# Table of Contents

# Deathmatch AI Kit

Before getting started, the Deathmatch AI Kit requires the following assets:

- Behavior Designer
- Behavior Designer – Movement Pack
- Ultimate Character Controller, UFPS, or Third Person Controller
- Behavior Designer / Character Controller integration

For the Behavior Designer / Character Controller integration, enter your Behavior Designer invoice number at the top of the downloads page, then click Download next to the "Ultimate Character Controller".


Ensure these four packages are imported before the Deathmatch AI Kit. If you forget to import one of these packages before importing the Deathmatch AI Kit there's a chance that the behavior tree will be reserialized with a missing task reference. In this case you should reimport the Deathmatch AI Kit again.

The Deathmatch AI Kit has the following folder structure:

- *Opsive/DeathmatchAIKit/BehaviorTrees*
  Contains the behavior trees for the AI agents. Includes two high level behavior trees and multiple subtrees. The Solo behavior tree should be used by agents who do not operate on a team. The Team behavior tree should be used by agents who operate on a team and want to create formations or call for backup.
- *Opsive/DeathmatchAIKit/Demo*
  Contains all of the assets necessary for the demo scene.
- *Opsive/DeathmatchAIKit/Editor*
  Contains the editor scripts for the AI logic components.
- *Opsive/DeathmatchAIKit/Scripts*
  Contains the logic for the AI agents.

If you are only wanting to use the behavior trees within your own game you can remove the Demo folder. The AI scripts do not depend on anything within this folder.

After the Deathmatch AI Kit has been imported, the first step in getting started with the Deathmatch AI Kit is to create your AI agent. The Agent Manager can be used to make this process as easy as possible. From there you can start to modify the behavior tree to fit your game. The Behavior Tree overview page is a great resource for learning why the agents operate the way that they do.

# Getting Started

## Importing

Before you import the Deathmatch AI Kit ensure that you have first imported:

- Behavior Designer
- Behavior Designer – Movement Pack
- Ultimate Character Controller, First Person Controller, or Third Person Controller
- Behavior Designer / Character Controller integration

For the Behavior Designer / Character Controller integration, enter your Behavior Designer invoice number at the top of the downloads page, then click Download next to the "Ultimate Character Controller".

**If you have an existing project we recommend testing the demo scene in a new project**. This will ensure the layers from the demo scene do not overlap with your existing layers. The following inputs are added:

- Scoreboard
- Toggle Weapon Wheel
- End Game

Along with the the following layers:

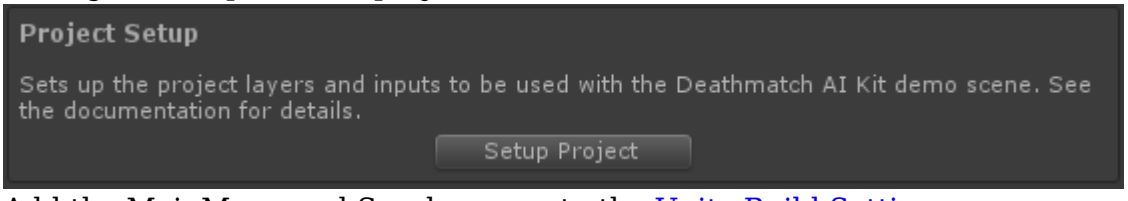
- Layer 8: RedTeam
- Layer 9: GreenTeam
- Layer 10: BlueTeam
- Layer 11: YellowTeam
- Layer 12: Ragdoll

We expect most Deathmatch AI Kit users are interested in the AI rather than creating an exact deathmatch scene replica. For this you do not need to import into a new project.

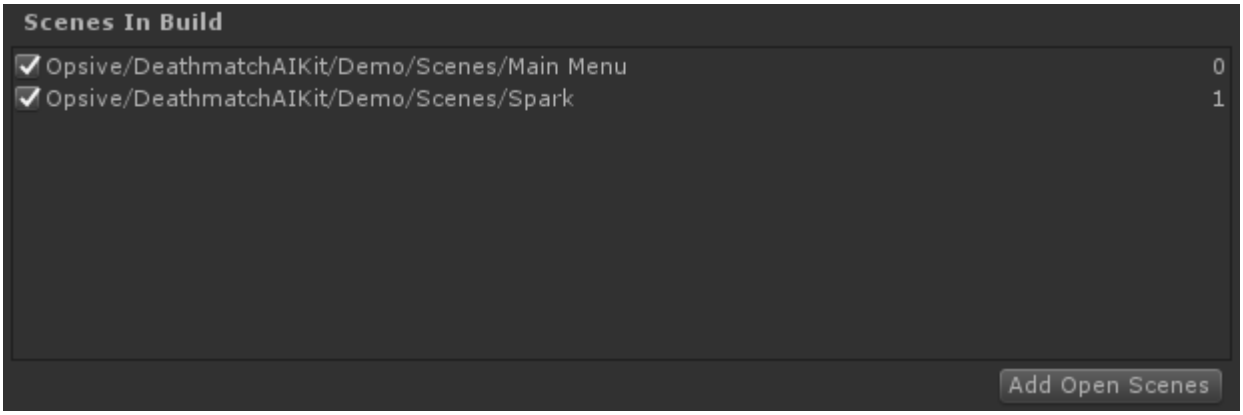
## Demo Scene

The Deathmatch AI Kit contains two scenes: a main menu and the actual demo scene. The Ultimate Character Controller demo assets are required by these demo scenes. There are a few steps required before being able to play the demo scene:

1. **Open the Main Menu and Spark scenes and save the files.** An editor script will automatically run ensuring all of the correct references are setup.
2. Open the Setup Manager from Tools -> Opsive -> Deathmatch AI Kit -> Setup Manager and update the project.




