
9.     }
10.
11.    // if parameter an int then we need some digits
12.    else if(Param.ParameterType == typeof(int))
13.    {
14.        Regex.Append(@"(\d+)");
15.    }
16.
17.    // if this is not the last parameter, add a comma
18.    if(Param != Method.GetParameters().Last())
19.    {
20.        Regex.Append(@"\s*,\s*");
21.    }
22. }
23.
24. Regex.Append(@"\s*\)");
```

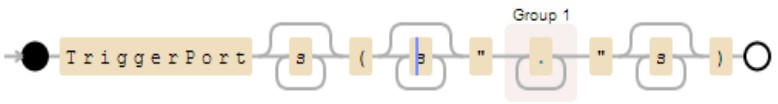
Code Sample 3.14: Code for generating regular expressions for interaction terminal functions

For example, I wrote the following method for triggering a specific port by ID. I then decorated it with the `[TerminalCommand()]` attribute.

```
1. [TerminalCommand()]
2. public static bool TriggerPort(string PortName)
```

Code Sample 3.15: Example interaction terminal function

The following regular expression was generated for this method:

`TriggerPorts*\(s*\"(.+)\"s*\)`	 *Fig 3.6: Graph of the regular expression generated for an example interaction terminal function*

When the user enters a command into the terminal, the editor matches the user input against all of the generated regular expressions. If there is a match, it can extract the parameters out and invoke the relevant method.

3.3.2.3 Rendering the Behaviours

How the behaviours are rendered is crucial to the editor's usability. The editor needs to be able to render a collection of textures (representing the nodes and ports) and lines (representing connexions between ports). It needs to be able to redraw these elements quickly so that the user can move the nodes about. Finally, the editor needs to support ways of interacting with the elements (for example, letting the user click and drag nodes about). I structured the program's behaviour editor code in such a way that I could switch between different rendering technologies. This allowed me the flexibility to consider different options.

WinForms[39]

This was my initial solution. Its main advantages was that it contained a lot of the boilerplate code I would need. For example, controls in WinForms already have support for context menus, being clicked and dragged and other standard user interactions.

The problem with WinForms was performance. Even on moderately sized behaviours, the screen would flicker when the nodes were moved. Further, it didn't provide mechanism for drawing lines between ports.

DirectX[40]

DirectX is a rendering technology used to power the Xbox and Windows games[41] and would easily be able to handle rendering the behaviours. However, it is very low level and general purpose. As a result, it doesn't implement GUI functionality. This means I would need to write code that handles things like detecting when you click on nodes. Also I did not have much experience using DirectX directly and as such this option would be a steep learning curve.

MSAGL[42]

Another option was Microsoft Automatic Graph Layout - MSAGL (formerly GLEE), a library developed by Microsoft for rendering graphs. Not only would this be able to render behaviours efficiently, it would handle things like arranging the behaviours.

However, behaviours aren't actually graphs since each node has ports which connect to other ports. Therefore, the behaviours would become quite messy as a node would have to be represented as a super node in the graph (a central node connected to port nodes). The port nodes would then in turn be connected to other port nodes.

Conclusion

Ultimately, the performance issues meant the I could not use WinForms. As MSAGL would mean that even the most simple behaviours would look confusing, I used DirectX through SlimDX[43] – a managed wrapper around DirectX. A sprite rendering library (SpriteTextRenderer[44]) allowed me to quickly render the behaviours. This was the best result that meant I wouldn't have to make the compromises the other solutions required.

3.3.2.4 R-Tree

As DirectX does not provide an input management system, I had to implement my own. This would handle detecting, for example, when nodes were clicked. To do this, I used a data structure called an R-Tree[45]. R-Trees are used for storing rectangles in a tree structure to allow for spatial searches to be performed efficiently.

For each node and port, I stored a rectangle associated with this item in an R-Tree. Then, each frame, the tree is searched for rectangles containing the mouse position. This allowed the editor to detect what the user had clicked on, if the user was hovering over a port etc.

My R-Tree had three main operations:

```
1. // Add a rectangle to the tree
2. public void AddRectangle(Rectangle Rectangle, T Contents)
3.
4. // Find rectangles under a point
5. public IEnumerable<T> GetRectsContainingPoint(Point Point)
6.
7. // Remove a rectangle from the tree
8. public T RemoveRectangle(Rectangle Rectangle)
```

Code Sample 3.16: Interface for the R-Tree Structure

Adding a rectangle

Each node in the tree has up to 4 children and stores a rectangle which contains all of the rectangles of its children. When a rectangle is added to a node, the node must expand its rectangle to make sure it contains the new rectangle

Next, a heuristic is used to determine how the new node should be added to the tree. I used a heuristic based on the smallest volume increase. When a rectangle is added to the tree, all possible places it could be inserted are evaluated and the one with the smallest rectangular area increase is selected.

```
Choose Rectangle By Smallest Volume(new rectangle: Rectangle, child nodes: Rectangle[])
    if |child nodes| < 4
        best cost = 25
        chosen child = new child
    else
        best cost = ∞
    end if
    ∀ child ∈ child nodes
        if child contains new rectangle
            best cost = 0
            chosen child = child
        else
            expanded rectangle = new rectangle ∪ child
            expanded area = area(expanded rectangle)
            if expanded area < best cost
                best cost = expanded area
                chosen child = child
            end if
        end if
    end loop
end method
```

Algorithm 3.4: Heuristic for evaluating which child node to add a new rectangle to in the R-Tree

This chooses the child rectangle that will have the smallest increase in area. Since there is a limit of 4 nodes (to stop the tree becoming too shallow), there is a cost associated with creating a new node, in the example above this is 25. This is to encourage putting the child in an existing node if it won't result in too big an increase. Under testing, this produced well balanced trees for example kAI behaviours.

Here is an example collection of nodes and the R-Tree generated.

Fig 3.7: Root of the R-Tree for an example behaviour

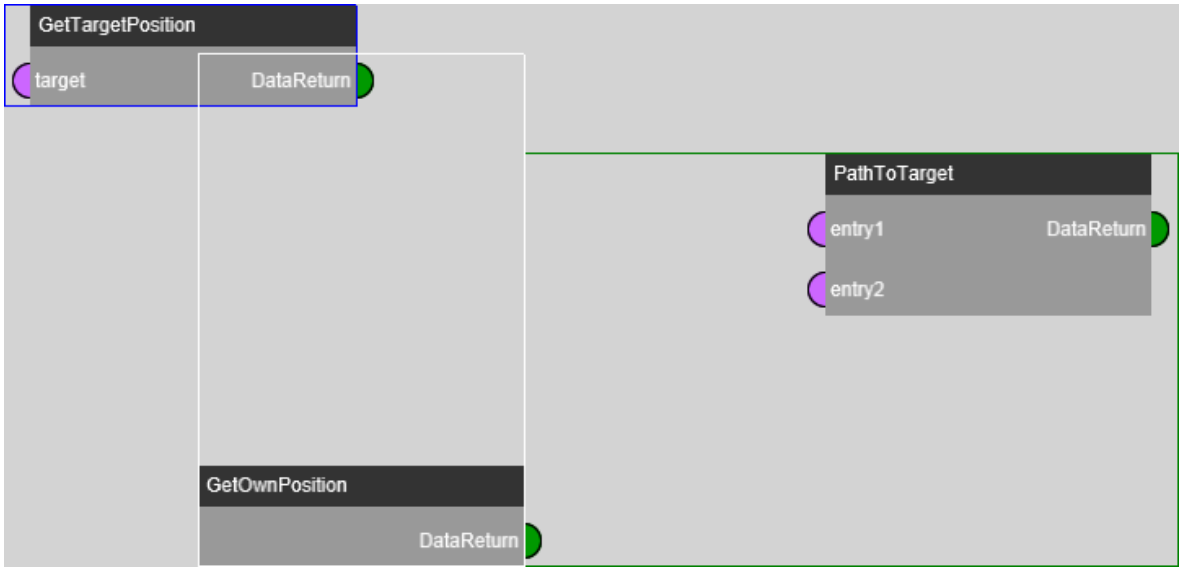


Fig 3.8: Tier 1 of the R-Tree - root split into three nodes

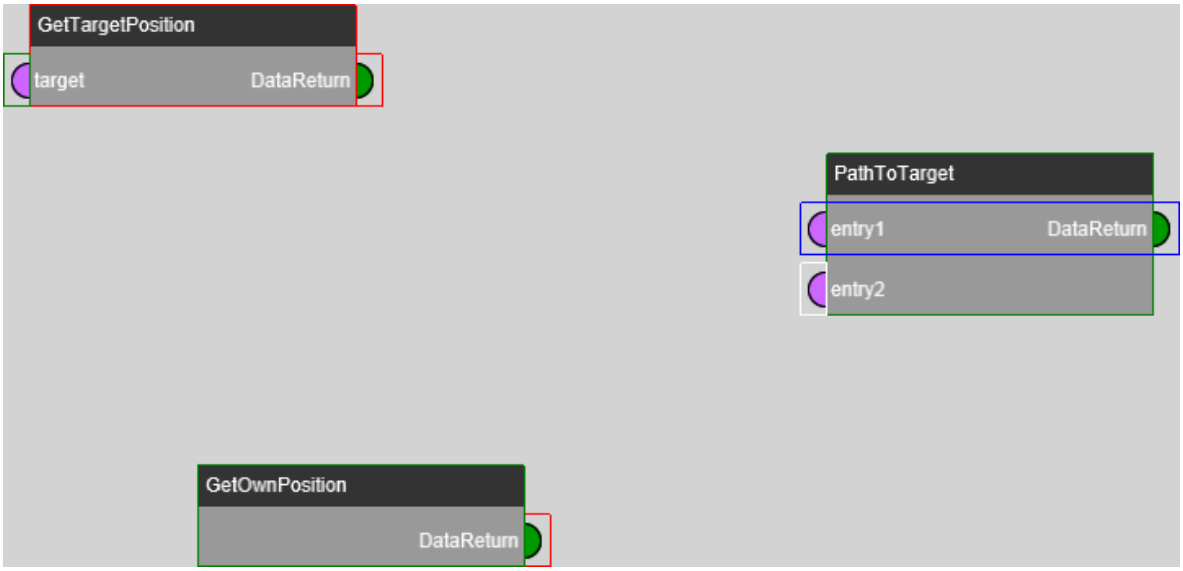


Fig 3.9: Tier 2 of the R-Tree - blue node splits into two leaves, white node splits into two, green node splits into four

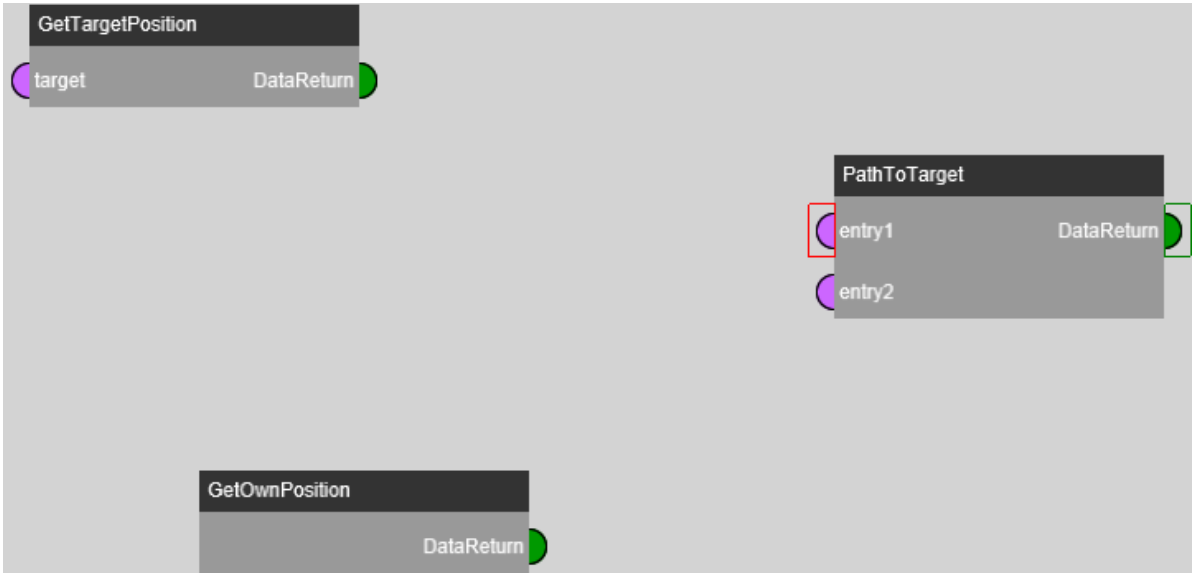


Fig 3.10: Tier 3 of the R-Tree - Last non-leaf node splits

For an example behaviour, this produces a rectangle tree like:

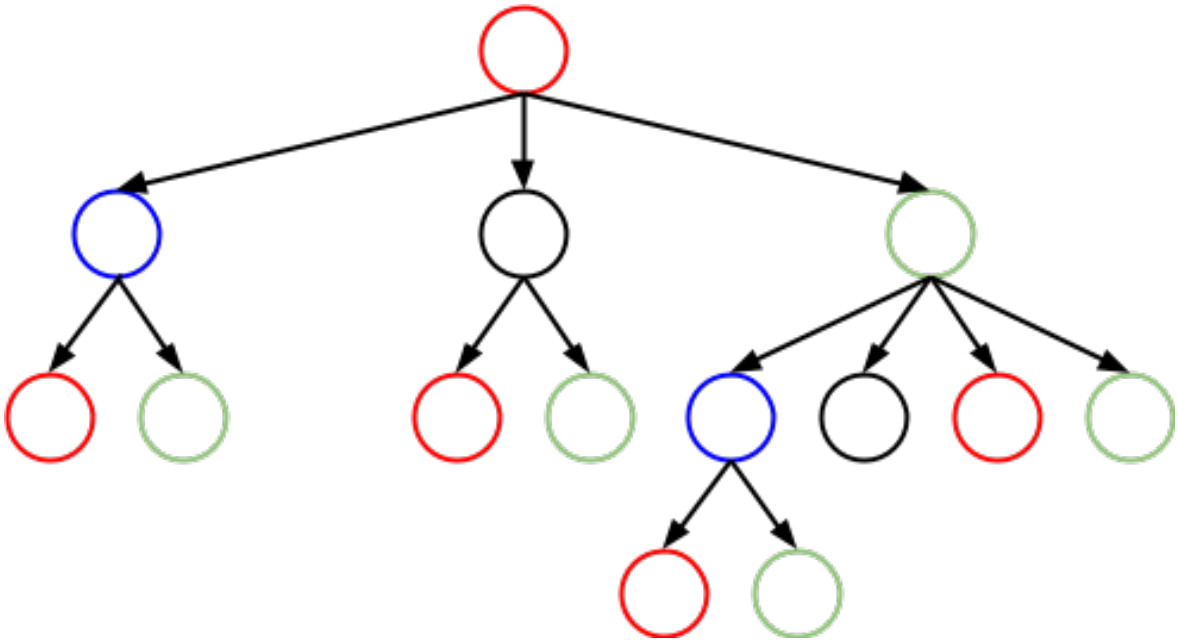


Fig 3.11: The tree created - note the black circles represent the white boxes in the above pictures

Having constructed the tree, it is possible to query the tree, giving it a specific point to check what rectangles are under the tree.

Getting Rectangles that contain a point

The next operation of the R-Tree is to query it with a point and find out what rectangles stored in the tree contain that point. To do this, a search is performed on the children whose rectangle contains the point.

