Stealth AI

By Lewis Arnold

Contents

[Behaviour Tree 2](#_Toc38035457)

[The Guards 3](#_Toc38035458)

[The Spy 4](#_Toc38035459)

[Diagrams & Examples 5](#_Toc38035460)

[Guard Behaviour Tree 5](#_Toc38035461)

[Spy Behaviour Tree 6](#_Toc38035462)

[Editor Window 7](#_Toc38035463)

[Bibliography 8](#_Toc38035464)

# Behaviour Tree

The guards make use of a behaviour trees which is a tree of hierarchical nodes that control the flow of decision making (Simpson, 2014). Knowing this, I set to work creating what would be a generic base node for a behaviour tree, consisting of nothing more than a virtual function to Tick the node, as well as a virtual Test function for debugging purposes and a quality of life function called GetPlayerPosition which will return the player’s position for use within the tree. The decision to make this function arose from needing a global blackboard or information for all my nodes to access, but since this is the only global data I needed, a function in the base node sufficed.

This was built on to create a selector, sequencer and behaviour node. As (Simpson, 2014) says in his article, these nodes will enable you to create quite complex behaviours. As I started adding behaviours, I realised that was true. I didn’t need to add any more node types to get the behaviours I wanted.

Most certainly the most impactful decision I made was to make the behaviour tree in a namespace. This is something very new to me and I see it being used in professional environments, for example “using UnityEngine.AI” or “using System.IO”, and thought I’d try my hand at it for no other reason than curiosity. Having now used namespaces, I can see their utility for bolting functionality into other projects and it has proven very useful. Another big decision was to make my base node class not inherit from monobehaviour. This was because I wanted to be able to expand this in the future and make a tree from an external file. This would involve using calls like “BT\_Sequencer seq = new BT\_Seqeuencer()” which is not possible with classes inheriting from monobehaviour. This was all good until I realised that since they do not inherit from monobehaviour, they can not access the local transform of the object they are controlling. This means passing a reference to the local transform to every behaviour in instantiation. I did not find a solution for this as I couldn’t get the nodes to work while inheriting from monobehaviour.

One giant advantage of behaviour trees which I have found is their ease of use when following behaviour. State machines get messy and can be hard to follow but a behaviour tree is much better laid out for following logical flow; making testing the trees much simpler. The only thing stopping me in my current situation was that I had no way of visualizing the tree and what logic is flowing. This lead me to do some research on unity editor scripting and to create a tool for my behaviour trees which allows the run-time viewing of the trees as well as each node’s state and what they do.

This showed me an issue I had that the editor script (as well as some behaviours) would be trying to pull data from one specific tree, the Guard Tree. This was discovered as I kept getting errors as soon as I implemented my Spy’s behaviour tree. This lead me to create a base class for local blackboards called LocalTree that held a tree and some common, needed functionality.

# The Guards

The guards in this game make use of the behaviour tree described above. The guards’ behaviour tree also makes use of a local blackboard, which was vital in getting the tree to work. The decision to use a local blackboard came about while adding the ability to go to a last known location. I did not keep reference to each individual node added into the tree because it could theoretically be generated from an external file and there be no issues. This meant that in order to store the last known location, I needed one centralised point for accessing data shared between the nodes. The same script where the tree was created was treated as a local blackboard for the nodes to use. With this in place, it was possible to create some more advanced behaviours where one node can influence the outcome of another by changing values in the local blackboard. For example, both the canSeePlayer and canHearPlayer nodes change the LastKnownLocation variable in the local blackboard, which then allows the guard to seek out the player after they have lost sight / just heard the player.

Adding this new, complex behaviour also created another issue: monolith classes. After receiving feedback on the product, it was made clear that the behaviour nodes needed to be more generic and tend towards a “one class, one responsibility” ideal. (Sutter, 2009) makes a guideline to stick to this rule where possible, and a behaviour tree is no exception. Before this realisation, the left-most sequencer with the canSeePlayer behaviour used to be just one behaviour of chasePlayer. This was then split into the canSeePlayer and moveToLocation behaviours which at first felt like a lot of extra work for just more classes. However, this led to more easily building the tree later. When I added the ability to sleep when tired and get coffee when lethargic, the moveToLocation behaviour combined with the local blackboard made this an easy task, with little new behaviour code having to be written.