3. Add the MainMenu and Spark scenes to the Unity Build Settings.

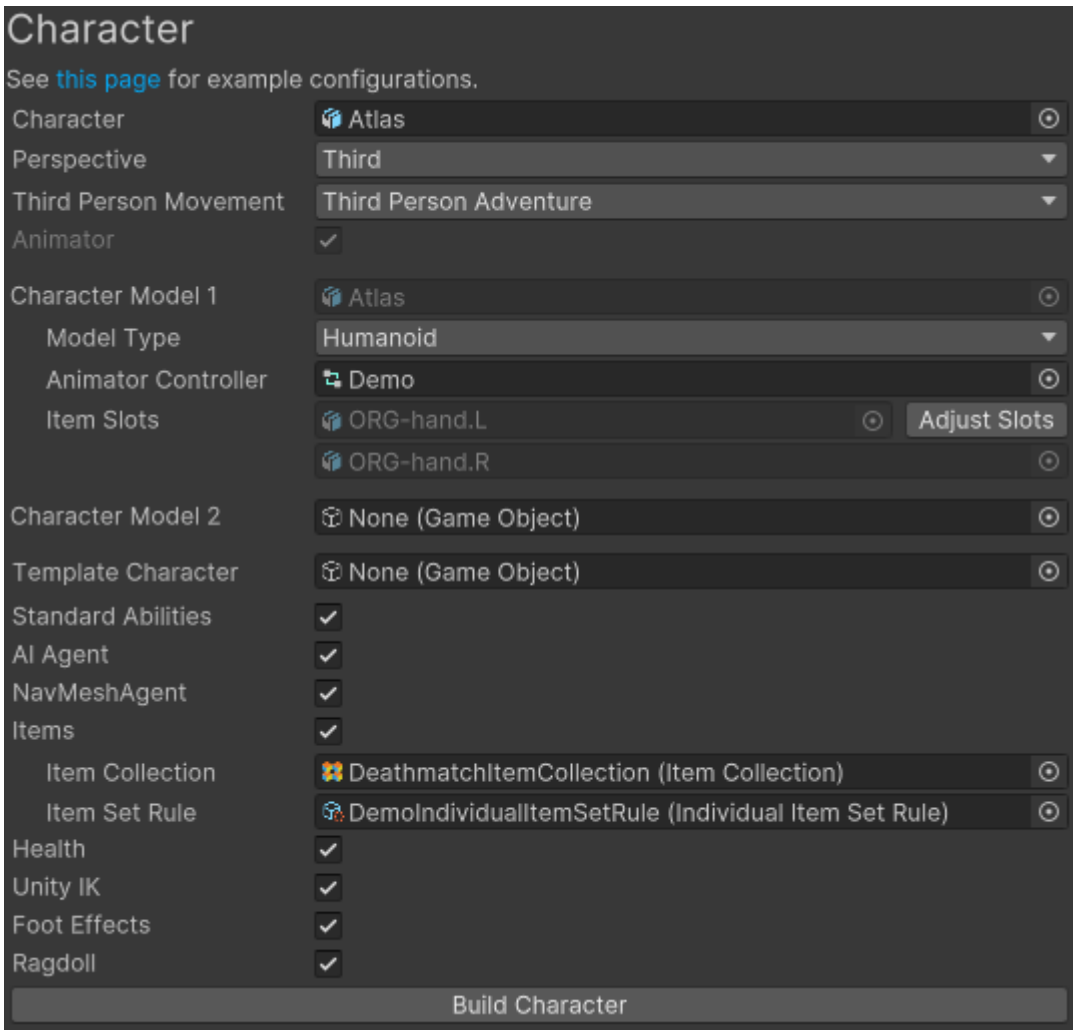



### Lightmapping

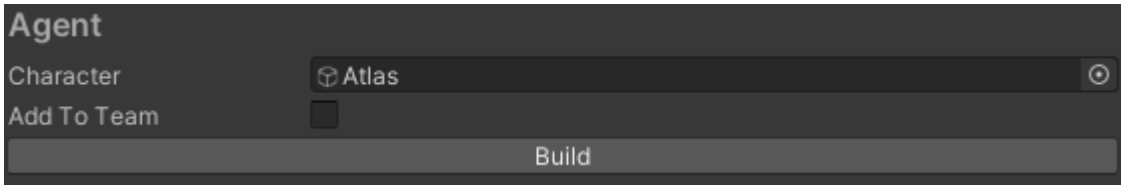
In order to reduce the download size the demo scene does not contain any lightmapping. When the scene first opens it will appear dark and this can be corrected by baking the scene. If the lights are not baked before hitting play they will automatically be disabled.

## Agent Manager

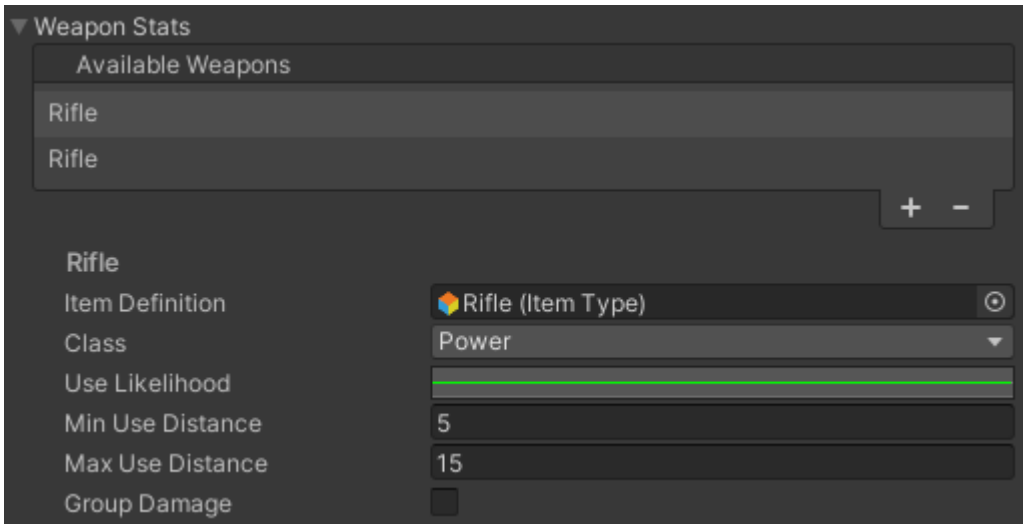
The Agent Manager makes setting up a new AI Agent extremely quick. Before running the Agent Manager the character must first be setup using the Character Manager.



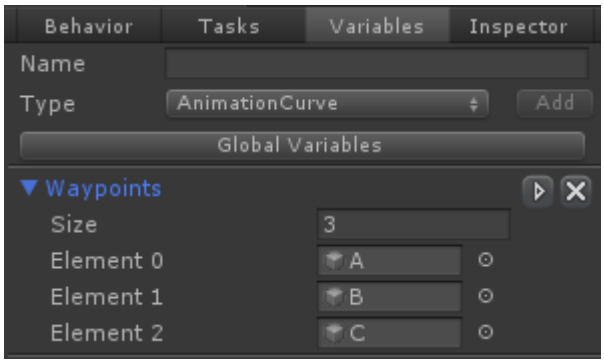

After creating the character you can then use that character within the Agent Manager.




If Team is selected then the Agent Builder will add the Team behavior tree to your agent, otherwise it will add the Solo tree. If your character already has items added to it then the Agent Builder will automatically add references for those items within the Deathmatch Agent component. However, the Agent Builder does not know the weapon properties (class, use likelihood, etc.) of that item so this needs to be changed after the agent has been built. The item will also need to be manually added to the list if you later add the item to the character.




After you have completed this setup you'll be able to hit play and the agent will attack if it sees an enemy. With this setup the agent will just stand in their current location because there are no waypoints setup. You can add waypoints by adding to the Waypoints variable within the Variables tab of Behavior Designer:




Your agent is now setup! The next step is to start to customize the AI for your game. Start with the Behavior Tree Overview page to get a sense for how the tree works.