```
Get Rectanlges Under Point( point : Point , child nodes : Rectangle[])
        List<Rectangle> rectangles = Ø

        ∀ child ∈ child nodes
                if (child contains point)

                        if (child.contents ≠ empty)
                                rectangles = rectangles ∪ {contents}
                        end if

                        child rectangles = Get Rectangles Under Point( point , child.children)
                        rectangles = rectangles ∪ child rectangles
                end if
        end loop
        return rectangles
end method
```

Algorithm 3.5: For finding rectangles in an R-Tree under a specific point

Adjusting the Tree

One challenge I had when implementing the R-Tree is the user can click and drag in the editor to move the entire behaviour network about.

I wished to avoid recreating the tree every time the user moves the camera as it is expensive to recreate the whole tree. To resolve this, when the mouse position is checked against the rectangles, it is offset by how much the user has panned about within the behaviour. This way, it is like the camera moved and the nodes remained fixed.

Another complication is when users do click and drag, some ports do not move (the global ports of that kAI behaviour are anchored to the edge of the window). To resolve this I have two R-Trees: one for fixed ports and one for movable elements. The fixed elements R-Tree does not perform the offset calculation.

Removing an element from the tree

When moving an individual node, the tree needs to be adjusted. Since this is just a single node, it is less expensive. This operation is performed once the user has finished dragging by removing and reinserting the node. This optimisation can be performed since the user cannot click on anything whilst they are still dragging a node.

The remove node operation works as follows:

```
Remove Rectangle(rectangle : Rectangle , child nodes : Rectangle[ ])
        ∀ child ∈ child nodes
                if (contains(child.rectangle , rectangle))
                        if (child.rectangle = rectangle)
                                return child
                        else
                                result = Remove Rectangle(rectangle , child.children)
                                if result ≠ Ø
                                        children = children \ { child }
                                        Optimize Rectangles(child nodes)
                                        return result
                                end if
                        end if
                end if
        end loop
end method
```

Algorithm 3.6: For removing a rectangle from an R-Tree

Where $Optimize\ Rectangle$ takes the node's rectangle and makes it as small as possible. This is required as a rectangle has been removed from within a node. This means the super rectangle representing the whole of that node could be made smaller.

3.4. Debugger

The debugger is an API in the core of kAI which allows for real time debugging of kAI behaviours. The editor is able to interface with this API. It can use this information to display the debug information of a kAI behaviour as it is running in a game on a separate process.

3.4.1. Key Classes

Class	Responsibility
`kAIDebugInfo`	Base class which all debug information is derived from (e.g. `kAIXmlBehaviourDebugInfo` which stores the debug information for a kAI behaviour.
`kAIDebugServer`	Manages the game side of the debugging - storing what debug information is available and where it can be retrieved from.
`kAIBehaviourDebugStore`	Represents the debug information of a single entity in the game.
`kAIDebugger`	Handles the editor side of debugging - loading the debug information upon request.

3.4.2. Key Technical Areas

3.4.2.1 Real Time Inter-Process Communication

Since the game will be running in a separate process, it needs to be able to communicate in real time with the editor. I initially explored named piped which are .NETs recommended method of interprocess communication[46]. However, the version of Mono (an open source implementation of C# and the .NET framework) that Unity uses does not support these.

Instead I used Memory Mapped Files[47] via an open source wrapper around the Windows function calls. Memory Mapped Files provide a file-like interface to memory which can be opened on different processes.

Each entity in the game registers itself with the DebugServer. This means the behaviour is added to a central memory mapped file. This memory mapped file contains a list of all the kAI behaviours running in the game and where their debug information could be found.

```
1. public static kAIBehaviourDebugStore AddBehaviour(kAIXmlBehaviour Behaviour, string ID)
2. {
3.         kAIBehaviourDebugStore NewStore = new kAIBehaviourDebugStore(Behaviour, ID);
4.         AddToList(NewStore);
5.
6.         return NewStore;
7. }
```

Code Sample 3.17: Adding a kAI behaviour to the debug server

The entity then owns and updates the `kAIBehaviourDebugStore` every frame. This writes the latest debug information in a separate memory mapped file.

```
1. public void Update()
2. {
3.     kAIXmlBehaviourDebugInfo DebugInfo = Behaviour.GenerateDebugInfo();
4.     WriteDebugInfoToFile(DebugInfo, BehaviourFile);
5. }
```

Code Sample 3.18: Updating a debug store for an individual kAI behaviour

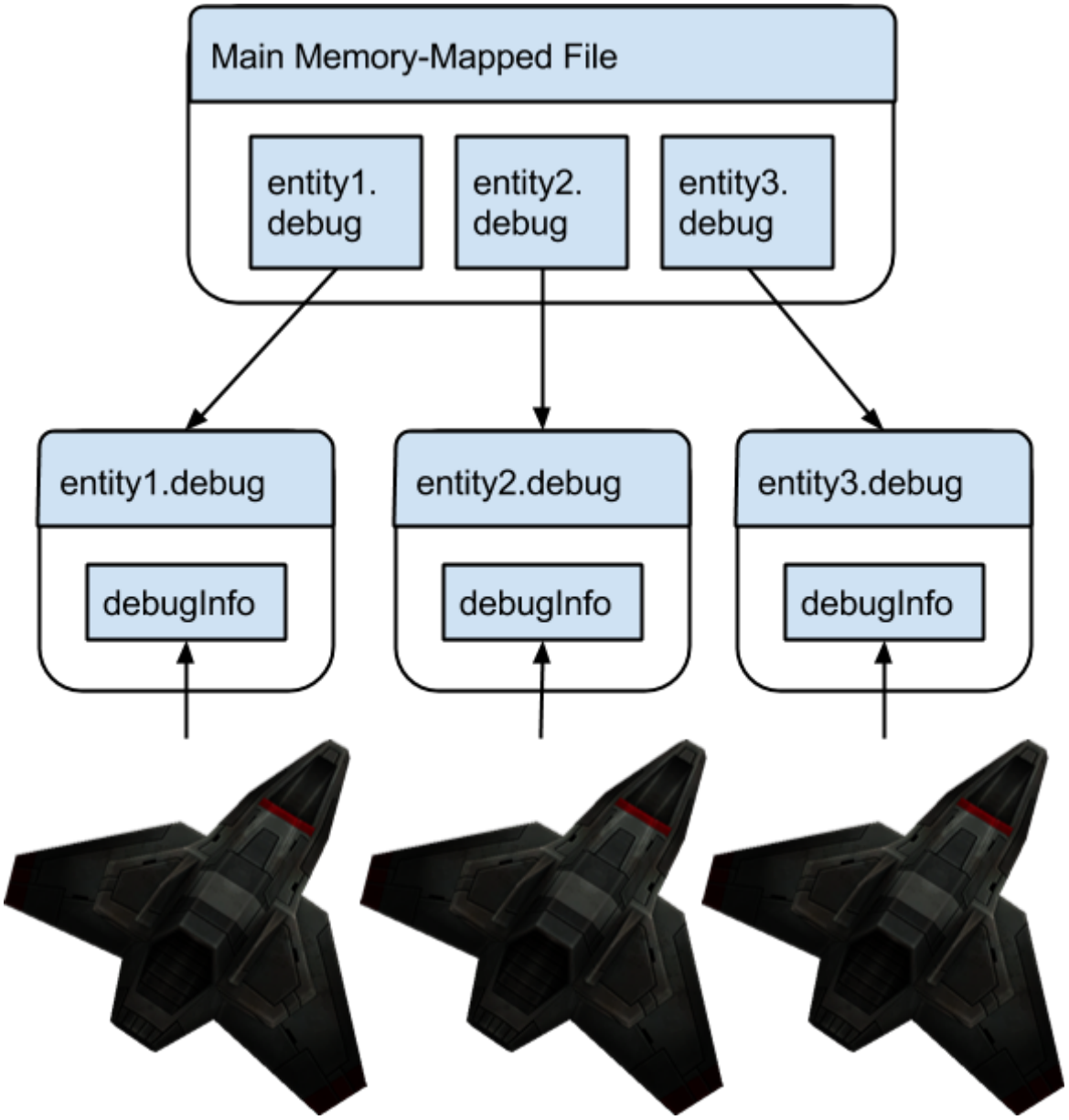


Fig 3.12: Different memory mapped files and what the contain

The editor (or indeed, any debugger) then opens the Main file to see a list of kAI Behaviours being simulated.

```
1. public IEnumerable<kAIBehaviourEntry> GetAvaliableBehaviours(Stream inStream)
2. {
3.     BinaryFormatter writer = new BinaryFormatter();
4.     return (IEnumerable<kAIBehaviourEntry>)writer.Deserialize(inStream);
5. }
```

Code Sample 3.19: Code for getting kAI Behaviours that are debug-able

The debugger can then choose to load the debug information of a specific behaviour. This then opens the relevant memory mapped file. It can then de-serialize the `kAIXmlBehaviourDebugInfo` contained within the file. This contains all the debug information about that kAI behaviour, such as the current state of each of the nodes within it. See figure 3.13 to see the debug information that each type has.

To prevent race conditions, to access the file the process must acquire a semaphore[48]. Again I used an open source wrapper around the Windows operations. I created a root semaphore and a semaphore for each kAI behaviours. As with the location of the memory mapped file, the root memory mapped file stores the names of the semaphores for each of the kAI behaviours.

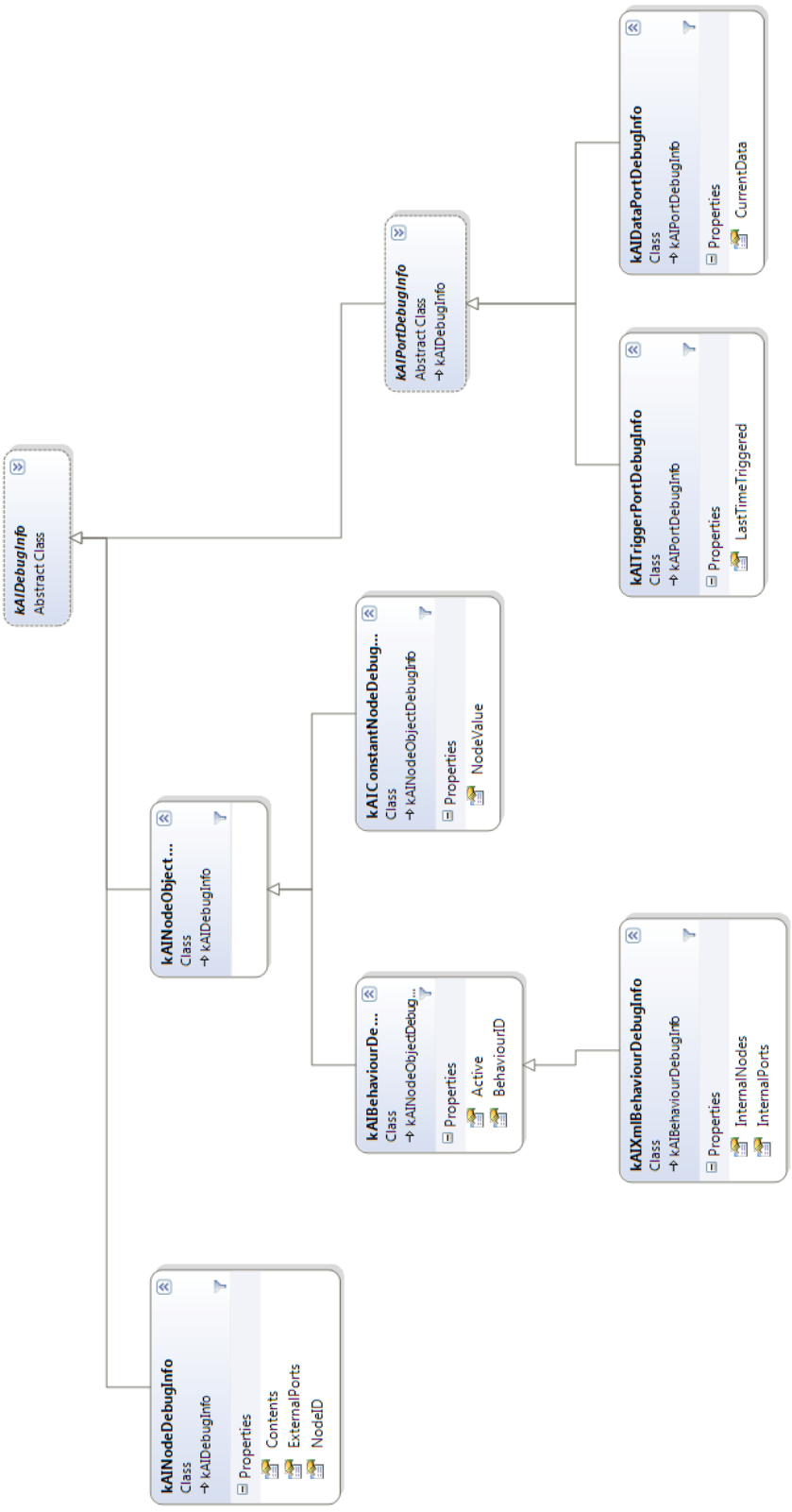


Fig 3.13: Contents of debug information

Once the debug information has been loaded, the editor displays the information in a useful way. For example, the trigger port debug information contains when it was last triggered. This allows the editor to display a tick when the trigger port is triggered.

