Documentation RoboCode behaviour tree

Casper van Battum | 3031249 | Kernmodule Game Development 3

www.github.com/Creator13/HKU-KMOD3

This documentation explains the functioning of the behaviour tree controlling the RoboCode bot 'Peter.'

Structure

The behaviour tree is a fairly simple concept and requires few classes by itself. Any behaviour tree is just a collection of nodes (BTNode) linked together. The three different types of nodes used in this project are the Composite, the Decorator, and the Action. Composites will run all their children nodes in a set order, Decorators alter the result of each node's Run function, and finally the magic happens in the Actions. These nodes actually control the robot and as such they are not reusable in other projects. The other two categories are reusable. In total, the project contains fourteen Action nodes.

The three different base types all have a default constructor with a parameter common to all nodes of that type. A composite node must always have a list of children, a decorator must always have one child node to decorate and an action must always have a blackboard with data to act upon. Each implementation of these abstract classes must override the default constructor. In some cases, especially true for action nodes, additional parameters can be added to the constructor to provide additional settings or data for that node to work with. Although C# requires you to override the constructor in every subclass, which means more work, this system allows for quite easy extension.

This project only used two types of composite nodes: the Selector and the Sequence. The Selector runs each of its children until one succeeds, while the Sequence runs all nodes until one fails. Other type of composites, like a random selector, were not needed.

In addition, also just two decorator nodes were needed: Invert and AlwaysSuccess. Invert turns success into failure and failure into success, while AlwaysSuccess will return success even if the node it decorates did not.

Action nodes

There are fourteen different action nodes in this project. Some are pretty straightforward, like Move, Turn or Fire. Others are more complicated, meant to carry out a very specific task or to test a condition.

Condition nodes

Sometimes the bot has to do a certain action based on whether a condition is true or not. A condition node can check whether this condition is true and return success if it is. A node like TargetStill will simply check whether the scanned target is not moving, but a node like EnergyConditional can actually check for a user-defined condition through a predicate. The node asks for a predicate with one input value (the robot's energy in this case), allowing the programmer to decide exactly at what energy values the node should return true. This is a very modular system.

Special turn and move nodes

The bot needed to be able to move randomly, or to move directly towards (or away from) the target. The plain move node wasn't enough for this, so it had to be extended using extra functionality. This was done by creating a wrapper node. Take for example the MoveRandom node. This node generates a random number first, then creates a new Move node with the randomly generated number and runs it. This is different from a decorator node in that the result of the child node is (generally) not changed, yet the input value is. As it relies on data from the blackboard, these nodes are simply other subclasses of Action. By modifying other nodes like this, actions can be extended in functionality without sacrificing any modularity.

Behaviour

The assignment was to make a behaviour tree with three distinct branches. This behaviour tree decides which branch to run based on the energy of the robot. The three main branches are called the targeting tree, the evasive tree and the distancing tree. When the robot's energy is above 80 it will run the targeting tree, when it is between 80 and 20 it will run the evasive tree, and when it falls below 20 it uses the distancing tree. Each of these trees are a sequence guarded by an EnergyConditional node. The first energy condition that succeeds will be ran.

Range checking

To achieve good behaviour, it was necessary to work with ranges. Is the robot in range to fire? Should it move closer or further away? The range class specifies a minimal distance, maximal distance and a target distance. The latter is the distance the robot should move aim to reach as soon as it is out of range.

Target tree

The target tree is meant to search the target, move towards it and fire at it if it is close enough. To avoid unnecessary moving, it first checks whether it is in range to fire. If it is it simply fires and goes to the next cycle. If it's not, it will move. This very basic behaviour is effective against targets that are moving within a close radius, but it leaves the bot vulnerable for attacks at it doesn't move much by itself.

Evasive tree

The evasive tree is based on random movement. The theory is that if you keep moving erratically, you won't get hit very easily. In this tree, the bot will not move towards a target. However when the target gets close enough, based on the same firing range as in the evasive tree, a bullet is fired at max power. Since the bot is hard to hit, it is not very vulnerable for attacks. The erratic movement and turning can however cause the bot to hit the wall often.

Distancing tree

The distancing tree tries to keep as much distance from the other bot as possible. It defines a range with a high minimal value, causing the bot to move away from the target if it gets too close. By constantly moving in a square and by being relatively far away from the other bot, it is not very vulnerable from attacks. This tree uses the same firing mechanic as in the evasive tree, which will fire only if the target happens to get close.

Firing when target is not moving

Not all bots are perfect and they can get stuck, stop in their tracks or leave themselves otherwise vulnerable for attacks by coming to a standstill. As soon as a bot is detected to not move, a bullet is fired at it at once, not matter the distance. A selector that checks whether the target is either standing still or in range of the bot decides whether to fire a bullet. This simple added behaviour proved very effective. This behaviour is currently implemented in every subtree separately, but it could even be a separate subtree which runs before all others. This ensures that a bullet will always be fired if the target bot is not moving.

Conclusion

The biggest challenge in this project was learning to work with RoboCode. It's not a very modern and sleek API, which I found slightly confusing at times. Building a behaviour tree for it doesn't feel like the easiest or even most effective way to program a robot. As the behaviour tree is very simple in its core, I didn't find much challenge in this assignment.

One thing I found interesting to figure out was setting up an efficient development environment. By creating a run configuration in Rider which started RoboCode using a command, I was able to test my robot by simply pressing the play button in my IDE. This saved me a lot of time, but it was also fun to figure out. The process of setting up a C# project outside of Unity was also an educational new experience.

PDFs with the UML diagram and the behaviour tree can also be found in the Github project (www.github.com/Creator13/HKU-KMOD3).