## Behavior Tree Overview

The purpose of this section is to give an overview of how the Deathmatch AI Kit behavior tree works. In addition to the Behavior Designer documentation, the following pages contain

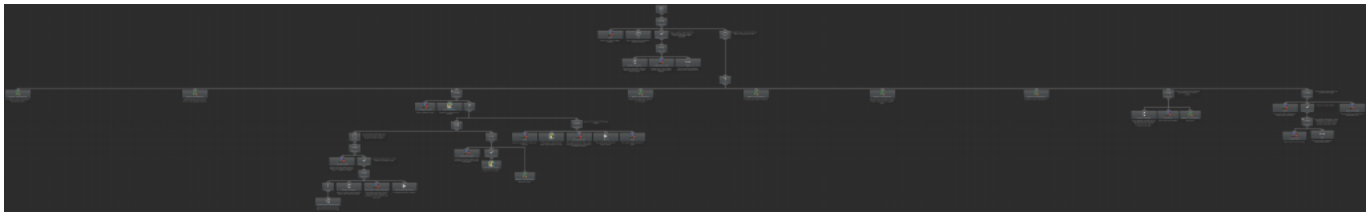
more resources for how behavior trees work:

- Gamasutra – Behavior trees for AI: How they work
- Opsive Character Controller Integration Explanation
- Awesome AI made Easy with Behavior Designer
- Practical Guide to AI in Unity
- Recreating the FALSE KNIGHT Boss Fight

The Deathmatch AI Kit behavior trees are saved using External Behavior Trees. External behavior trees are used in this situation because they allow you to dynamically assign the behavior tree at runtime. In the case of the Deathmatch AI Kit, at the start of the game either the Solo or Team behavior tree is used. These trees are extremely similar, except the Team behavior tree contains tasks specific to operating on a team (such as requesting backup). The Solo and Team behavior trees then use the Behavior Tree Reference task to load subtrees at runtime. This allows us to reference the same subtree within both the Solo and Team behavior tree. For the first part of this overview we will only be focusing on the Solo behavior tree. The reason for this is that the Solo behavior tree is a subset of the Team behavior tree.

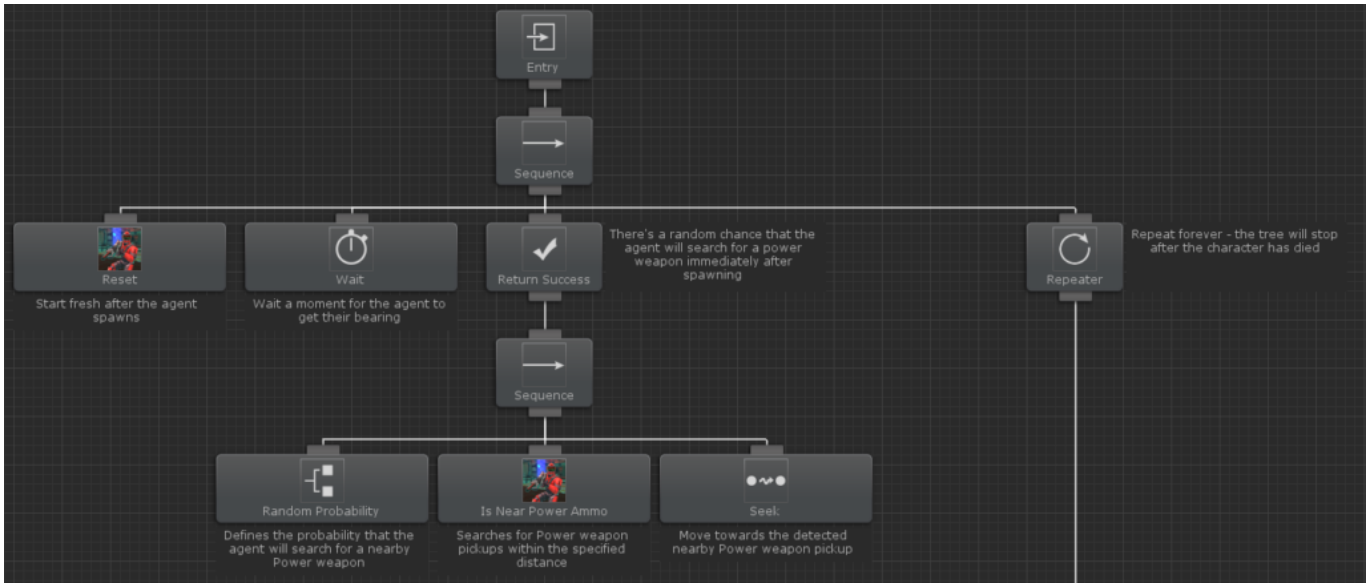
## Solo Tree

When you open the Solo behavior tree within Behavior Designer you'll see (click to enlarge):



At this point the game is not running so the Behavior Tree Reference tasks are still visible within the tree. As soon as you hit play the tree will look like the following:

This tree contains a lot of tasks so for this section we'll be taking it one step at a time, going through the tree in the same order that the behavior tree is evaluated. The first section that we are going to look at is the following:




Behavior trees are evaluated from top to bottom, left to right. The very first task that runs within the tree is the Sequence task. The Sequence task will run its first child task, the Reset task. Reset is used to reset all of the SharedVariables back to their default values. On first run the variables do not need to be reset but when the agent dies and respawns the behavior tree will start again from the beginning. It is here that we want to make sure all of the variables have a fresh start.

Immediately after the Reset task is the Wait task. The Wait task will delay the tree from executing for a very short amount of time which mimics the agent getting their bearing. The next child task that runs is the Random Probability conditional task. When the agent first spawns they may want to search for a power weapon (the rocket launcher). This random probability will introduce a randomness in determining if the agent should search for a power weapon. If Random Probability returns Success the Is Near Power Ammo task runs. If the agent is far away from a power weapon they should not search for it so this task performs that check. If Is Near Power Ammo returns Success then the Seek task will run. This task will seek towards the ammo position so the agent can pick up the power weapon.

In this child branch there were two conditional tasks which could return failure: Random Probability and Is Near Power Ammo. If either of these tasks return Failure then the Sequence task is also going to return Failure. The Return Success decorator is used to keep the tree running even if one of the child tasks return Failure.

The next task that runs is the Repeater. The behavior tree should continue to execute even if any of the child branches complete (successfully or unsuccessfully). The Repeater allows us to do just that. For example, if the agent kills the target then the attack branch will return Success. If there was no Repeater the behavior tree execution would then be over and the agent would just stand there. We don't want this to happen so the Repeater will allow the agent to start searching for a new target.