```
1. void DisplayTriggerPortDebugInfo(kAITriggerPortDebugInfo debugInfo)
2. {
3.     TimeSpan TimeSinceLastTrigger = DateTime.Now - debugInfo.LastTimeTriggered;
4.
5.     if (TimeSinceLastTrigger <= new TimeSpan(0, 0, 0, 0, 500))
6.     {
7.          float Alpha = 1.0f - ((float)TimeSinceLastTrigger.Milliseconds / 500.0f);
8.          SpriteRenderer.Draw(EnabledTexture, Alpha);
9.     }
10.}
```

Code Sample 3.20: Function for displaying a tick when a trigger port is triggered

Conclusion

The debugger provides real time information to developers, allowing them to see why the AIs are behaving as they are. This addresses the problem of debugging AI in games since you can leave the game running and in a second window observe what is causing the behaviour.

3.5. Example Games

3.5.1. Key Classes

Class	Responsibility
`AIBehaviour`	The component attached to AI objects in the game. Has a property to specify which kAI Behaviour to load.
`AIDebugBehaviour`	The component that handles making a specific AI entity have its kAI behaviour debuggable.
`MoveTo`	A code behaviour that handles moving the ship

Creating the examples consisted of 2 main tasks. Firstly, I had to implement the game, including the code behaviours that would be the primitive operations of the AI. Then I had to create the kAI behaviours that would actually control them.

I created two example games to demonstrate kAI: a sword duel and a strategy game. The games were made in Unity3D – a game engine for creating 2D and 3D games[49]. It is available for free for small budget games. It consists of a behaviour driven engine and an editor for assembling levels in.

3.5.2. Sword Duel

Fig 3.14: Screen shot from the sword duel game

The sword game features a player and an AI entity having a sword fight. The AI moves towards the player and tries to hit the player with the sword. When the AI is on low health, they run away from the player until they recover.

The AI uses a couple of code behaviours - one to move towards the player and one to swing the sword. The kAI behaviour has a data port that communicate the current health of the AI.

The kAI behaviour contains three embedded kAI behaviours: one to handle when the health is above the threshold; one to handle when it is running away; and one to control when to swing the sword. The global behaviour handles when to activate and deactivate these behaviours.

3.5.3. Strategy Game

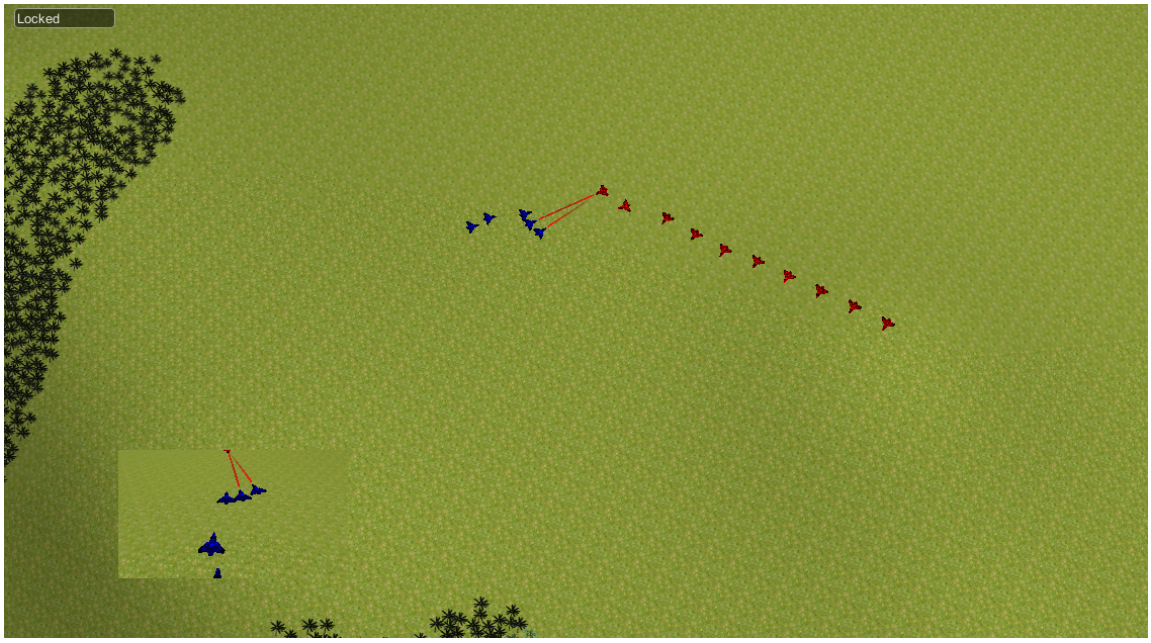


Fig 3.15: Screen shot of the strategy game

The strategy game consists of two opposing armies of ships. The player controls one side. Each of the ships runs a kAI behaviour that controls it. They then belong to a squad of 5 ships, which is run by another kAI behaviour that gives orders to each of the ships. The player issues orders to the squads.

3.5.3.1 The Individual AI

The individual AI, which is run on each of the ships, determines how they behave. It receives orders from the squad leader AI and acts accordingly. It has a self preservation behaviour which looks at the current health of the ship and, if this falls below a certain threshold, it will run away. It also looks for nearby targets to automatically shoot if it isn't doing anything.

All of these behaviours make use of some lower level kAI behaviours such as shoot target which checks if they are in range of the target. If the target is in range of the AI, it activates the code behaviour to shoot it. If not, then it activates the code behaviour to move towards it. This demonstrates the modular way behaviours can be created in kAI.

3.5.3.2 The Squad AI

In addition to the individual ships running a kAI behaviour, each squad is also running a separate kAI behaviour. It receives orders from the player and distributes them to the squad. This allows for implementation of formations. When the individual AI is told to go somewhere, it goes directly to it. The

squad AI therefore generates a set of points – offset from each other – to force the squad to move in arrow formation.

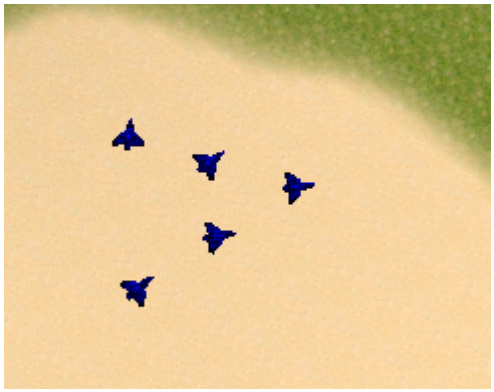


Fig 3.16: A squad of ships in arrow formation

Further, when the squad wants to attack some target, it will first issue a move order to each of the squad members, telling them to line them up in a semi-circle around the target. Then it issues an attack order causing the ships to move straight towards the target.

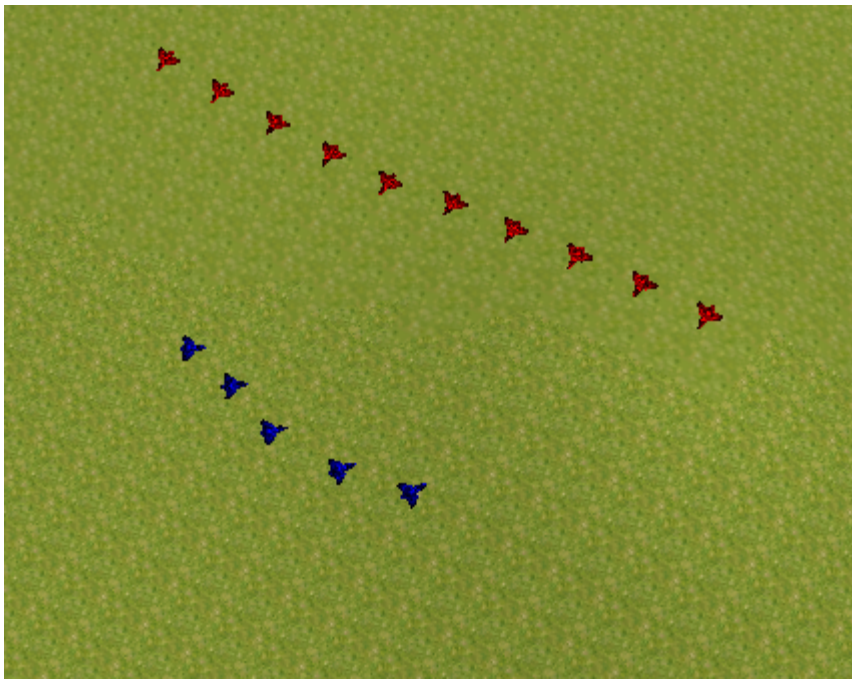


Fig 3.17: Semi-circle attack formation before the ships attack their target

The formations are a code behaviour that are embedded in to the squad behaviour. For the arrow formation, the index of the squad member and where the squad is moving to are used to generate an individual order for that squad member.