When I first thought about how I would implement sounds and hearing into the guard’s behaviour, I obviously thought of using a trigger sphere and basing their hearing off this. However, (Brown, 2020) makes good reasoning that more realistically predictable hearing models can be attained from using the pathfinding system that’s already being used. This works by calculating a path from the source of the sound to all nearby guards. If those guards are within the range of the sounds (the total length of the path is less than a value for how loud the sound is) then they will be made aware. This creates a much better model for hearing sounds as guards cannot hear the player through walls, but rather around them, which makes a lot more sense. Another method would be one discussed by (Vekhala & Jack, 2013) who talk about using a heatmap of suspicious activity which the guards can use to determine where they should look when alerted. The heatmap is given value in the area around where the player has made some noise

As for how the guards see the player, this is done with a simple raycast from the guard to the player. If the ray hits the player, the player has been spotted. Other stealth games use multiple raycasts to different bones (Walsh, 2014) to simulate the player being in different levels of cover. However, my project does not utilise cover, so I decided it best for one simple raycast be made from guard to player to simulate a line of sight.

# The Spy

My spy implementation was very easy to implement. With the behaviour tree already built and some experience building a tree under my belt, it was only a matter of pulling apart my original one-script state machine into separate behaviour scripts to be added to a new tree.

One behaviour I had to decide on with the spy is what they do when detected, which is run to the nearest hiding spot. They calculate this much in the same way that hearing is calculated. They use the pathfinding already in the game to calculate lengths to different paths. I chose to use this method over a birds-eye method as it makes the spy that little bit more believable since instead of running through a maze to get to a hiding spot the other side of the wall it is next to, it will instead move round the corner to the actual closest spot.

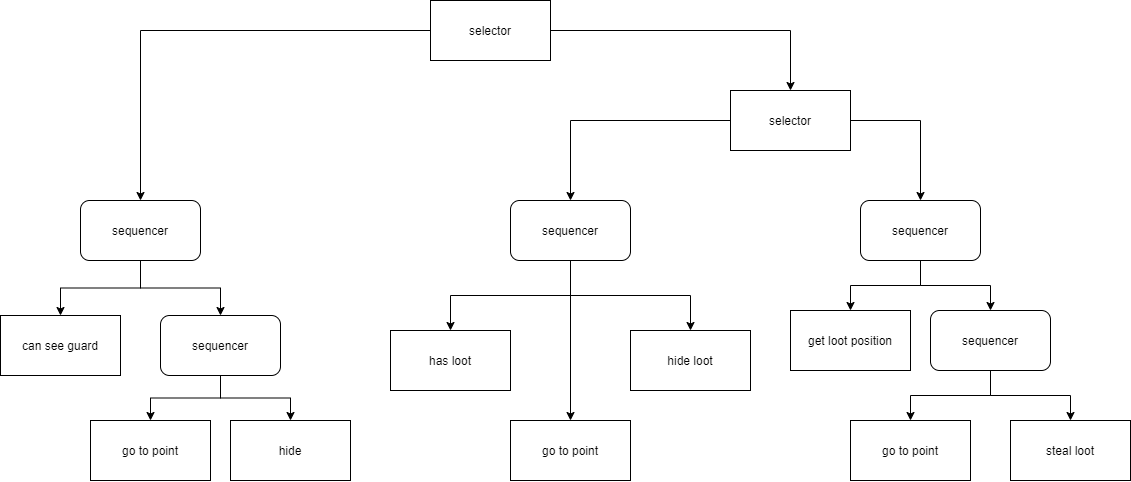
Some values that determine the spy’s behaviour are randomized on initialization. This is to get different scenarios every time the game plays. The values I randomize are the time to crack the safe, in an attempt to simulate different people having different skill levels. I also randomize the time the spy will hide in cover for. From having played stealth games with friends, it is clear that some people will hide considerably longer than they really have to.

# Diagrams & Examples

## Guard Behaviour Tree

A visual representation of the behaviour tree used by the guard

## Spy Behaviour Tree



A visual representation of the behaviour tree used by the spy

## Editor Window

A view of the editor window used for debugging the behaviour trees

# Bibliography

Brown, M. (2020, March 25). *How Stealth Game Guards See and Hear | School of Stealth*. Retrieved from Youtube: https://www.youtube.com/watch?v=Ay-5g36oFfc

Simpson, C. (2014, July 17). *Behavior trees for AI: How they work*. Retrieved from Gamasutra: https://www.gamasutra.com/blogs/ChrisSimpson/20140717/221339/Behavior\_trees\_for\_AI\_How\_they\_work.php

Sutter, H. (2009). Retrieved from GotW: http://www.gotw.ca/gotw/084.htm

Vekhala, M., & Jack, M. (2013). *Spaces in the Sandbox: Tactical Awareness in Open World Games*. Retrieved from GDCVault: https://www.gdcvault.com/play/1018038/Spaces-in-the-Sandbox-Tactical

Walsh, M. (2014). *Modeling AI Perception and Awareness in Splinter Cell: Blacklist*. Retrieved from GDC Vault: https://www.gdcvault.com/play/1020436/Modeling-AI-Perception-and-Awareness