If we look at the tree from a zoomed out view we'll see that it has the following structure:

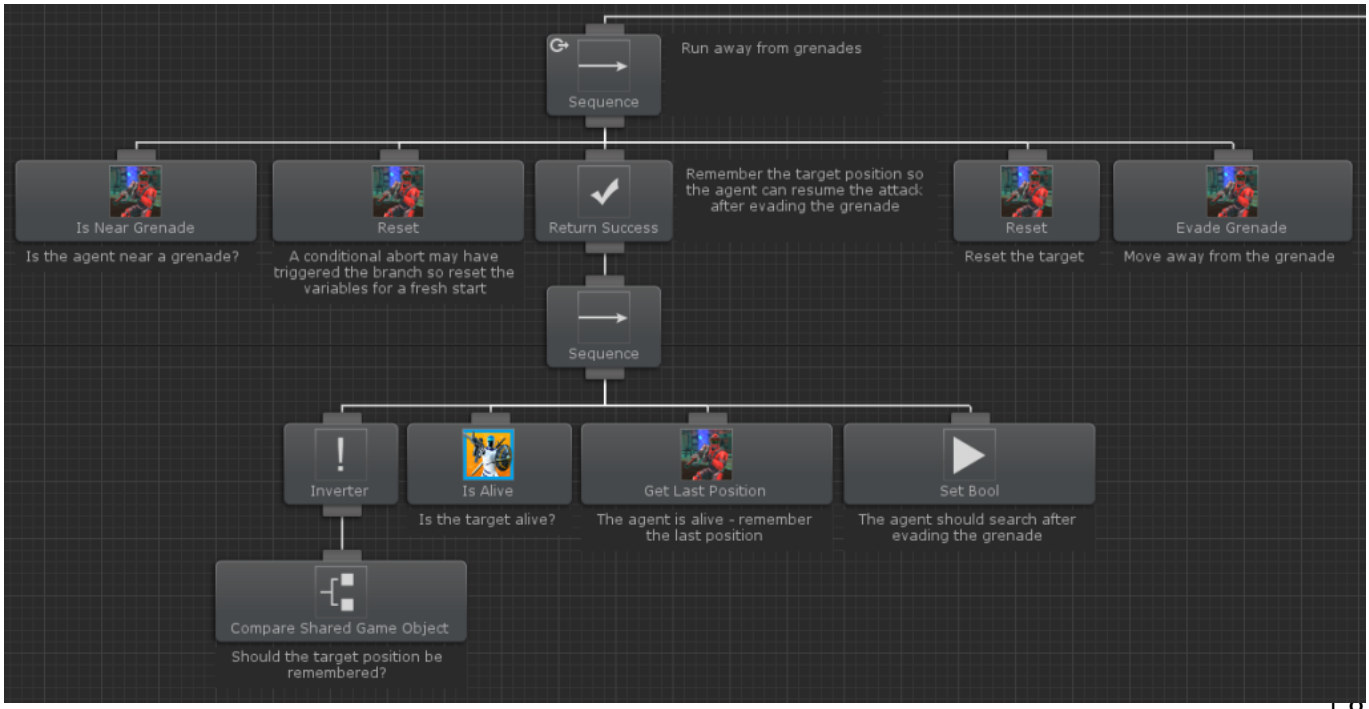

Each of the numbered branches is parented to the Selector task. The branches are arranged in a high to low priority so the Selector will run the branches from left to right. If a higher priority branch (a branch on the left) then the Selector will run the next branch to the right of the just failed branch. This allows us to organize the tree so high priority items (such as evading a grenade) are run before low priority items (such as searching for a target). The following branches are used within the Solo tree:

- Near Grenade
- Damaged
- Can See Target
- Search For Health
- Can Hear Target
- Search For Lost Target
- Search For Ammo
- Take Cover
- Search For Target

Some of these branches only contain a few tasks (such as Search For Health), while others contain multiple subtrees (such as Can See Target). For organizational purposes the rest of this overview is separated into sections based on the branch that they represent.

## Near Grenade

The branch that has the absolute highest priority is the Near Grenade branch:



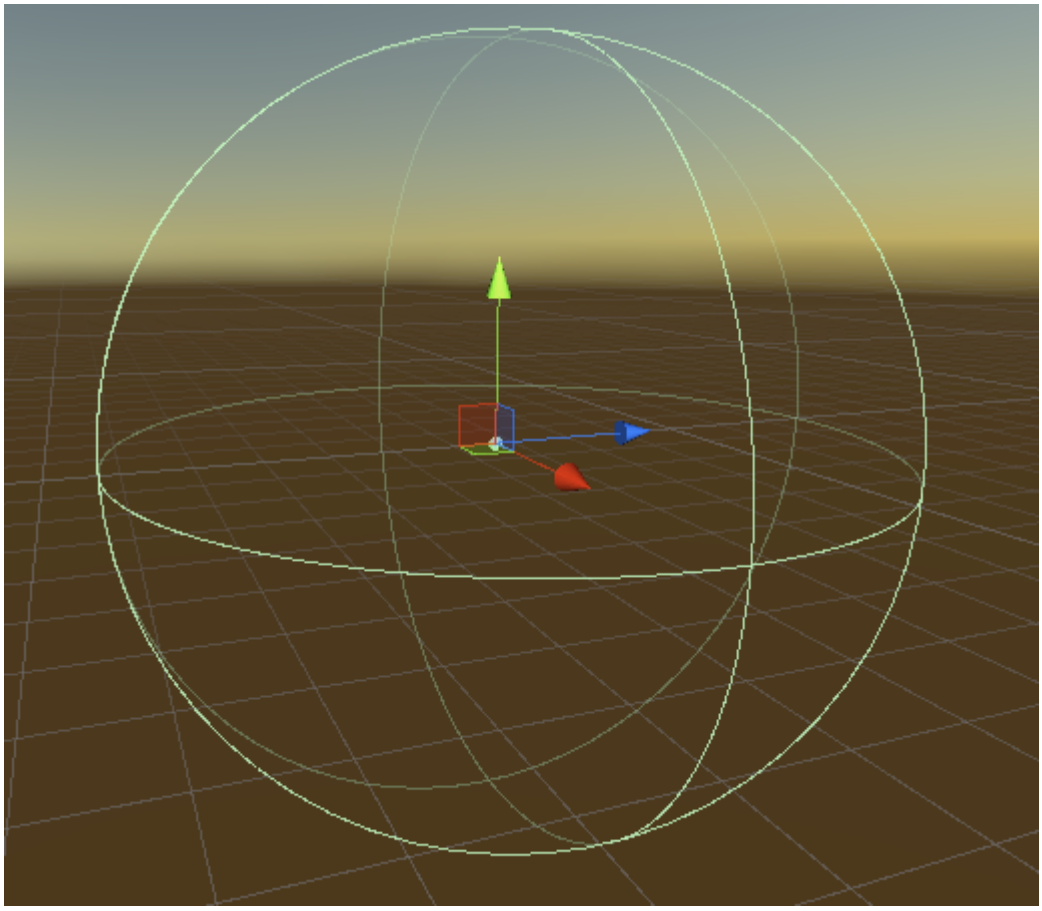
No matter what else the agent is doing, if a grenade comes near the agent then the agent should evade that grenade. This dynamic response is triggered with Conditional Aborts. As a quick recap, conditional aborts allow higher priority conditional tasks to stop the execution of lower priority tasks. The Near Grenade branch uses a Both conditional abort. This means that the Is Near Grenade task can stop the execution of any task that is to the right of it. As an example, if the agent is currently attacking within the Attack branch and a grenade comes near the agent, the Is Near Grenade task will reevaluate from Failure to Success because there is now a grenade near the agent. This will then stop the Attack branch and start the Near Grenade branch.

If Is Near Grenade returns Success the next task that is run is the Reset task. Similar to the Reset task from the previous page, this Reset task will reset the variables so they start fresh. The reason this is needed is because a conditional abort may have interrupted a lower priority branch so that lower priority branch didn't get a chance to clean up after itself.