```
1. IndividualOrder CreateOrder(int index, SquadMember memeber, Vector3 destination,
   Vector3 direction)
2. {
3.     /*            0
4.      *          1 | 2           depth: 1
5.      *        3   |   4         depth: 2
6.      *      5     |     6       depth: 3
7.      *            |
8.      * side:   -1 | +1
9.      * */
10.    if (index == 0)
11.    {
12.        return new IndividualMoveOrder { Destination = destination };
13.    }
14.    else
15.    {
16.        index = index - 1;
17.
18.        // this is the row in the arrow rank
19.        int depth = (index / 2) + 1;
20.
21.        int side = index % 2; // 0 means left hand side, 1 means right hand side
22.
23.        // transform so -1 means left hand size, 1 means right hand side
24.        side *= 2;
25.        side -= 1;
26.        Vector3 perpToDirection = Vector3.Cross(direction, Vector3.up);
27.
28.        Vector3 newDestination = destination;
29.
30.        // move the position back along the direction of motion
31.        newDestination -= rowSize * direction.normalized * depth;
32.
33.        // move the position side ways according to the size of the ship
34.        newDestination += columnSize * perpToDirection.normalized * side * depth;
35.
36.        return new IndividualMoveOrder { Destination = newDestination };
37.    }
38.}
```

Code Sample 3.21: Generating positions for an arrow formation

3.5.4. A Sample Behaviour

This is a detailed walk through of the simple behaviour I created for the sword fighting game. The images are taken from the kAI editor.

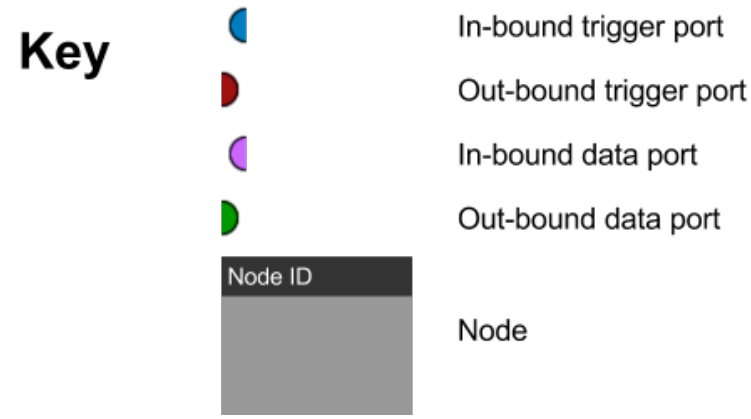


Fig 3.18: Key for elements of example behaviours

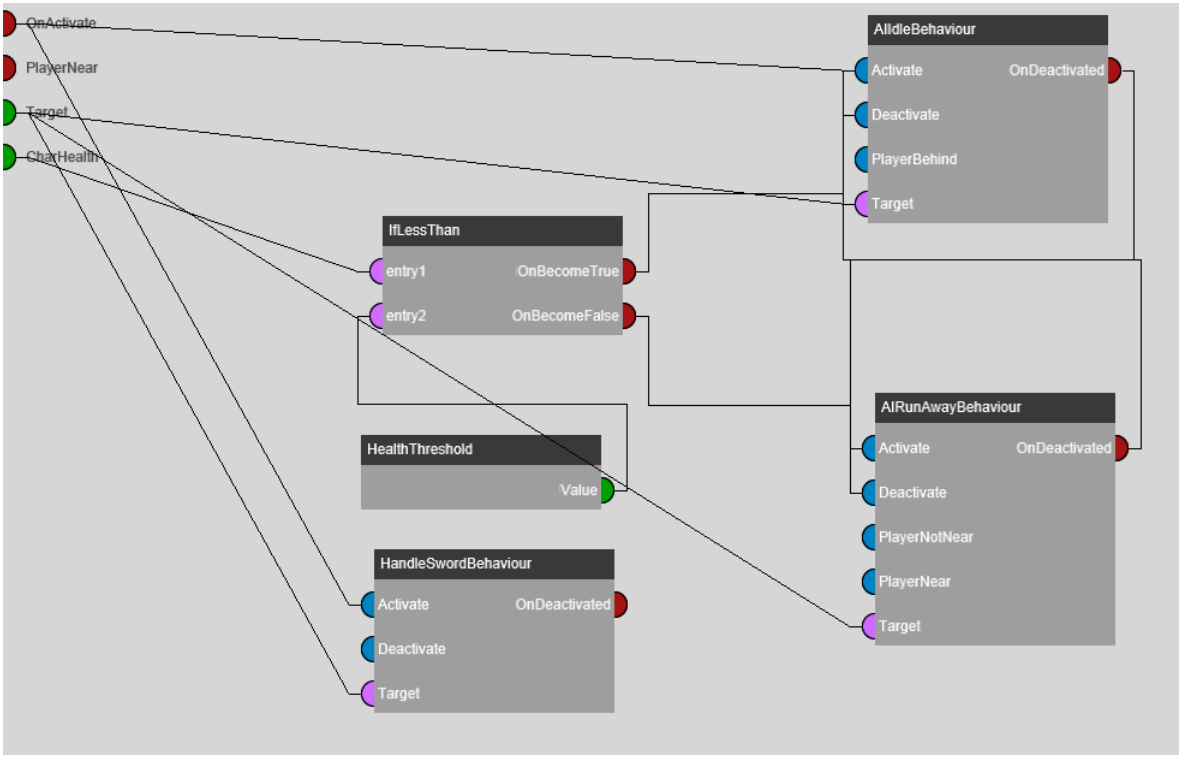


Fig 3.19: Example behaviour - the root behaviour

The root behaviour is the one that would be attached to the AI in the game. There three kAI behaviours in the network. AIIdleBehaviour and AIRunAwayBehaviour operate as a finite state machine, with the root behaviour switching between them. Then there is the HandleSwordBehaviour which deals with swinging the sword.

The OnActivate trigger port in the top right is triggered when this AI becomes active. This is connected to the Activate of the AIIdlleBehaviour and the HandleSwordBehaviour, causing these kAI behaviours to become active.

There is also a port labelled Target which contains what target the AI is going after. Finally, there is a data port containing a floating point number, CharHealth, which contains the health of the AI.

In the centre there is a function node, labelled IfLessThan, which compares the value in the CharHealth data port with the value stored in the constant floating point node, HealthTheshhold.

Since the function returns a boolean, when the node was created, it was configured to have a trigger port that is triggered when the value becomes true or becomes false.

When the CharHealth data port is less than the HealthThreshhold value, OnBecomeTrue is triggered. This trigger port is connected to the Deactivate trigger port of the AIIdleBehaviour, deactivating this behaviour. This causes the trigger of the OnDeactivated trigger port of this node. This, in turn, is connected to the Activate trigger port of the AIRunAwayBehaviour, activating that behaviour instead.

Considering the update algorithm, what will actually happen when the health falls below the health threshold. The function will be evaluated in the kAI update and trigger the port. This will mark both it and the deactivate port as triggered. At the start of the subsequent frame, these will both be released and the deactivate action will be taken. In the update, since the node has been deactivated, the OnDeactivate port will be marked, as will the port is it connected to: Activate on the AIRunAway behaviour. Finally, the third frame, these two ports will be released, activating the AIRunAway behaviour.

When the health value rises above the threshold, the opposite happens and the AIRunAwayBehaviour is deactivated, activating the AIIdleBehaviour.

In the HandleSwordBehaviour, a simple structure making use of the code behaviour, SwingSwordAction, can be seen.

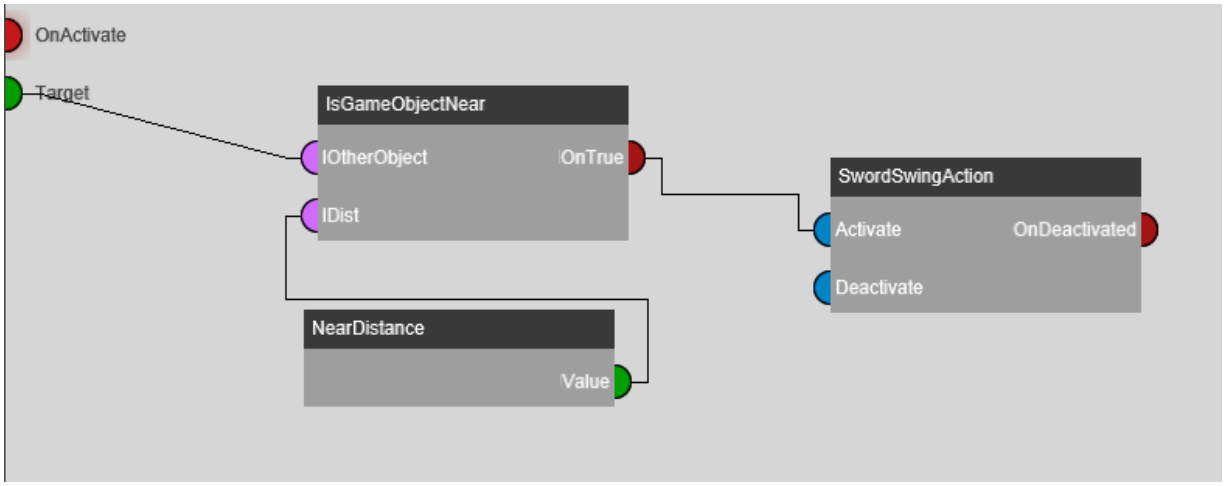


Fig 3.20: The sword swing kAI Behaviour

There is a function node called IsGameObjectNear which takes the target from the data port and a floating point value. The first parameter of a function can be specified, when configuring a function node, to be the parameter passed in as the object. In this example, the function IsGameObjectNear takes 3 parameters, the first being a GameObject that represents the AI.

Again this is configured to trigger on true since it returns a boolean. This TriggerOnTrue trigger port is connected to the Activate trigger port of the SwordSwingAction, causing the sword to swing. This code behaviour handles the animation and game play code behind making the sword swing.

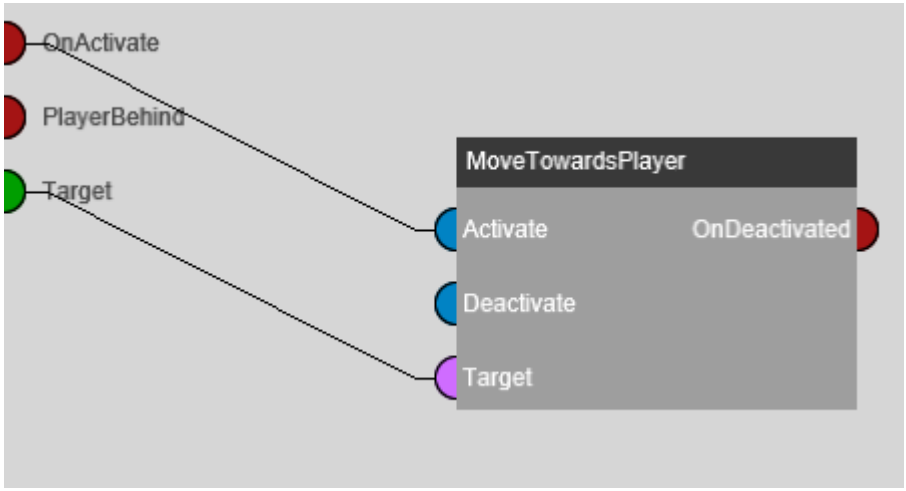


Fig 3.21: AIIdleBehaviour for controlling the AI when it is not on low health

Looking in the AI Idle Behaviour we can see that the AI moves towards the player using the MoveTowards code behaviour. This handles whether the player is in front of or behind the AI.

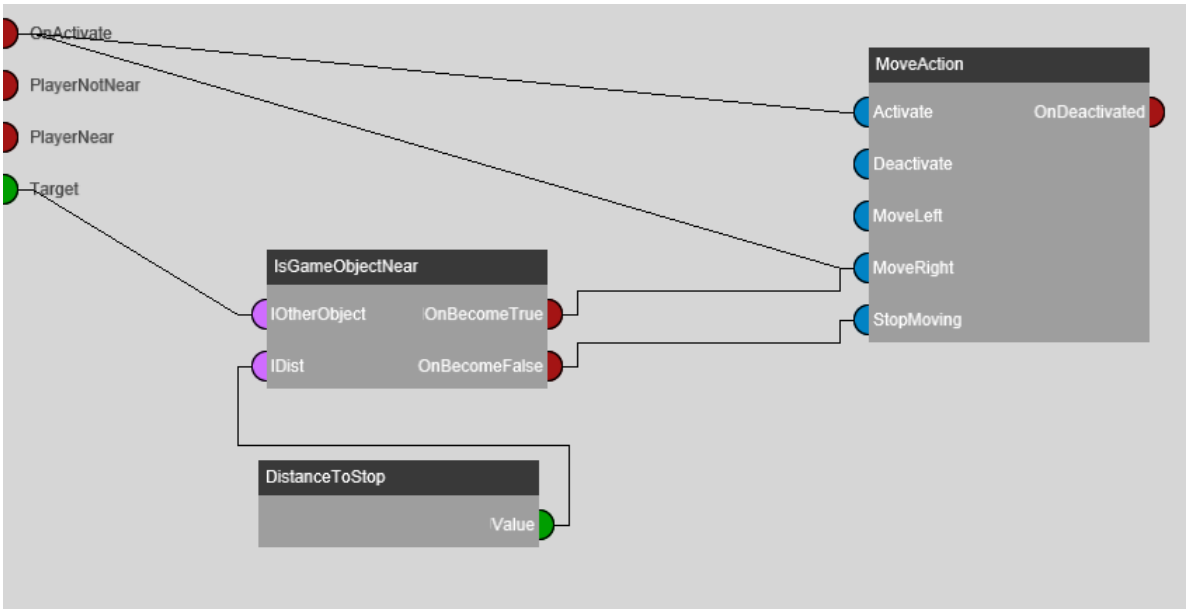


Fig 3.22: AIRunAway kAI Behaviour handles what the AI does when it is on low health

Finally, in the AIRunAwayBehaviour, the AI examines how near it is to the player. This is again done via the IsGameObjectNear function. If the AI is near, it moves away from the player, otherwise it stands still

4. User Guide

Below are two guides on using kAI for game studios. The first is aimed at designers using the tool to create kAI behaviours. The second is for coders to integrate kAI into the game engine and develop the behaviours that designers will use.

4.1. Designers Guide

This section is aimed at designers wanting to create kAI behaviours in the editor. It assumes some code behaviours and functions have been developed by the code team.

4.1.1. Introduction and Terms

kAI is a tool for creating AIs for games. In the editor, you will create kAI behaviours. These are made up of nodes. Nodes can be:

- **kAI behaviours** – other behaviours created with the tool
- **Code behaviours** – primitive actions developed by coders that your AI can do (such as moving somewhere)
- **Functions** – utilities such as mathematical operations developed by the coders (such as computing the average of a set of vectors)
- **Constants** – values that are used in calculations (such as what health should the AI run away)

These nodes have external ports which are used to connect them together. The ports come in 3 types. Each port has a direction; outbound ports are connected to inbound ports of the same type.

Port type	Inbound use	Outbound use
Trigger port	Activating and deactivating node, telling code behaviours when something has happened	Game triggers them when something happens. Nodes can trigger them when something happens internally
Data port	Receiving data of an arbitrary type such as the entities health	Sending data of an arbitrary type such as the entities health
Enumerable data port	Having multiple data ports connect to one port and combine all the entries into a list.	Enumerable ports cannot be outbound.

kAI behaviours have their own internal ports which can be used when designing the behaviour.

4.1.2. Setting up a kAI Project

A kAI project stores all the kAI behaviours that have been created. You must first create this before you can start creating kAI behaviours.

4.1.2.1 Project Name and Location

To create a project, open up the kAI Editor and click File>New Project. This will bring up the project properties dialogue (accessible via the Project menu once your project has been created).

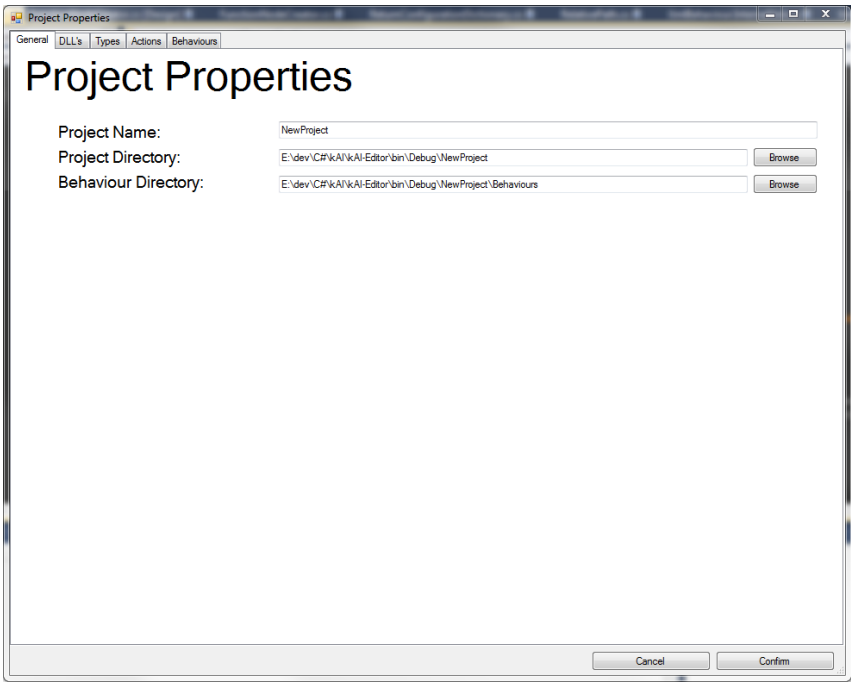


Fig 4.1: Project Properties Dialogue for creating a new project

First you need to name your project. This name should identify the project. Next, in the two boxes below, select a root directory for your project and your behaviours.

The project directory will contain the project file (which stores what behaviours have been made). This is not needed when the game is exported. Therefore, it should not be included in your game's build directory.

The behaviours directory is where all kAI behaviours will be saved. These are required for the game export and should be located accordingly.

4.1.2.2 Project DLLs

In the DLLs tab of the project properties, you need to specify the location of any DLLs (libraries of code) the project will need. These are where the code behaviours are loaded from, the coding team should be able to locate them for you. If they have any dependencies, you will be asked to load them too.

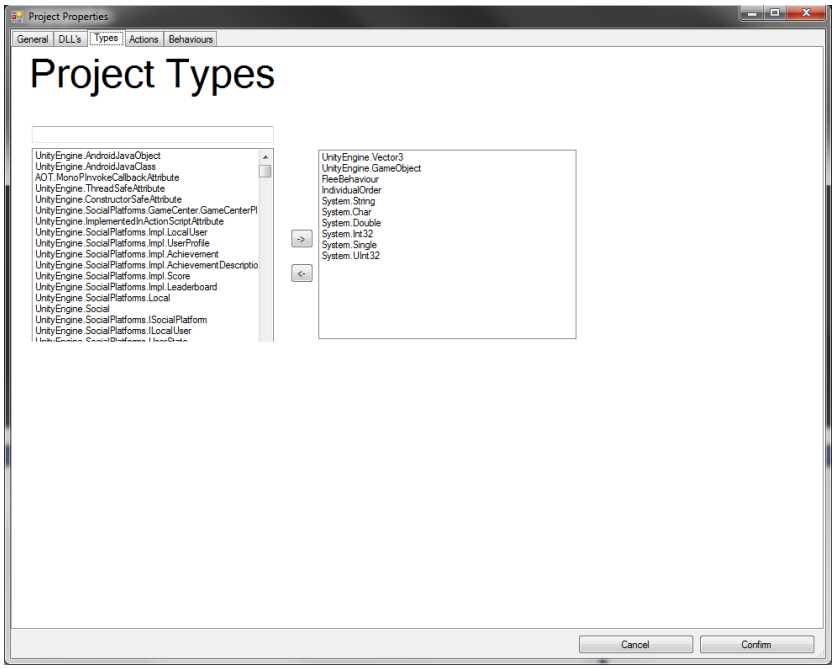


Fig 4.2: Types tab of the project set-up dialogue

4.1.2.3 Project Types

In the types tab you need to specify what types (like vectors) you want exposed to be used as data ports. The editor will have selected some default types. The left hand side is a list of types loaded from the selected libraries. Select the ones you want and click the arrow to add them. You can use the search box the filter the types to find a particular type.

4.1.2.4 Project Functions

In the actions tab, you select what functions you want to be able to make function nodes out of.

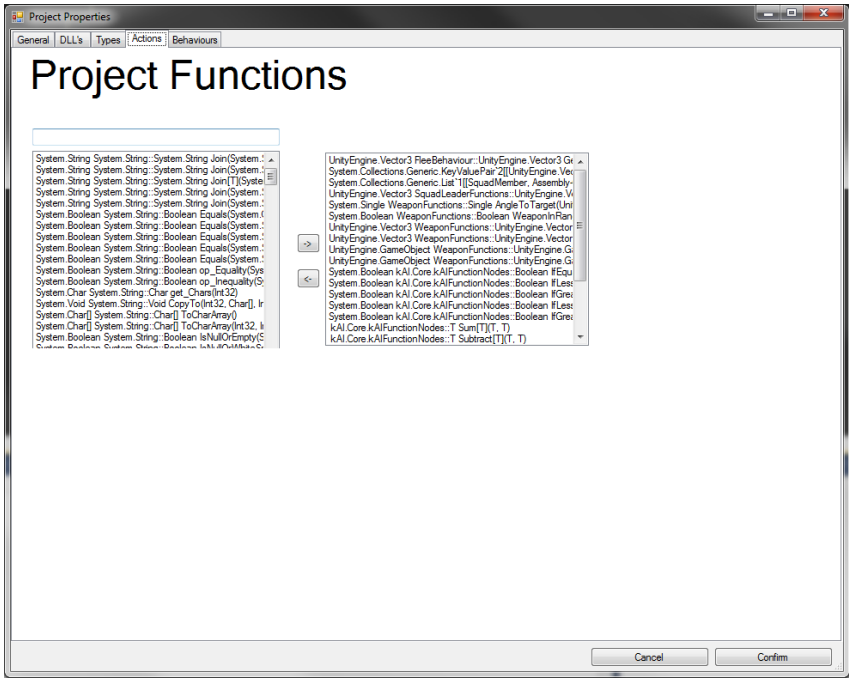


Fig 4.3: Actions tab of the project set-up dialogue

Click confirm to create your project. This will generate a .kAIProj file which contains all the information about your project such as what behaviours it has and what libraries it references.

4.1.3. Creating your first kAI behaviour

Having set up the kAI project, you now need to create some kAI behaviours.

4.1.3.1 Creating the behaviour

To create a new kAI behaviour, click Behaviours>Create New Xml Behaviour. This will take you to the Behaviour Properties dialogue.

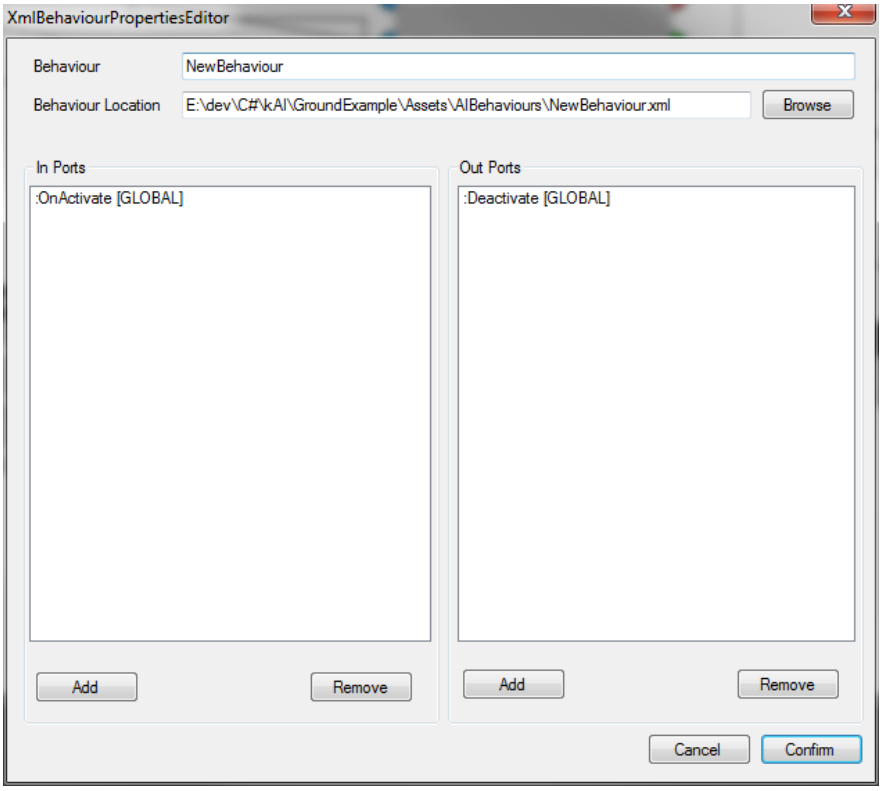


Fig 4.4: New behaviour properties dialogue

This window is used to give the behaviour a name, where it should be saved and what internal ports it should have. Should you need to adjust the ports, this window is accessible via Behaviours>Behaviour Properties once you've made a behaviour.

Name and Location

The name should be a unique ID for this behaviour. It should indicate what the behaviour does. The location is simply where the file is saved. This should be accessible from the game as this is the file that the behaviour is loaded from. By default, this will be in the behaviours directory you specified when creating the project.

Internal Ports

You need to define any ports you want the behaviour to have. These are accessible from outside the behaviour and can be connected to things inside the behaviour. There are two lists, the in-bound ports and the out-bound ports.

The in bound ports will be triggers and data your behaviour receives. What goes here depends on your behaviour. It could include a data port of type float containing the health of the AI or a trigger port for when the player hits the AI.

The out bound ports are data and triggers your kAI behaviour wants to send out. If you are creating a root behaviour (i.e. the one you are going to attach to an AI entity) you probably won't need these. If however, you are using this kAI behaviour within another kAI behaviour, then they can be used to communicate things out to the parent kAI behaviour. For example, a vector indicating what direction this behaviour wants to move in (this can be seen in the steering behaviours example in section 4.1.4.2).

Click add to add a port to either list, you will be presented with the Add Port dialogue box.

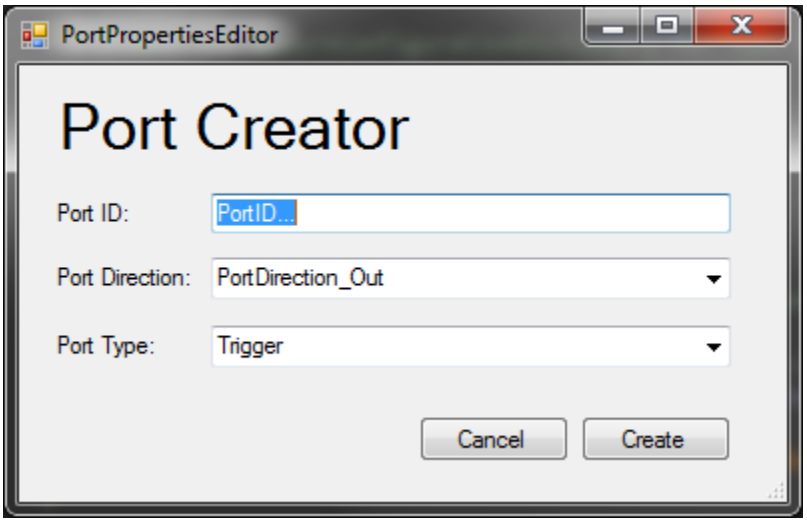


Fig 4.5: New port dialogue

You need to name your port (which must be unique across all the ports this behaviour defines). If you want a trigger port, the type should be trigger, otherwise select the type of data port you want.

Once you've added the ports you require, click confirm to create the behaviour.

4.1.3.2 Adding nodes to the behaviour

Once you've created your behaviour, you will be presented with the behaviour editor window, with the ports you specified (plus the built in ones: OnActivate – for when the behaviour becomes active – and Deactivate – used when you want to deactivate the behaviour). The properties window will also appear. When you select ports and nodes, this will provide additional information. It can be closed and reopened via the View menu.

To create behaviours you need to add nodes and connect them. To add a node, right click anywhere in the behaviour. Click Add Node to add a behaviour (either a code behaviour or a kAI behaviour), Add Function to add a function node or Add Constant to add a constant.

Adding a behaviour node

To add a behaviour, after clicking AddNode you will be presented with a list of code and kAI behaviours.

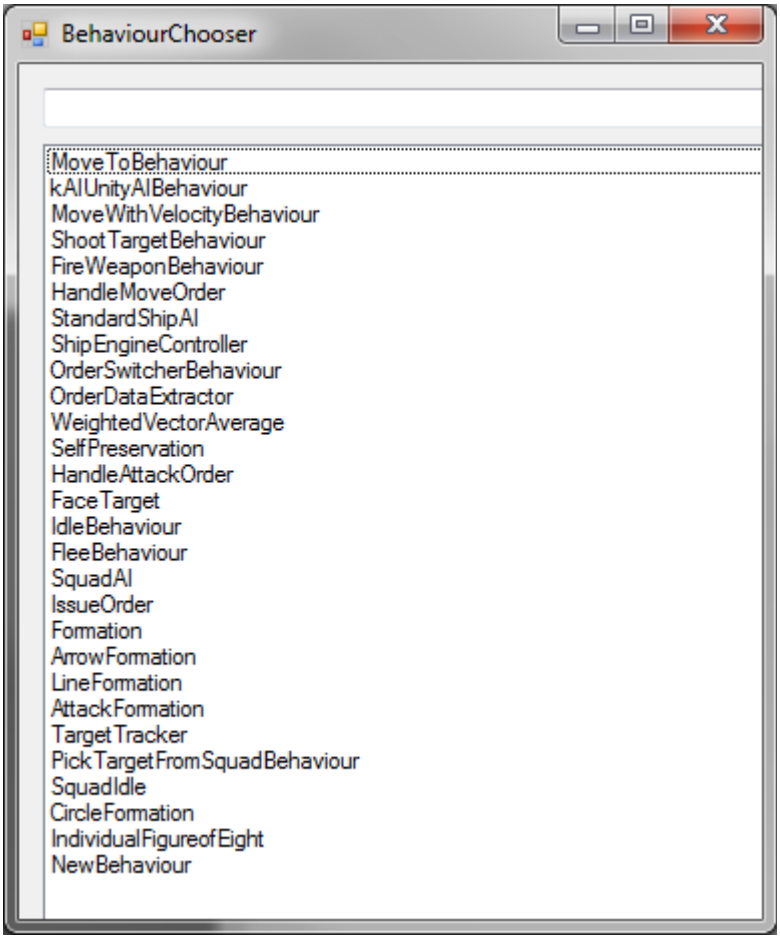


Fig 4.6: The behaviour chooser dialogue

You cannot embed kAI behaviours within themselves (this would create an infinite recursion) but any other kAI behaviour or code behaviour can be added. Double click on the behaviour to add it. This will cause the node to appear.

Adding a function node

For function nodes, you will be presented with the function node creator dialogue.

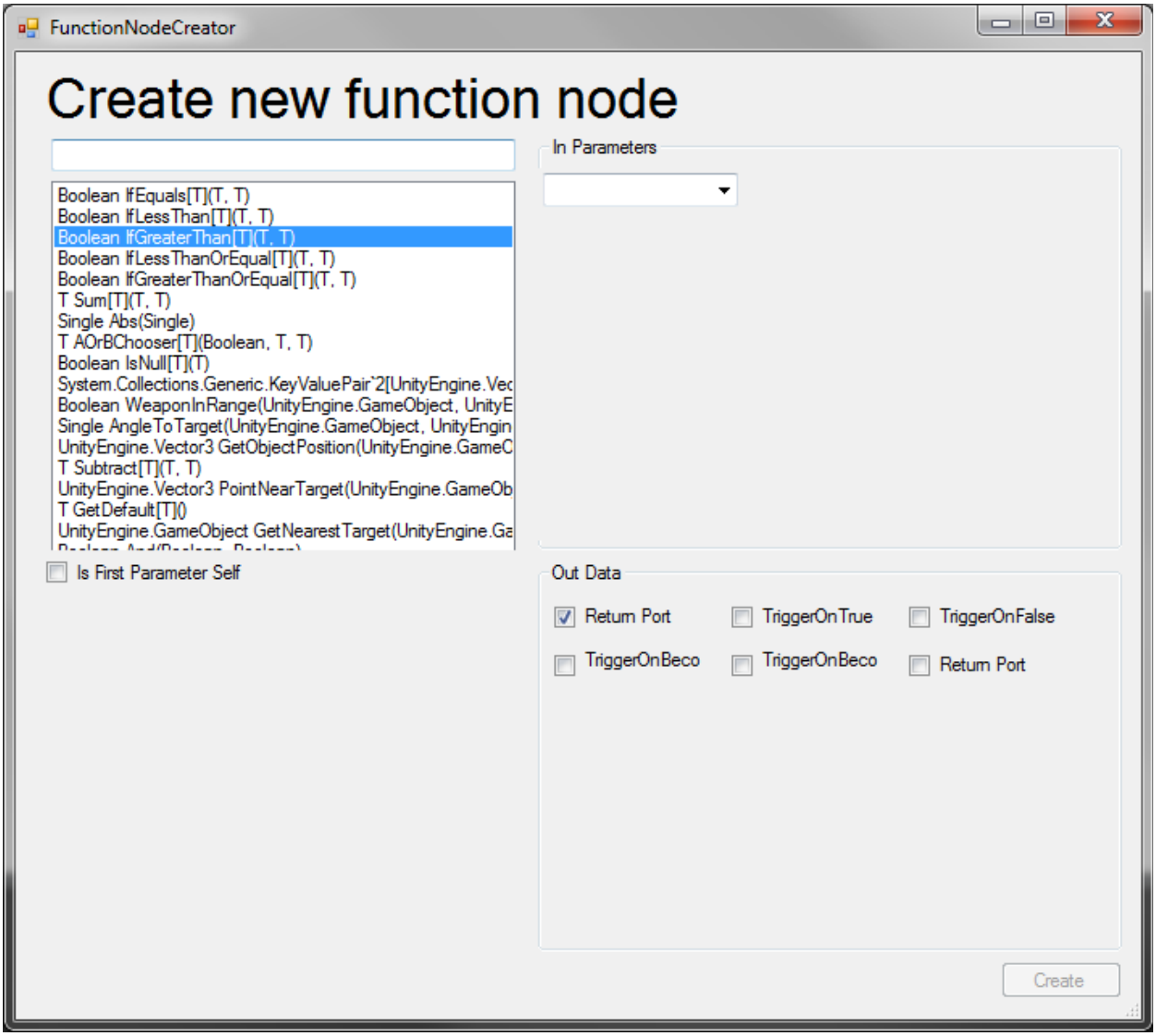


Fig 4.7: Function node creator dialogue

In the top left, you have a list of imported functions. The top right allows you to configure generic parameters. This will need to be used if the function can be used on more than one type. In these cases, you will need to specify what type you want to use it on this time. For example, a function that checks if two values are equal can be applied to any type. A drop down list will present all suitable types for the selected function.

In the bottom left, you can choose whether the first parameter of the function is the AI object. To find this out you will need to check with the coders. If the function does something with the AI itself (as opposed to some general operation like adding two numbers) then it probably does.

Finally, in the bottom right you have options for configuring the return of the function. Here there will always be a tick box labelled Return Port. If this is ticked, then whatever value the function returns will be accessible as a data port.

If the function returns a boolean, additional options are available. They allow the function node to have trigger ports depending on the value returned by the function.

Option	When triggered
`TriggerOnTrue`	Triggered every time the function returns true
`TriggerOnFalse`	Triggered every time the function returns false
`TriggerOnBecomeTrue`	Triggered on the first frame the function becomes true after previously being false. Subsequent frames where it returns true will not trigger this port.
`TriggerOnBecomeFalse`	Triggered on the first frame the function becomes false after previously being true. Subsequent frames where it returns false will not trigger this port.

Coders can provide options for other return types.

Adding a constant node

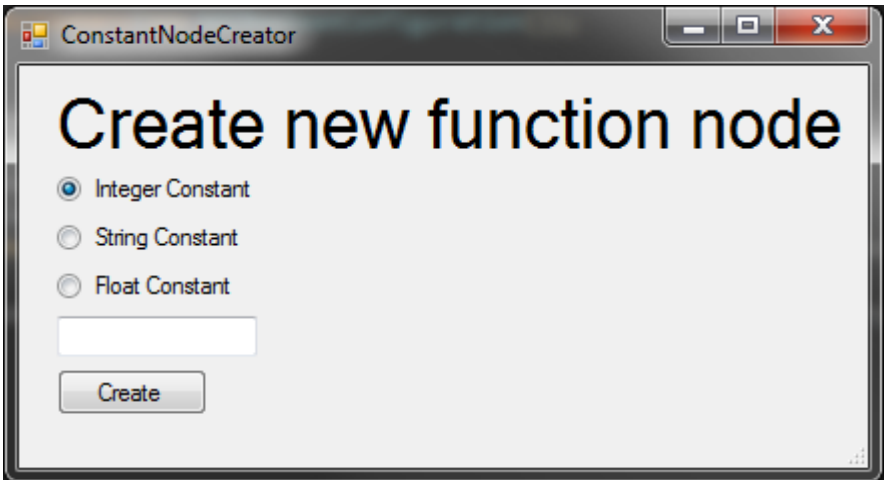


Fig 4.8: Constant node creation dialogue

Constant nodes are limited to either floating point numbers, integers or strings. The constant node editor allows you to choose from one of these types and enter to value in to the box.

4.1.3.3 Connecting nodes

When you create a node of any type, it will appear as a grey rectangle in the editor. The ports on the left and right hand side of the rectangle are the ports defined by that node. If it is a kAI behaviour, these are the ports that were defined in the editor by whoever create the behaviour. If it is a code behaviour, they are ports defined by the coders that developed the code behaviour. Function nodes will have ports for each of the parameters. Constant nodes will just have a port for the value of the constant node.

Ports are colour coded according to their type and direction.

	Inbound Port	Outbound Port
Trigger Port	Blue	Red
Data Port	Mauve	Green

In bound ports are connected to out bound ports. If they are data ports, the types must match.

To connect two ports click and drag from one to the other. A line will appear indicating that the two ports have been connected.

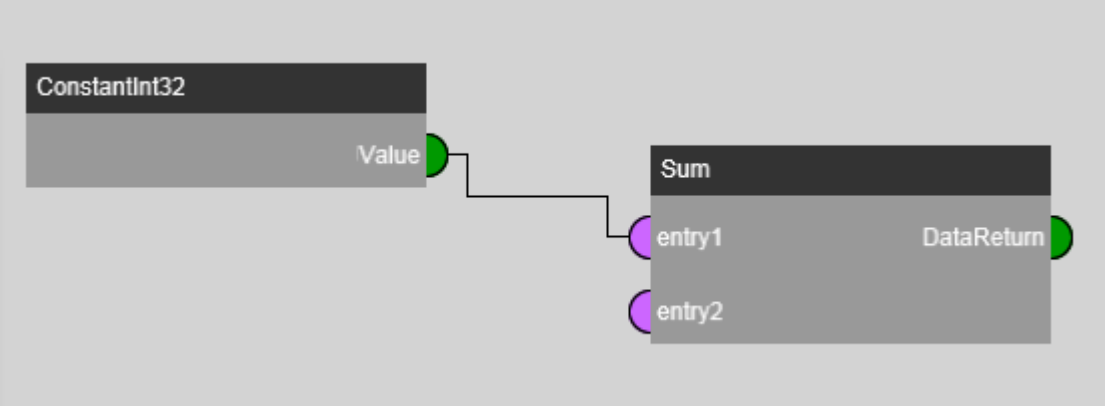


Fig 4.9: Two nodes connected together via their ports

4.1.4. Design Patterns

When designing kAI behaviours, there are some things to consider. It is important to keep the behaviours as small as possible to keep them clear and readable. This has the advantage that is easier to reuse them as well. Finally, the more modular they are, the more robust they are to outside changes. Below are some archetypal AI structures and how they can be implemented in kAI.

4.1.4.1 Finite State Machine

A finite state machine (FSM) can be used in AI behaviours where there are a set of different modes the AI can be in. These are often useful at the root of your kAI behaviour, switching between different sub kAI behaviours that control more specialist behaviour such as moving towards a target and running away from a target.

To implement a finite state machine in kAI, you must first implement your states as kAI behaviours (or code behaviours if that is appropriate). Having implemented these behaviours, add them your kAI behaviour.

Next you will need some controls to handle when to switch states in the FSM. This depends on what is triggering the state change. For example, if you are monitoring the health of the AI and switching when it falls below a certain value, you might use a function node that takes the health as a parameter and compares it against a constant node and trigger a port when it falls below a value.

In any case, you need trigger ports that are triggered when a specific state transition is required.

Then you connect these trigger ports up to the Deactivate port of the state/node you are transitioning from. Next, you connect the OnDeactivate port of that node to the Activate port of the state/node you are transitioning in to. This ensures that precisely one state is active in any given frame.

This will simulate a finite state machine.

4.1.4.2 Steering Behaviours

Steering behaviours are a method of controlling an entity which moves continuously (as opposed to on a grid) and can be controlled by a single vector pointing in the direction the entity wishes to move.

To implement this in kAI you need to first implement the steering behaviours as kAI or code behaviours. These should represent a single aspect of the AI's behaviour. For example, you could have one called Seek which represents the AI wanting to go towards a target and another called Flee which is the AI wanting to get away from another target.

Each of these behaviours should have an outbound vector data port. Inside each steering behaviour, you implement whatever is required to compute the desired vector. This desired vector is then connected to the outbound data port.

Once you've implemented all the steering behaviours, you need to assign a weight to each vector. This can be done using another behaviour that is a finite state machine that has an out bound data port for each weight of each steering behaviour.

Next you combine the corresponding vector port with a weight port in a function node that multiplies the two values. Finally, you connect all the weighted vectors in to a code behaviour which takes weighted vectors as an enumerable data port and returns the average.

The resultant vector should be fed in a code behaviour that controls how this vector is translated into actual motion.

4.1.5. Using the Debugger

The debugger is a powerful tool within the editor for working out why your kAI behaviours aren't working as expected. To use it, press the connect button in the lower left hand corner, below the list of kAI behaviours in your project.

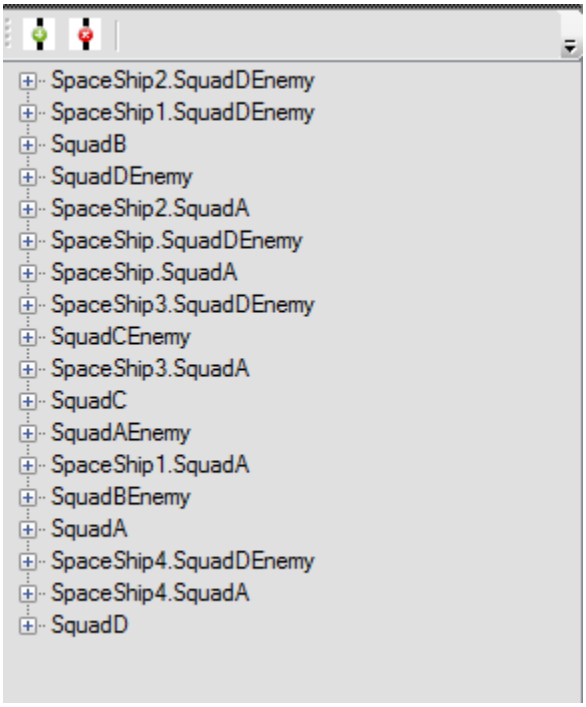


Fig 4.10: Debugger control connected to a game, the two buttons at the top allow the debugger to be connected and disconnected from the game

This will present a list of all the kAI behaviours running in the game with the debug behaviour. Clicking on one node in the tree will load up the kAI behaviour that it is simulating and show you debug information in real time.

Clicking on elements in the tool will reveal additional information in the property grid.

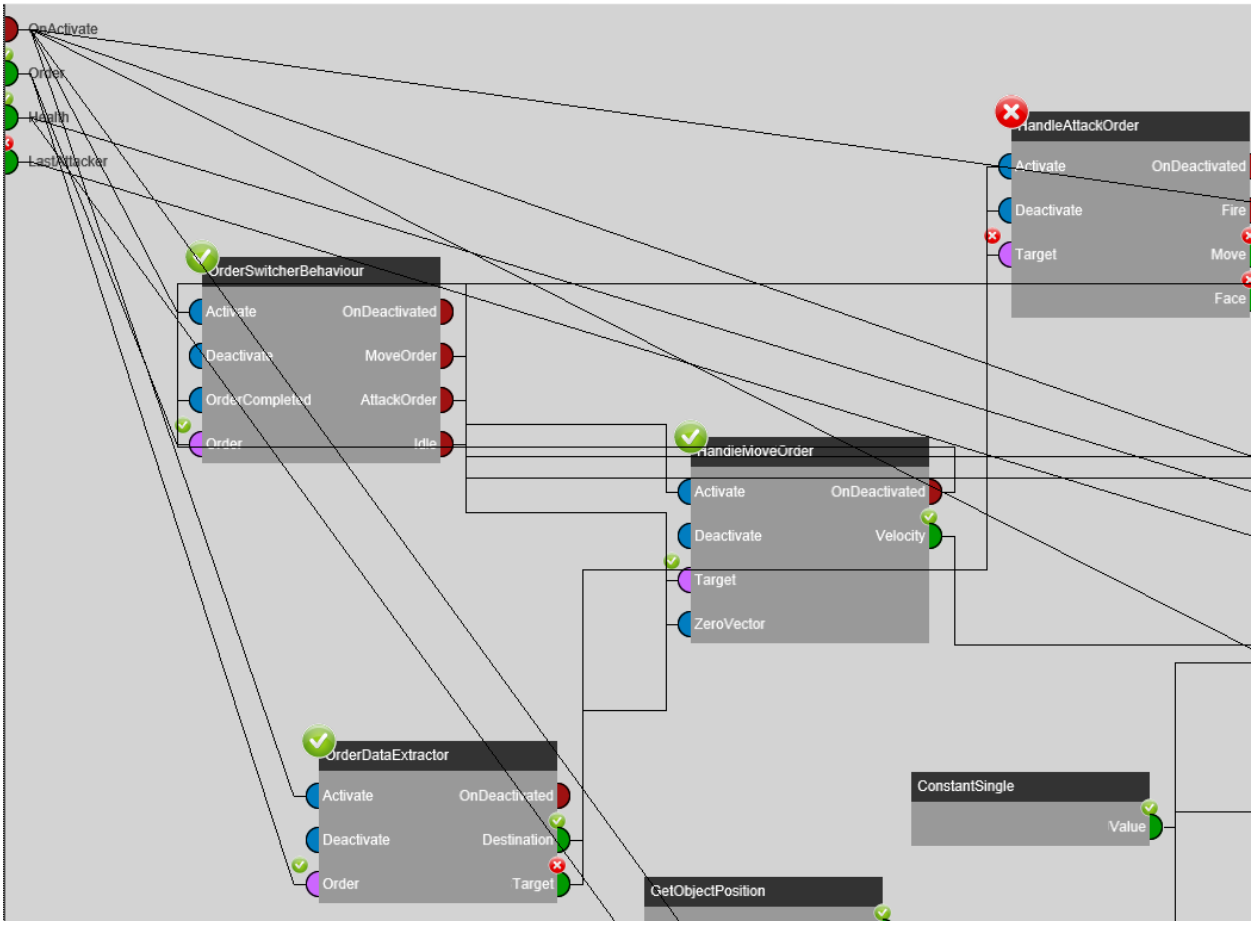


Fig 4.11: A kAI behaviour being debugged

Node Type	Visible Information	Property Grid Information
kAI Behaviour	A tick when the behaviour is active, a cross when inactive	None
Code Behaviour		Can be defined by coders when creating the code behaviour
Function node	None	None
Constant node		
Trigger port	A tick when the trigger port is triggered	When it was last triggered
Data port	A tick once the data port has a value different from the default (for example, a number data port will have a tick once the value isn't 0).	What the current value of the data port is

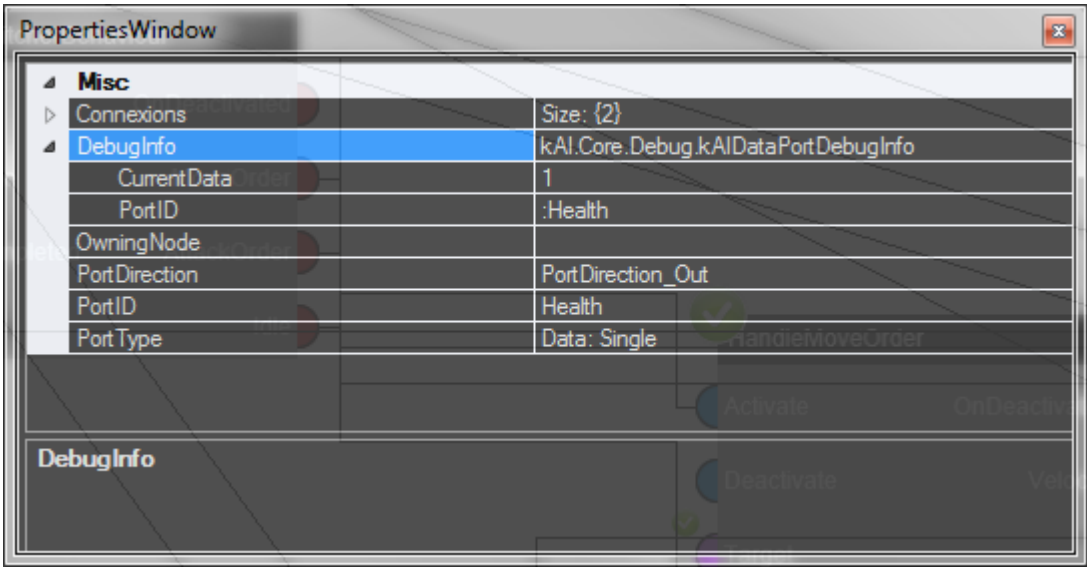


Fig 4.12: Property Grid showing the debug information of a data port

Double clicking on a kAI behaviour node within a kAI behaviour you are debugging will show that kAI behaviour and the debug information associated with it. Alternatively, you can find nested kAI behaviours within the debug window on the right.

4.2. Coders Guide

This guide is aimed at coders at the development studio. It will talk the coders through configuring kAI to work with their game engine, creating code behaviours and function nodes and connecting the editor as a debugger.

4.2.1. Configuring kAI in your game engine

4.2.1.1 Loading a kAI behaviour

There will need to be some way of attaching a kAI behaviour into AI entities in the game. The component needs to be able to be configured with a path to the kAI behaviour (the XML file created by the tool). On creation of the object, you must load the behaviour using the following method:

```
1. public static kAIXmlBehaviour LoadFromFile(FileInfo Path,
                                GetAssemblyByName AssemblyGetter)
```

Code Sample 4.1: Method deceleration for loading a kAI behaviour from a file

The `FileInfo` provides the location of the file to load.

`GetAssemblyByName` should be a method which kAI can resolve missing DLLs for code behaviours. It must match the following signature:

```
1. public Assembly GetAssemblyByName(string FullName)
```

Code Sample 4.2: Method signature for an assembly resolution method

This method must take an assembly name and return the assembly matching it.

The load method will return a `kAIXmlBehaviour`. This behaviour must be activated and informed it is the global behaviour (i.e. not nested within a kAI behaviour). This can be done via the following 2 method calls:

```
1. XmlBehaviour.SetGlobal();
2. XmlBehaviour.ForceActivation();
```

Code Sample 4.3: Method calls for setting up a kAI behaviour as the root (controlling) behaviour

The Update method must be called every frame on each AI:

```
1. public void Update(float DeltaTime, object UserData)
```

Code Sample 4.4: Method signature for the kAI behaviours update

When calling this method, `DeltaTime` should be the time since the last frame. The `UserData` should be a reference to the entity the AI is controlling. This is accessible from the code behaviours in their update and can be specified to be the first parameter of a any function nodes.

4.2.1.2 Communicating with a kAI behaviour

When you simulate a kAI behaviour, you will need to relay stimuli from the game to the behaviour. This is done via ports. Ports come in two varieties: trigger ports and data ports. Trigger ports correspond to events such as the AI being attacked. Data ports have a specific type associated with them and transmit a continuous stream of data.

When creating kAI behaviours, the designer will specify what externally accessible ports the behaviour has. These are retrieved via their unique ID using the following method:

```
1. public kAIPort GetPort(kAIPortID PortID)
```

CodeSample 4.5: Method signature for getting a port from a kAI behaviour

This will return the port with the corresponding ID. If the port is a trigger port, it can be triggered. If it is a data port it can have it's value set. It is also possible for the kAI behaviour to have ports that send data to the game (e.g. a trigger port that tells you when it has been triggered), but these are less useful. For more details on using the ports, see section 4.2.2.3 on communicating with code behaviours.

4.2.1.3 Setting up the logger

The last step of configuration is to set up the game to be the logger. This can be used to print debug information and is used by kAI to output errors and warnings.

To do this, a class implementing `kAIILogger` must be created. Then, in the initialization code, an instance of this class should be set as the global logger.

```
1. kAIObject.GlobalLogger = this;
```

Code Sample 4.6: Setting the global logger

This will then be used by all objects in kAI to print messages.

4.2.2. Creating Code Behaviours

Code behaviours provide the primitive actions that the AI entities perform. They are the nodes that designers can use in their kAI behaviours.

What a primitive action is is highly dependent on what the AI is doing in the game. For example, there could be a Move To code behaviour that handles the A* algorithm involved in moving the agents about. The designer would then be able to activate this behaviour when they want the AI to move.

In general, code behaviours should be as basic as possible, without forcing the designer to handle algorithmically complicated elements.

4.2.2.1 Setting up the behaviour class

A code behaviour is a class that inherits from `kAICodeBehaviour`. It must implement the `InternalUpdate` method:

```
1. protected abstract void InternalUpdate(float DeltaTime, object UserData);
```

Code Sample 4.7: Method signature for the update of a code behaviour

The code behaviour will be passed the time since the last frame and the user data passed in when updating the root kAI behaviour. This will be executed every frame that the node is active.

Additionally, the following methods can be overridden to handle other scenarios:

```
1. protected virtual void OnActivate()
2. protected virtual void OnDeactivate()
```

Code Sample 4.8: Method signatures for when a behaviour becomes active or inactive

These will be called when the node containing the code behaviour is activated or deactivated. These functions should not directly trigger trigger ports; see the section on triggering ports in the release phase (4.2.3.) for more information.

4.2.2.2 Deactivating a code behaviour

A code behaviour may wish to deactivate itself, for example when it has completed the action that is meant to be doing. This can simply be done by calling the following method in the `InternalUpdate`:

```
1. protected void Deactivate()
```

Code Sample 4.9: Method signature for deactivating a code behaviour

4.2.2.3 Interacting with code behaviours

As with the game interacting with kAI behaviours, designers are able to interact with code behaviours via trigger and data ports. These ports are defined by the code behaviour.

Inbound trigger ports

An inbound trigger port can be used to inform the code behaviour that something has happened. To create a trigger port you create it in the constructor of your code behaviour:

```
1. kAITriggerPort(kAIPortID PortID, ePortDirection PortDirection)
```

Code Sample 4.10: Constructor for a trigger port

The `PortID` is a string which must be unique amongst all ports (of any type) within this code behaviour. The port direction corresponds to the direction. For an in bound trigger, the direction should be

```
1. ePortDirection.PortDirection_In
```

Code Sample 4.11: Port direction for an inbound port

Next, you must add the port as an external port:

```
1. protected void AddExternalPort(kAIPort NewPort)
```

Code Sample 4.12: Method signature for adding a port to a code behaviour

Finally, you must listen to the event of when the port is triggered and write an event handler to handle what should happen when the trigger port is triggered.

```
1. public event TriggerEvent OnTriggered;
```

```
2. fireAction.OnTriggered += fireAction_OnTriggered;
```

```
3. void fireAction_OnTriggered(kAIPort lSender)
4. {
5.     shouldFireThisFrame = true;
6. }
```

Code Sample 4.13: Event signature for when a trigger port is triggered and an example event handler

It is important that trigger event handlers do not trigger other ports, see the section about triggering ports in the release phase (4.2.3) for more details.

Outbound Trigger Ports

Outbound trigger ports are used to communicate when something has happened within the code behaviour (such as reaching a destination). These are created in the same way as in bound trigger ports. Instead of listening to the `OnTriggered` event, code behaviours can instead trigger the port by calling the `Trigger` method:

```
1. public void Trigger()
```

Code Sample 4.14: Method signature for triggering a trigger port

Data Ports

Data ports are used to transmit data of a specific type. Again, they should be constructed in the constructor of the code behaviour and added to the behaviour via the `AddExternalPort` method. The generic parameter determines what data the port sends or receives.

```
1. public kAIDataPort<T>(kAIPortID PortID, ePortDirection PortDirection)
```

Code Sample 4.15: Constructor for a data port of type T

Depending on whether you are setting the data or getting the data, the `PortDirection` should be chosen appropriately. Then, you can just get or set the data whenever you require.

```
1. public T Data
2. {
3.     get; set;
4. }
```

Code Sample 4.16: Property signature used for getting and setting the data on a data port

Enumerable Data Ports

A data port can only have one data port connected to it, corresponding to the one value you get out if it. However, you may want to receive a collection of inputs without knowing in advance how many there

are going to be. This can be used to create a code behaviour that takes all of the ships in a squad and tells them each where to go according to a formation, for example.

As with other ports, to add an enumerable port, you construct it in the constructor of your code behaviour and add it to the node via the `AddExternalPort` method.