After the Reset task is called there is another branch which looks similar to the power weapon search branch from before. The Compare Shared GameObject task will compare the Target variable to null. If the Target variable is not null and the target is alive then the agent should go back to searching for the target after they have evaded the grenade. The Get Last Position task will store the target's position into a Vector3 and Set Bool will then set the Wander variable to true. The Reset task is then called again so the Target variable can be reset.

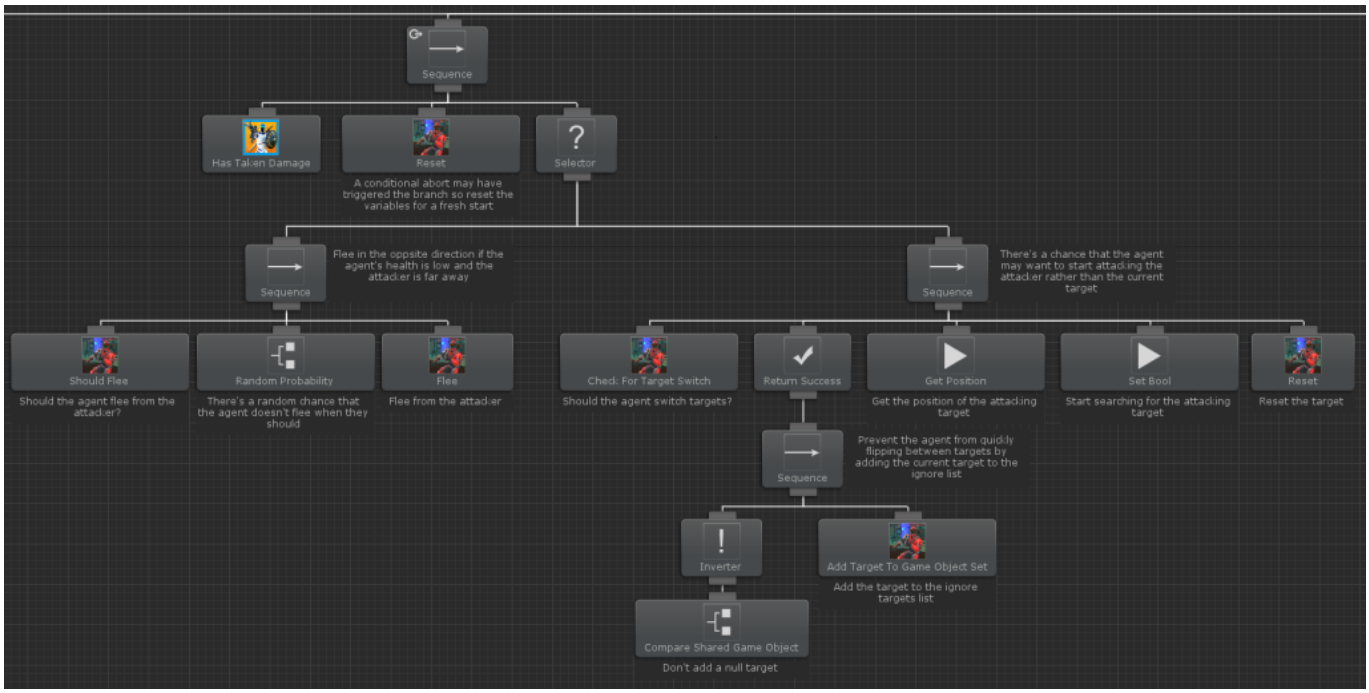
Once these tasks have run the next task that runs is the Evade Grenade task which moves the character to evade the grenade. This task will return Success as soon as the agent is far enough away from the grenade. Notice that the top Sequence task is using a Both conditional abort. This means that the Is Near Grenade task is still reevaluated when any task within this branch is run. This allows for Is Near Grenade to return Failure if for example the grenade explodes and the agent is still alive.

The agents can determine if they are near a grenade by a trigger placed on the grenade. This trigger is in addition to the normal grenade collider and allows the Is Near Grenade task to return Success or Failure depending on if the agent is inside or outside of the trigger.



## Damaged

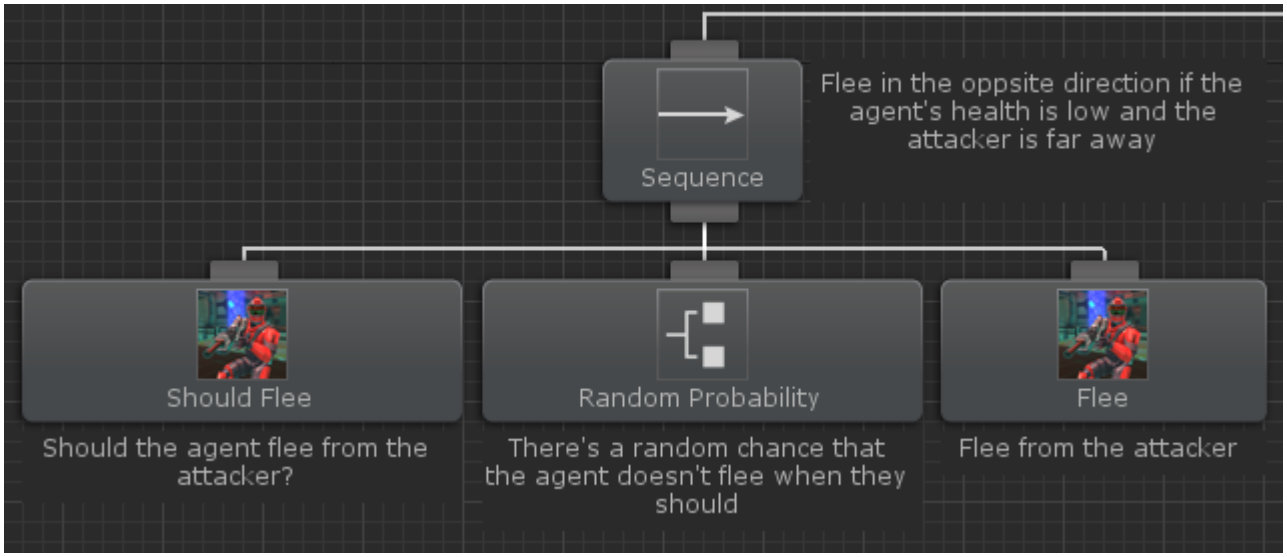
The next highest priority branch after the Near Grenade branch is the Damaged branch:




The Damaged branch makes use of conditional aborts similar to the Near Grenade branch. This time though the top Sequence task uses a Lower Priority abort type instead of a Both abort type. The reason for this is because we don't want to stop any task within the Damaged branch from running if the conditional task changes status. The Lower Priority conditional abort will reevaluate Is Damaged every tick to determine if the agent has taken any damage. If the task returns Success then the conditional abort will trigger assuming any

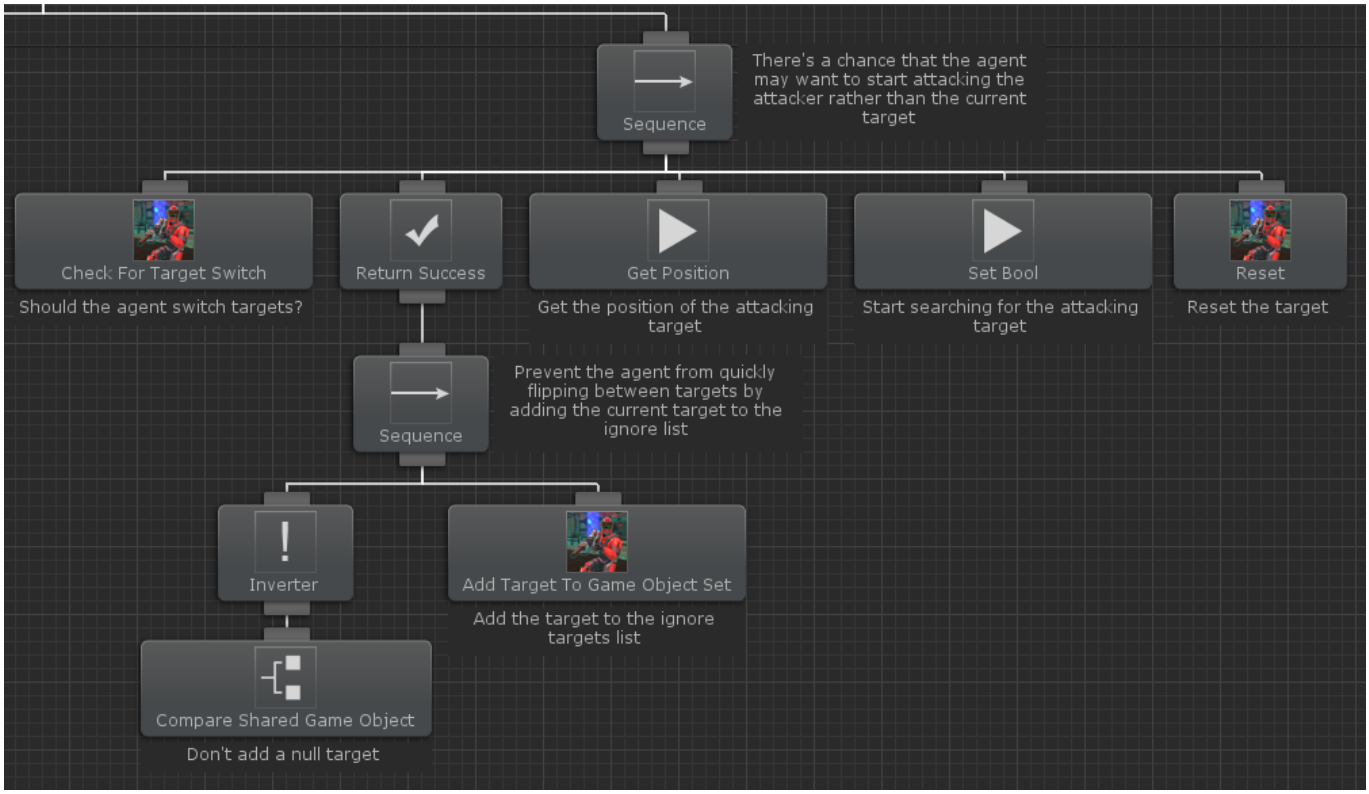
lower priority branch is running. This basically means that the conditional abort will trigger if any branch besides the Near Grenade branch is running.

The first task to run is the Reset task similar to the Near Grenade branch. We want to make sure almost all of the variables start fresh in case the conditional abort triggered the branch to run. Following the Reset task we have a Selector which will first try to run the left child branch and then if that returns Failure it'll try to run the right child branch. First we'll take a look at the Flee branch:




The first task within the left child branch is the Should Flee task. This task decides if the agent should flee from the attacker: if it returns Success then the agent should flee. A Random Probability task is then added to the right of Should Flee to give the agent a chance to run the right branch even though Should Flee said that the agent should flee. If both conditional tasks return Success then the Flee task will run which will move the agent away from the attacking target.

If the agent does not flee than the right branch runs:




The goal of this branch is to determine if the agent should switch its target to the attacking target. The first task that runs is the Check For Target Switch conditional task. This task will evaluate if the agent should switch from the current target to the attacker. If Check For Target Switch returns Success then a small branch will run which determines if the current target should be added to the Ignore Targets variable.

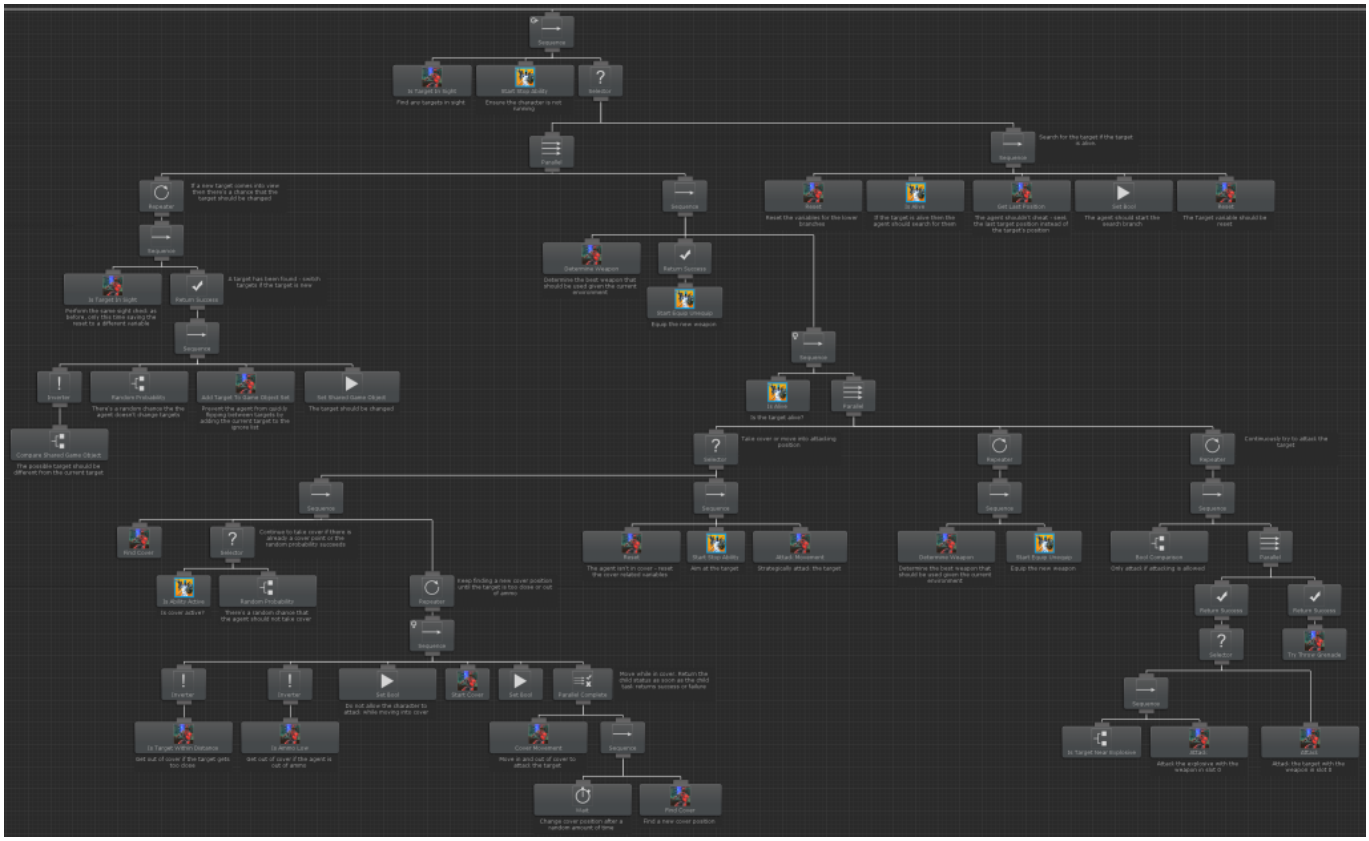
The Ignore Targets variable is a set which keeps a list of targets that should be ignored. This is useful to prevent the agent from quickly switching back and forth between targets. For example, if the agent is originally attacking Character 1 and then Character 2 starts to also attack the agent. The Damaged branch may switch the target to Character 2 but in the meantime Character 1 is still attacking. At this point Character 1 would have been added to the Ignore Targets set so the agent will not switch back to Character 1.