```
1. public kAIEnumerableDataPort<T>(kAIPortID lPortID)
```

Code Sample 4.17: Constructor signature for an enumerable data port

They can only be inbound so the constructor does not take a port direction. When the code behaviour is used in the editor, multiple data ports may be connected to a single enumerable data port. In your code behaviour, you will be able to access the values as an enumerable list:

```
1. public IEnumerable<T> Values
2. {
3.     get;
4. }
```

Code Sample 4.18: Property signature for accessing the list of values on an enumerable data port

4.2.3. Triggering trigger ports in the wrong phase

Although it not deterministic what order nodes will be updated within a kAI behaviour, kAI guarantees that the frame that a node becomes active or inactive is deterministic. To ensure this, you cannot trigger a trigger port in the release phase. The release phase is when actions on trigger ports are taken. Specifically, trigger ports should be be triggered in any of the following circumstances:

- In the method body of the `OnActivate` method
- In the method body of the `OnDeactivate` method
- In the event handler of another trigger port

If a trigger port needs to be triggered on one of these events, a boolean should be marked. Then in the update the relevant trigger port can be triggered. If a trigger port is triggered in one of these phases, the following exception will be thrown:

```
1. TriggeredPortInReleasePhaseException
```

Code Sample 4.19: Exception for when a trigger port is triggered in the wrong phase

4.2.4. Creating Functions for Function Nodes

Function nodes provide simple, stateless, utilities for designers to use in their behaviours. For example, they can be used to perform mathematical calculations or for querying the scene.

To create such a function, create a public static method that is within one of the DLLs imported into the editor.

4.2.4.1 Parameters

Each parameter for the function is represented as a data port on the function node. Parameters cannot be `ref` or `out` parameters. There are some special cases for parameters below.

Self Parameters

If your function performs an operation on or with the AI, the first parameter can be specified as "self". This means the object passed in as the user data when updating the kAI behaviour can be specified to be the first parameter of the function. This is a tick box in the editor that needs to be checked when creating a function node from a function that uses this.

Enumerable parameters

If a function is defined with one or more parameters of type `IEnumerable<T>` for some T, kAI will construct enumerable data ports for these parameters. As with code behaviours with enumerable ports, designers can connect multiple data ports of type T to these and they will be combined in to a list for the function. This can be used to create a function which computes the average of a set of numbers.

Generic Parameters

kAI supports generic parameters for functions. However, when the generic parameters are used, it must be used as the parameter, not as a nested parameter in a generic type. This means, the following method signature is not supported.

```
1. public static bool ExampleFun<T>(Tuple<T, T> param)
```

Code Sample 4.20: Example method for a generic parameter that is not supported in kAI

This is because the T is nested within another type.

Generic Constraints

In addition to C#'s interface constraint system, kAI supports an additional constraint system for operators. It allows the developer of the method to require that a generic parameter implements a mathematical operator.

If you need this restriction, you can define your method as follows:

```
1. [StaticConstraint(StaticConstraint.StaticConstraintType.eConstraint_Minus)]
2. public static T Subtract<T>(T entry1, T entry2)
```

Code Sample 4.21: Example method signature that specifies a static constraint on the generic parameter

This method will require that T implements the minus operator.

4.2.4.2 Method Returns

Functions that return a value will have a data port of the corresponding type.

Boolean returning functions

Functions which return a boolean value have additional options when creating the function node. These options allow the function to trigger ports when the returned value becomes true, for example.

Extending functions that don't return boolean

If you have types which you can provide general options for all functions, (as kAI provides for functions that return boolean) you can define a set of properties for this type. The properties must be of the following form:

- Be either on or off
- Define at most one new port the function node should have (either a trigger or data port)
- Can be evaluated using only the current and previous values that the function evaluated to

To create properties for type T, you must extend `kAIReturnConfigurationDictionary<T>` with generic parameter T. Then you must add your properties using the `AddProperty` method to configure your properties.

You must give them a name, a lambda function which creates your port (if required, otherwise just return null) and a lambda function that takes the node (`kAINodeObject`), the current value of the function (T) and the value of the function last time it was evaluated (T).

For example, the boolean property class adds the following property:

```
1. const string kPortID = "OnTrue";
2. Func<kAIPort> lOnTruePortCreate = () =>
3. {
4.   return new kAITriggerPort(kPortID, kAIPort.ePortDirection.PortDirection_Out);
5. };
6.
7. Action<kAINodeObject, bool, bool> lOnTrueAction =
8.   (lNodeObject, lResult, lOldResult) =>
9.   {
10.      if (lResult)
11.      {
12.          kAITriggerPort lActualOnTrue = lNodeObject.GetPort(kPortID);
13.
14.          lActualOnTrue.Trigger();
15.      }
16.};
17.AddProperty("TriggerOnTrue", lOnTruePortCreate, lOnTrueAction);
```

Code Sample 4.22: Setting up the "OnTrue" property for boolean returning functions

On lines 2-5, a function that returns a port is defined. It returns the port that should be created, if the property is enabled.

On lines 7-16, the action that should be performed when the function is evaluated is defined.

Finally, on line 17 the property is added to the dictionary, giving it a name.

Simply defining the class will cause it to be added in the editor. To make the function properties work in the game, you need to add the class to the dictionary using the following function:

```
1. Vector3Propertes x = new Vector3Propertes();
2. kAIReturnConfigurationDictionary.AddDictionary(x);
```

Code Sample 4.23: Adding a dictionary of properties to the set of properties

4.2.5. Using Code Behaviours in the editor

To use the code behaviours and functions that have been created, the DLL must be imported into the tool. To do this, after creating the project,go into project properties (Project>Project Properties) and go to the DLLs tab.

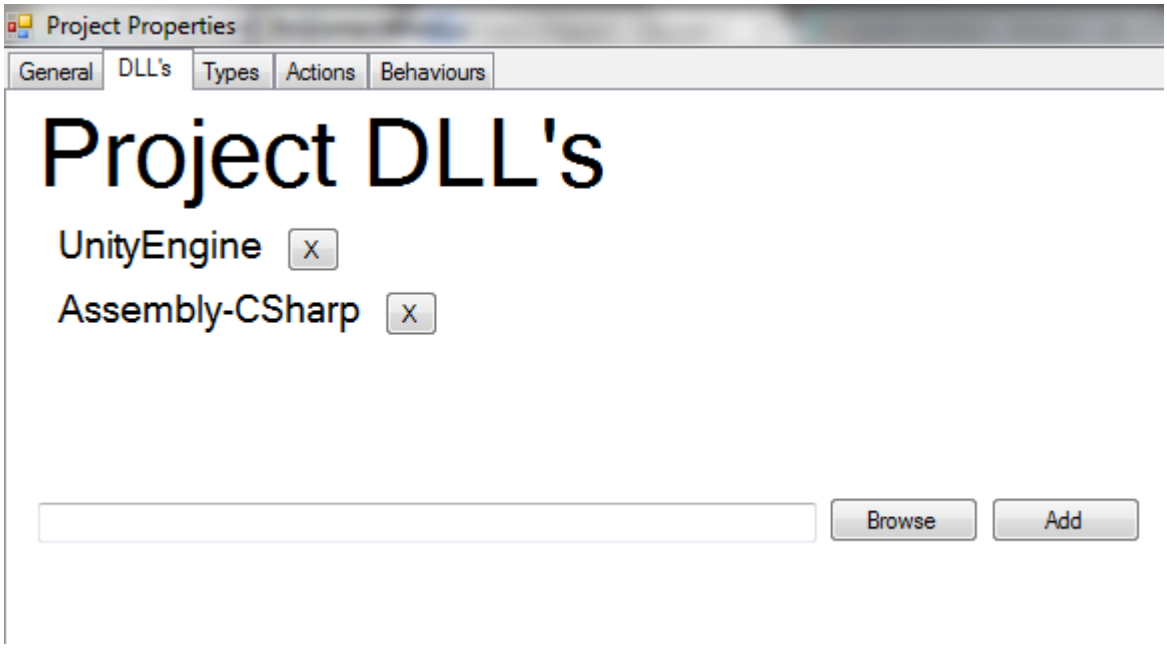


Fig 4.13: Project DLLs section of the project properties. For loading DLLs

Select browse to locate your DLL and then click add to add it the project. If it depends on other DLLs, you will be requested to locate these as well.

4.2.6. Debugger

The editor can be used to debug the kAI behaviours in real time. It will show what behaviours are active and the values of data ports. Code behaviours can be configured to provide useful debug information to designers also.

4.2.6.1 Connecting the debugger

The first step is to initialize the debug server in your game.

```
1. kAIDebugServer.Init(string ProcessID);
```

Code Sample 4.24: Command for initialising the debug server in the game

The ID will be used by the editor to connect to a specific instance of your game.

Next, for each component in your game, you must register it with the debug server:

```
1. public static kAIBehaviourDebugStore AddBehaviour(kAIXmlBehaviour Behaviour,
                                                     string objectID)
```

Code Sample 4.25: Method signature for registering a new kAI behaviour with the debug server

This will register the behaviour with the server using the ID provided. The ID should be unique amongst all AI behaviours being registered. The debug store returned represents the debug file for this entity. This should be updated each frame to ensure it has the latest debug info:

```
1. public void Update()
```

Code Sample 4.26: Method signature for updating a debug store

Remark: If you have a lot of entities, you can stagger the updates of entities to avoid hitting the frame rate when debugging.

To connect the editor, press the connect button in the bottom right hand window. The editor will then present a list of entities active within the game. Selecting one of these entities will load the relevant kAI behaviour in the editor with real time debug information.

4.2.6.2 Extending code behaviours debug information

The code behaviours can be configured to present more debug information to the designers. To do this, you must extend the `kAIBehaviourDebugInfo` class with your own class. The class must have the `[Serializable]` attribute and consist of public properties that should be visible in the property grid.

For this class to be used, in your code behaviour, the following method must be overloaded to return your debug information.

```
1. public abstract Debug.kAINodeObjectDebugInfo GenerateDebugInfo();
```

Code Sample 4.27: Method signature for creating debug info for a code behaviour

Remark: This method is called every frame that a kAI behaviour using this code behaviour has its debug information updated. As a result, if the creation of your debug information is costly and doesn't change much, it may be worth trying to avoid recreating it each frame.

When the designer clicks on a code behaviour node, this information will be displayed in the property grid.

5. Project Management

I used an agile methodology[50] in my development of kAI.

5.1. Stories

I used AgileZen to manage the stories that represented features of the project. Each story consisted of some tasks that would need to be completed to finish the story. At the beginning of the project, I put the stories in to a timetable of which sprint they would be completed in. I then adjusted both the stories and the timetable based on progress in the sprint reviews.

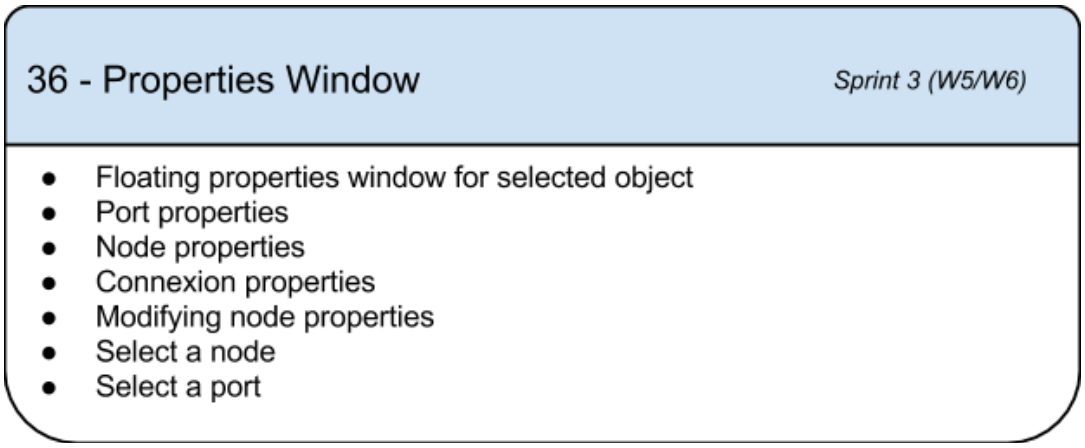


Fig 5.1: An example story card

5.2. Sprints

I divided the project development time in to 2 week sprints. Initially, I spread the stories out between these periods. At the end of each sprint I would discuss with my project supervisor which stories I had completed and what stories I should focus on in the next sprint. If there were issues we would discuss how important these were and, if necessary, plan a one week sprint with a very specific focus. The development timetable at various stages are viewable on the CD in appendix E.

5.3. Dailies

I did “rubber duck dailies” at the start of each day. This is a adaptation of the idea of rubber duck debugging[51]where I would record in a document what I did yesterday and what I planned to do today. I also used this document to keep notes in the sprint reviews. Below is a sample daily, the complete log can be found on the CD in appendix D.

18/10/2013

Yesterday: Implemented disconnecting ports (both individually and the whole lot). The DLL stuff is working and I don't think there is any point adding more functionality to it (eg sorting through different types) until we have data ports. As a result, I have modified the story to reflect that and pushed it into the next sprint.

Today: Going to resolve the issues I found with the update yesterday, specifically my checking we are not in a pass phase does not apply for bound ports. Going to implement a non-console logger with a permanent bar for entering commands. At that point, I am going to review where I am midway through this sprint and work out what I am doing for the next week (as quite ahead of where I need to be for this sprint).

Fig 5.2: Example daily

5.4. Coding Standard and Documentation

As this was a large software development project, I adopted some standards to maintain conformity and readability across the code base.

I used a reduced version of Hungarian Notion[52]. Local variables started with an `l`, member variables with an `m`, constants with a `k` and statics with an `s`. This helps to keep track of where variables are declared.

For the core of kAI I compiled with warnings as errors. This forces proactive refactoring of code. For example, it won't let you have unused variables.

I used Microsoft's C# code documentation system, which can be used to generate documentation for code. It looks like this:

```
1. /// <summary>
2. /// Update this behaviour (if active).
3. /// </summary>
4. /// <param name="lDeltaTime">The time in seconds that has passed since the last
   frame. </param>
5. /// <param name="lUserData">The user data. </param>
6. public override sealed void Update(float lDeltaTime, object lUserData)
```

Code Sample 5.1: Comment function signature

To generate the documentation, I used a tool called Sandcastle[53]. This produces a series of HTML pages for each of the classes and methods in the project. I enabled failure to complete these tags a warning, again forcing me to document as I went. The generated documentation can be seen on the disk in appendix F.

Note: to keep the code samples readable in this report, I have excluded the scope prefixes on some variable names and made other adjustments when I don't feel it is helping understanding. See Appendix Bfor details regarding the samples.

5.5. Source Control

I used Git as my source control solution. I made use of Github's free hosting for open source projects to store my project[54]. When submitting code changes, I had a system for marking them depending on what they contained. This allows for filtering to find specific changes.

The system was as follows:

```
[CORE|EDIT|EXAMPLE|GROUND][#|!|+|-] {Description of the change}
```

Where the first tag indicates what part of the project it is related to:

- `CORE` – the central library
- `EDIT` – the editor
- `EXAMPLE` – the simple example game
- `GROUND` – the more complex example game

The second tag represented the type of change:

- `+`: new feature/element
- `-`: removing redundant code/feature
- `#`: change to existing feature
- `!`: fixing a bug

5.6. Problems and Delays

The project underwent some delays due to unforeseen problems. I had developed a basic version of the editor in June 2013 using WinForms to render the behaviours. However, there were performance issues even with simple behaviours. This was due to the overhead in moving a single control in WinForms is large.

To resolve this, one of the first tasks I undertook in October was rendering the behaviours in DirectX. As I did not have very much experience in DirectX, this took a longer to get working than I had hoped.

The other main delay I had was with the second example I tried to make. One of the key aspects of kAI is the ability for coders to hide complexity from designers in the form of code behaviours.

To demonstrate this, I had planned to make an inertia based flight model in my strategy game. Then the code behaviours would handle operations like moving towards a destination. These are non-trivial operations in a frictionless physics driven environment, involving complex mathematical formulae for computing the forces required. As a result, just to get to the basic level needed to start work on the AI took too long. Me and my project supervisor agreed it would be more beneficial to write a simpler game – one without the inertia flight model – and spend more time on the AI.

6. Further Development

There are a lot of directions that could be explored if this project were to be taken further. I have outlined a few here.

6.1. Core

These are the areas I think the core of kAI could be extended.

6.1.1. Behaviour Interfaces

When coders are interfacing with kAI behaviours, for example to trigger relevant ports at relevant times and transmit data in to data ports, they must access these ports via their unique ID. However, since this is a string, the entry of this is prone to errors. For example, the coder might think the data port is called “CharHealth” but the behaviour calls it “CharacterHealth”.

A feature could be implemented to resolve this which would work like programming interfaces. These would be written by the coders and then imported into the tool. They would specify what ports the behaviour needs to have and what types they are. Then, the designers, when creating a kAI behaviour, could pick a set of interfaces that their behaviour should conform to. The tool would then ensure the required ports are present in the behaviour.

This would reduce errors due to name mismatches and would allow for greater separation between the designers tasks and the coders tasks.

6.1.2. Scripting

Another powerful feature that kAI could be extended to support would be a scripting system. This would allow more technical designers to attach scripts (chunks of C# code) on to trigger ports and have it executed when triggered.

The primary use could be for printing out additional information but could also be used to do other things such as manipulating the code behaviours in a way that doesn’t naturally fit in to the port driven interaction system.

6.2. Debugger

6.2.1. Breakpoints

Breakpoints are a standard feature of modern debuggers. Breakpoints pause the execution of the program when it hits a certain line of code. This could be implemented into the kAI debugger with breakpoints being associated with trigger ports and would pause the game when certain trigger ports are triggered.

6.2.2. Interactive Debugging

At the moment, the debugger allows the editor to see the debug information and display it in real time. An extension would be to allow the editor to influence the game. A simple example would be allowing trigger ports to be triggered in the editor. This way, if there was a game event that was rare, the designers could still test their AI easily by manually triggering the port. Alternatively, the values in data ports could be set in the editor, again to test out different behaviours.

6.2.3. Step by Step Debugging

One powerful feature of most advanced debuggers is the ability to step through code line by line. The editor could be extended to step through the game frame by frame. This could be taken further, allowing for step by step update of the kAI behaviours themselves. This would allow designers to see what the trigger ports are triggered and when they become active.

7. Legal, Cultural and Ethical Issues

7.1. Questionnaire

I asked a number of game developers to answer a questionnaire to find out if there is an industry need for such an AI tool. However, since the people who filled out the questionnaire were from within my own social circles, no deception or coercion was used, no substances were administered and no invasive procedures were involved , approval from the BSREC for the questionnaire was not required.

7.2. Source code, libraries and tools used

To speed up the development, I used a number of libraries to help with low level issues. Here is a full list of the libraries used and the licenses associated with them:

7.2.1. DirectX

DirectX was used to render the behaviour. It is closed source but it is freely usable and re-distributable under the Microsoft Public License.[55]

7.2.2. SlimDX

SlimDX is a managed wrapper around DirectX allowing it to be used in C#. It is released open source under the MIT license which allows modification, redistribution and selling without restriction.[56]

7.2.3. SprintTextRenderer

This is a small library to assist in rendering of sprints (2D images) and text. This is also released under the MIT License which allows modification, redistribution and selling without restriction. [57]

7.2.4. MiscUtil

This is a utility library for C#. I have used it to provide operation calling on generic types for function nodes. It is released under the Apache License. This allows modification, redistribution and selling providing you include an acknowledgement to the source.[58]

7.2.5. FileMap

This a library for using the Win3D API calls for creating memory mapped files. It is released under the GNU lesser public license which grants permission to copy, modify and redistribute the library.[59]

7.2.6. ThreadMessaging library

A general purpose library for interprocess communication. I use its implementation of the semaphore object. This is released under a modified BSD license. This allows redistribution of the library providing the copyright notice is retained. [60]

7.2.7. Unity3D

Unit3D is a closed source library and tool and is used under the Unity Software Licences Agreement. This specifies extensive conditions, limiting the use of it on sales over $100,000 etc. However, for free open source projects, it can be used to publish and distribute the game. [61]

7.2.8. Glee

The graph library, which wasn't used in the end but is still within the source code, was developed by Microsoft and released under their Microsoft Research Shared Source License agreement. It allows for any copying, modification and redistribution on all non-commercial projects. [62]

7.2.9. Sandcastle

Sandcastle is a Microsoft plug-in for Visual Studio which allows for the automatic generation of code documentation. It is usable under the Microsoft Public License which allows free usage of it. [63]

8. Conclusion

In writing this tool I have developed a deeper understanding of some of the unique challenges involved in AI game development. I believe this tool provides a powerful, open-ended framework for developing AI's for games, whilst maintaining a structure to help designers avoid common programming pitfalls.

It is clear from the questionnaire that there is an industry need for tools that help develop AI. I received overwhelmingly positive reaction to the proposed tool.

The update algorithm implemented in kAI, which prevents loops and non-deterministic behaviour, is an area of kAI I am particularly pleased with. This feature differentiates kAI from other more general visual programming tools such as Blueprint. It makes the tool much more robust for designers to use. Further, the guarantee of determinism is an important quality when ensuring that a game is bug free.

Moreover, kAI maintains flexibility through its implementation of code behaviours and function nodes, allowing coders to configure the nodes to support any kind of AI.

Finally, I am pleased with the real time debugger of the behaviours. It proved useful both in debugging my own behaviours, and demonstrating how the kAI behaviours worked. The ability to see what a behaviour is doing whist the game is still running gives the kAI debugger an advantage over traditional AI development methods.

If I had further development time, I would like to have demonstrated more complex AI. In particular, I would have liked to create a strategic AI in kAI.

Overall, I feel that kAI has addressed a real problem in game AI development. With further work, I believe kAI could help studios develop more compelling AI's for their games.

9. Citations

1. Ichiro Otobe, Innovations in the video game industry, http://asia.stanford.edu/events/fall07/slides/otobe.pdf (accessed 07/04/2014).

2. Brian Ashcraft, http://kotaku.com/5260424/assassins-creed-iis-team-has-tripled, http://kotaku.com/5260424/assassins-creed-iis-team-has-tripled (accessed 17/04/2014).

3. Sony, Playstation 4 Technical Specification, http://us.playstation.com/ps4/features/techspecs/ (accessed 17/04/2014).

4. David M. Bourg and Glenn , AI for Game Developers, 2004

5. Creative Skillset, Job Roles in Computer Games, http://creativeskillset.org/creative_industries/games/job_roles (accessed 27/04/2014).

6. Scott Rogers, Level Up, 2010

7. P. Hart, N. Nilsson and B. Raphael , A Formal Basis for the Heuristic Determination of Minimum Cost Paths, 1968

8. Nareyek, AI in Computer Games, 2004

9. Fu, Daniel, and Ryan Houlette, Putting AI in entertainment: An AI authoring tool for simulation and games., 2002

10. Different Methods, React, https://www.assetstore.unity3d.com/#/content/516 (accessed 18/04/2014).

11. James Golding, Blueprint, https://www.unrealengine.com/blog/blueprint-basics (accessed 18/04/2014).

12. Rogers, Scott, Level Up, 2010

13. Bourg, Seeman, AI for Game Developers, 2004

14. Born Ready Games, Strike Suit Zero, http://strikesuitzero.com/ (accessed 27/04/2014).

15. Reynolds, Craig, Steering Behaviors For Autonomous Characters, 1999

16. Fairclough, Chris, et al., Research directions for AI in computer games, 2001

17. Ponsen, Improving adaptive game AI with evolutionary learning, 2004

18. Millington, Ian, and John Funge, Artificial intelligence for games, 2009

19. Smedberg, Niklas, Wright, Rendering Techniques in Gears of War 2, *Game Developers Conference*, 2009

20. Unreal, Flow Control, https://docs.unrealengine.com/latest/INT/Engine/Blueprints/UserGuide/FlowControl/ (accessed 29/04/2014).
21. Different Methods, React, https://www.assetstore.unity3d.com/#/content/516 (accessed 27/04/2014).
22. Björn Knafla, Introduction to Behavior Trees, http://www.altdevblogaday.com/2011/02/24/introduction-to-behavior-trees/ (accessed 19/04/2014).
23. Microsoft, Visual C#, http://msdn.microsoft.com/en-us/library/kx37x362.aspx (accessed 19/04/2014).
24. 22, What is a DLL, http://support.microsoft.com/kb/815065 (accessed 19/04/2014).
25. Bourg, Seeman, AI for Game Developers, 2004
26. Microsoft, MethodInfo Class, http://msdn.microsoft.com/en-us/library/system.reflection.methodinfo(v=vs.110).aspx (accessed 19/04/2014).
27. Microsoft, Generic Methods, http://msdn.microsoft.com/en-us/library/twcad0zb.aspx (accessed 19/04/2014).
28. Microsoft, Constraints on Type Parameters, http://msdn.microsoft.com/en-us/library/d5x73970.aspx (accessed 19/04/2014).
29. Microsoft, Solution for overloaded operator constraint in .NET generics, http://stackoverflow.com/questions/147646/solution-for-overloaded-operator-constraint-in-net-generics/147656#147656 (accessed 19/04/2014).
30. Microsoft, Introduction to Attributes, http://msdn.microsoft.com/en-us/library/aa288059(v=vs.71).aspx (accessed 19/04/2014).
31. Skeet, Gravell, MiscUtil, http://www.yoda.arachsys.com/csharp/miscutil/ (accessed 29/04/2014).
32. Aho, Hopcroft, Ullman, Data Structures and Algorithms, 1983
33. Microsoft, Data Contract Serializer, http://msdn.microsoft.com/en-us/library/ms731072 (accessed 25/04/2014).
34. Microsoft, Windows Forms, http://msdn.microsoft.com/en-us/library/dd30h2yb(v=vs.110).aspx (accessed 19/04/2014).
35. SlimDX, SlimDX, http://slimdx.org/ (accessed 28/04/2014).
36. Schertler, SlimDX SpriteTextRenderer, https://sdxspritetext.codeplex.com/ (accessed 28/04/2014).
37. Malenfant, Jacques, Jacques, and Demers, A Tutorial on Behavioral Reflection and its Implementation, 1996

38. Microsoft, Reflection in C# and Visual Basic, http://msdn.microsoft.com/en-us/library/ms173183.aspx (accessed 25/04/2014).
39. Microsoft, Windows Forms, http://msdn.microsoft.com/en-us/library/dd30h2yb (accessed 29/04/2014).
40. Microsoft, Developing games, http://msdn.microsoft.com/en-us/library/windows/apps/hh452744.aspx (accessed 29/04/2014).
41. The Economist, The meaning of Xbox, 2005
42. Microsoft, Microsoft Automatic Graph Layout, http://research.microsoft.com/en-us/projects/msagl/ (accessed 29/04/2014).
43. SlimDX, SlimDX, http://slimdx.org/ (accessed 29/04/2014).
44. Schertler, SlimDX SpirteTextRenderer, https://sdxspritetext.codeplex.com/ (accessed 29/04/2014).
45. Guttman, R-Trees: A Dynamic Index Structure for Spatial Searching, 1984
46. Microsoft, How to: Use Anonymous Pipes for Local Interprocess Communication, http://msdn.microsoft.com/en-us/library/bb546102.aspx (accessed 28/04/2014).
47. Microsoft, Memory-Mapped Files, http://msdn.microsoft.com/en-us/library/dd997372(v=vs.110).aspx (accessed 19/04/2014).
48. Dijkstra, Edsger, Cooperating sequential processes, 1965
49. Unity, Unity, http://unity3d.com/unity (accessed 19/04/2014).
50. Beck, Kent; et al., Manifesto for Agile Software Development, 2001
51. Hunt, Thomas, The Pragmatic Programmer: From Journeyman to Master, 1999
52. Simonyi, Hungarian Notation, 1999
53. Microsoft, Woodruff , Sandcastle, The generated documentation can be seen on the disk in appendix F. (accessed 28/04/2014).
54. Github, Github, http://github.com/ (accessed 29/04/2014).
55. Microsoft, Microsoft Public License, http://archive.msdn.microsoft.com/mag201310DXF/Project/License.aspx (accessed 21/04/2014).
56. SlimDX, SlimDX License, http://slimdx.org/license.php (accessed 21/04/2014).
57. Nico Schertler, MIT license for SDX, http://sdxspritetext.codeplex.com/license (accessed 21/04/2014).
58. Skeet, Gravell, "Miscellaneous Utility Library" Software Licence, http://www.yoda.arachsys.com/csharp/miscutil/licence.txt (accessed 21/04/2014).

59. Restrepo, GNU Lesser Public License, https://github.com/tomasr/filemap/blob/master/License.txt (accessed 21/04/2014).
60. Ruegg, ThreadMessaging.NET License, https://p-gm-front.aws-codeproject.com/system/cp/53063d8e702d676ed7040000/raw/55858ce0e92b401b4c7923bdce185c74d1316521/threadmsg_src/ThreadMessaging.1.1-src/License.txt (accessed 21/04/2014).
61. Unity3D, Unity Software License Agreement 4.x, https://unity3d.com/company/legal/eula (accessed 21/04/2014).
62. Microsoft, Microsoft Research Shared Source License Agreement, http://research.microsoft.com/en-us/downloads/f1303e46-965f-401a-87c3-34e1331d32c5/GLEEEula.rtf (accessed 21/04/2014).
63. Microsoft, Sandcastle License, https://sandcastle.codeplex.com/license (accessed 28/04/2014).

10. Appendices

Appendix A - Using the CD

The CD contains the full source code for kAI, a built version of the editor and built versions of the games. There is also a copy of this report and further material (listed in Appendix C). The editor is Windows only, but there are Linux builds for the two example games.

The file structure is as follows:

- /Appendices – The accompanying material for this report (see appendix C for a list)
- /Builds – Pre-compiled versions of the editor and games
- /src – All the source code for kAI
    - /FileMap – Library for FileMap code
    - /GroundExample – Source code for the strategy game example
    - /kAICore – Source code for the core of kAI
        - /Debug – Source code for the debug API
    - /kAIDocs – project for generating source code
        - /hmtl – The generated documentation
    - /kAI-Editor – Source code for the editor
    - /kAI-Example – Source code for the sword duel example
    - /SpaceExample – Source code for the abandoned example with the inertia flight model
    - /SpriteTextRenderer – Source code for the SDX library used in the editor
    - /ThreadMessaging – Source code for the semaphores library that was used
    - /Tools – Installers for compiling kAI

Inserting the disk into Windows should cause the auto-run to launch. If it does not, it can be run by double-clicking on Autorun.exe. From here, you can select to launch one of the example projects in the editor or run the example games. As the CD drive is read-only, you will need to copy the contents of the CD on to the computer to edit the projects.

To see the debugger in action, run one of the games and launch the accompanying kAI project. Once the game is running, click the connect debugger icon (green connect button in the lower left hand panel in

the editor). Selecting a node from the debug panel will show you the information in real time for that entity.

Appendix B - Code Samples

Code samples are used in this report to illustrate key areas of technical importance. in some instances, in the interest of keeping the samples brief and clear, the samples have been modified from the original source. Below is a complete list of the code samples in this report.

Appendix C - Digital Appendices List

The following appendices are included on the disk in the Appendices folder.

- Appendix D – Dailies and sprint reviews
  - A full copy of all of the rubber duck dailies and sprint reviews taken during this project
- Appendix E – Timetables
  - Copy of the timetable at various points through out the project
- Appendix F – Generated documentation

- The generated documentation for this project

- Appendix G – Specification
  - The original specification for the project
- Appendix H – Presentation
  - The presentation given at the end of the pro

Appendix I – Survey responses

- Spreadsheet of all the responses to the survey