The Compare Shared GameObject compares the Target variable to null. If the target is null then the task will return Success but we want to know if the target is not null so an Inverter has been parented to the task. If the target is not null then the Add Target To GameObject Set task is run.

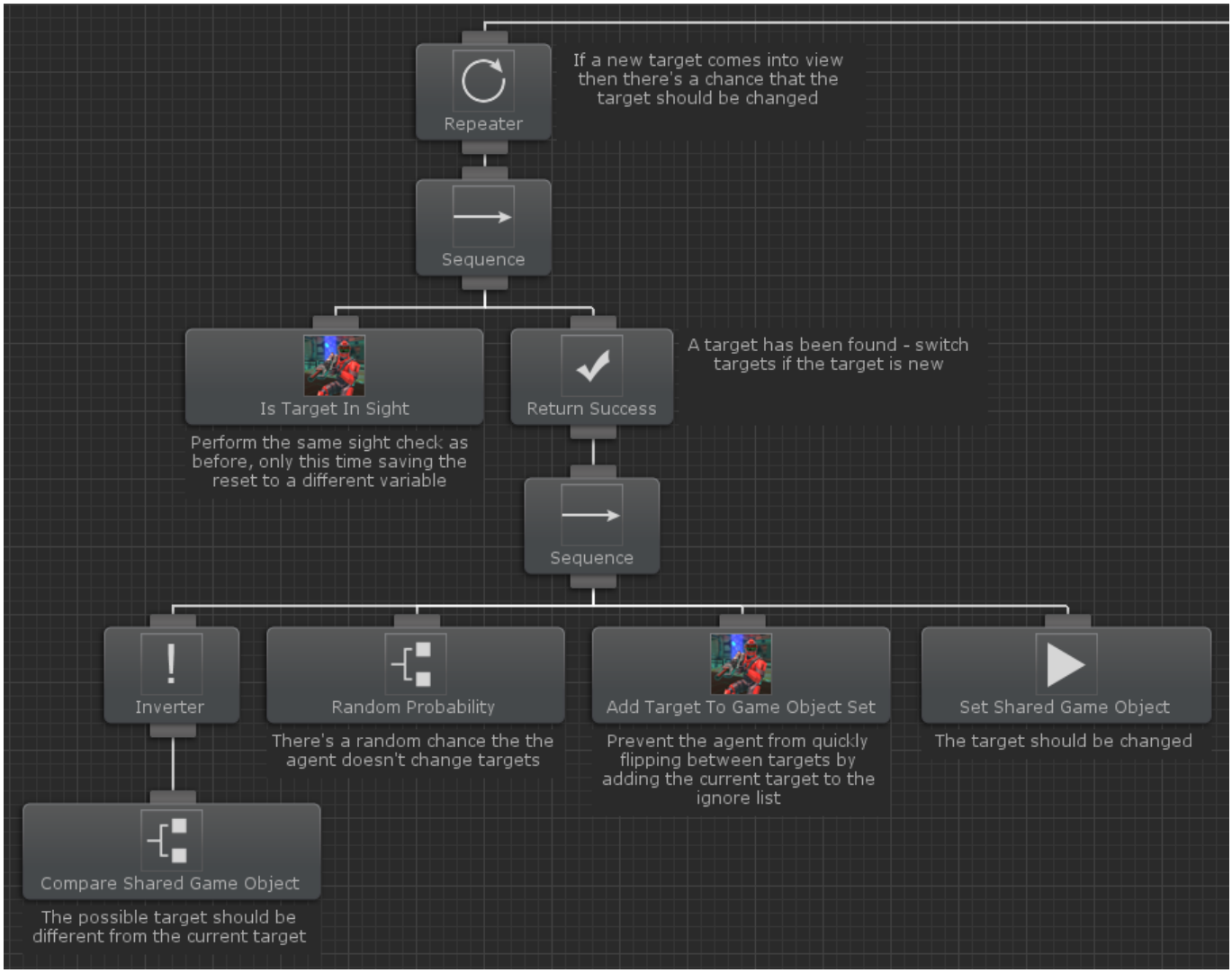
After the target was possibly added to the Ignore Target set the next task will get the position of the target and following that the Search variable will be set to true. This allows the agent to start looking for the attacker that hit the agent. The last task that is run is the Reset task which will reset the Target back to null since no targets are currently within sight.

## Can See Target

The Can See Target branch contains all of the logic pertaining to attacking if a target is within sight. This branch contains many tasks and has been divided into multiple braches and two subtrees. Here’s what the branch looks like with just a reference to those subtrees:



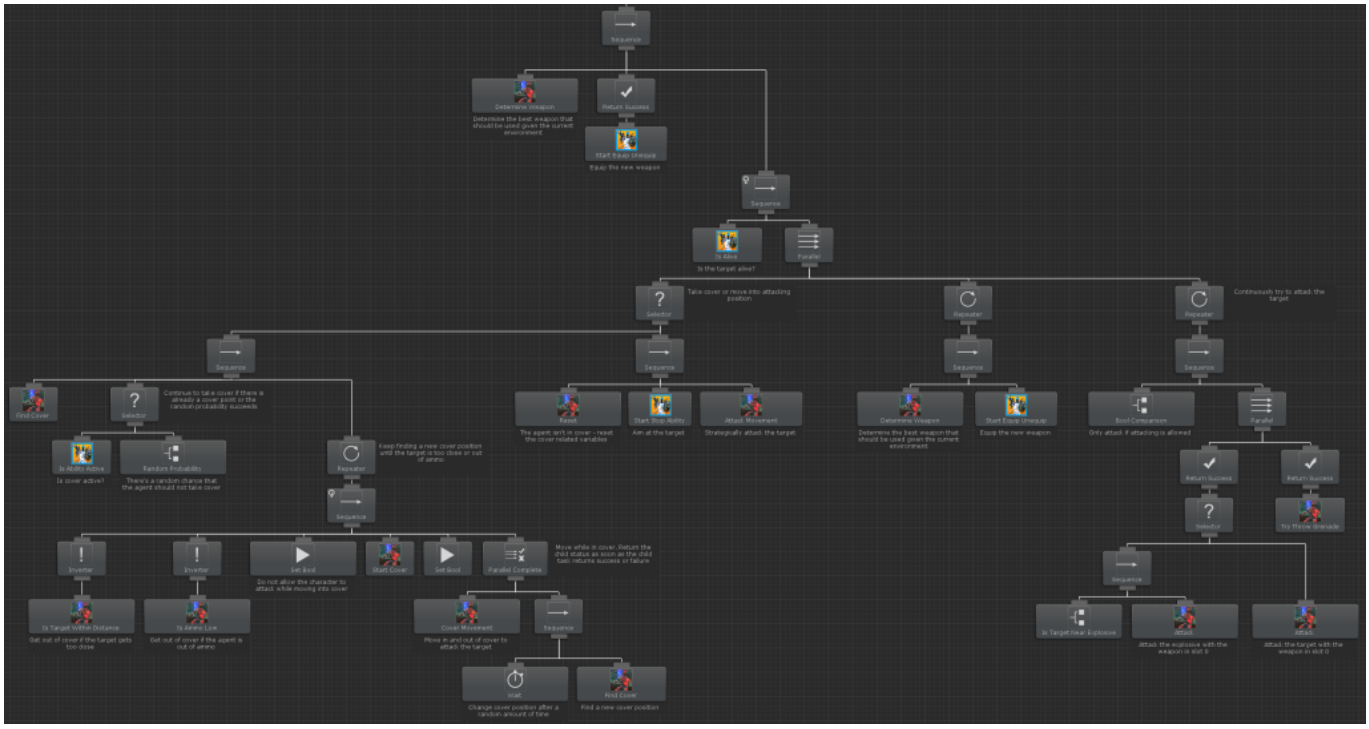
A Lower Priority conditional abort is used so the Is Target In Sight task will be reevaluated if any branch to the right of the Can See Target branch is active. When Is Target In Sight returns Success it means that there is a target within the agent's field of view. The Start Stop Ability will then run to ensure the character is not running. A Selector is placed so the left branch can attack and the right branch will be run after the attacking is finished. Let's look at the left branch first. The Parallel task is used to run both the left and right branches at the same time. This allows the agent to possibly change targets while still attacking the original target. The left branch looks like the following:




At the top of this branch is a Repeater. The Repeater will allow the child tasks to be reevaluated each tick. The child Sequence task then contains two children: Is Target In Sight and Return Success. The Is Target In Sight is the same as the one at the top of the Can See Target branch except this one will store the target in the Possible Target variable instead of the Target variable. This allows the existing target to be kept if for some reason the agent doesn't want to switch to the new target that just came within sight.

The branch below the Return Success looks similar to a setup before where we first compare the Possible Target variable to the current Target variable and if it's different then there's a chance that the agent should switch based on the Random Probability. If the agent should switch then the current target is added to the Ignore Targets set and the Set Shared GameObject task will set the Possible Target to the Target.

As the agent is continuously checking for new targets they are also attacking with the branch on the right:



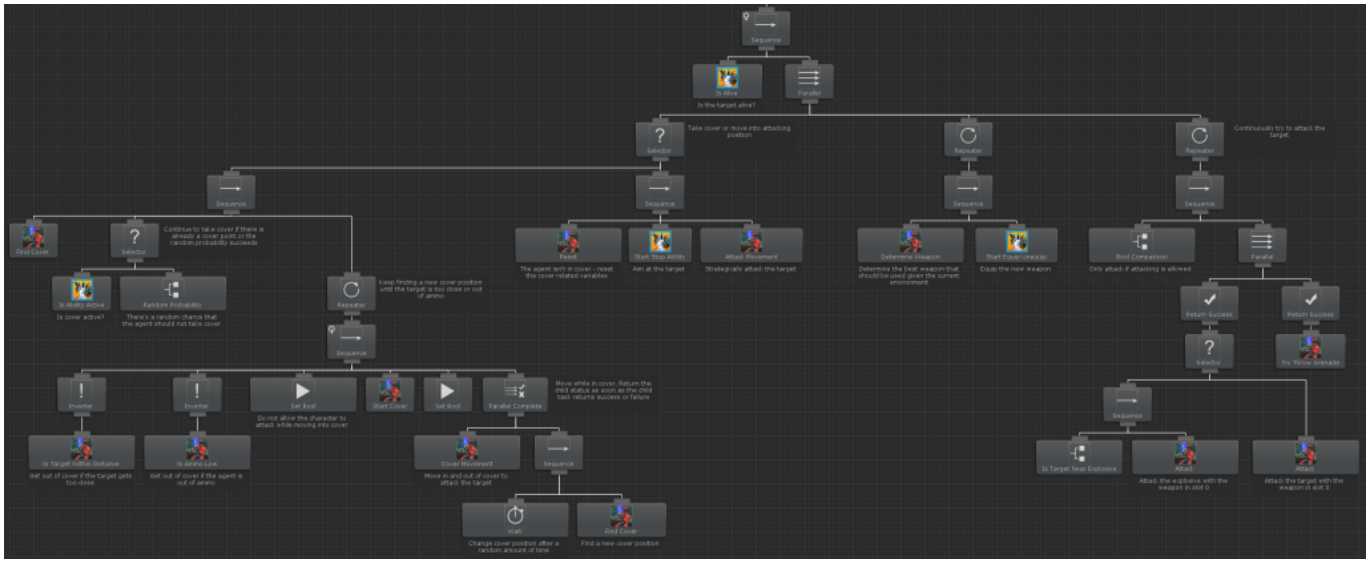
Without the Attack subtree pulled in this is a small branch which first determines the best weapon for the current game state. This task is only run once and allows the agent to have a weapon ready to go when the Attack branch runs. The Start Equip Unequip task will equip the Item Set returned by the Determine Weapon task. The rightmost task is a Behavior Tree Reference task which loads the Attack branch.

The Attack branch is explained on this page but a quick summary is that it will allow the agent to actually fire at the target. As soon as the agent is done attacking (because the target died or is no longer in sight), the following branch will run:


The Reset task will run and reset all of the variables so the lower branches start fresh. If the target is still alive then Is Alive will return success and the next three tasks will run. The target will still be alive if Is Target In Sight returns failure meaning the target is no longer in sight (such as if the target ran away and took a corner). The agent will then get the last position of the target and set the Search variable to true so the agent will search at that position. Finally, the Target variable is reset since it's not needed anymore.

## Attack

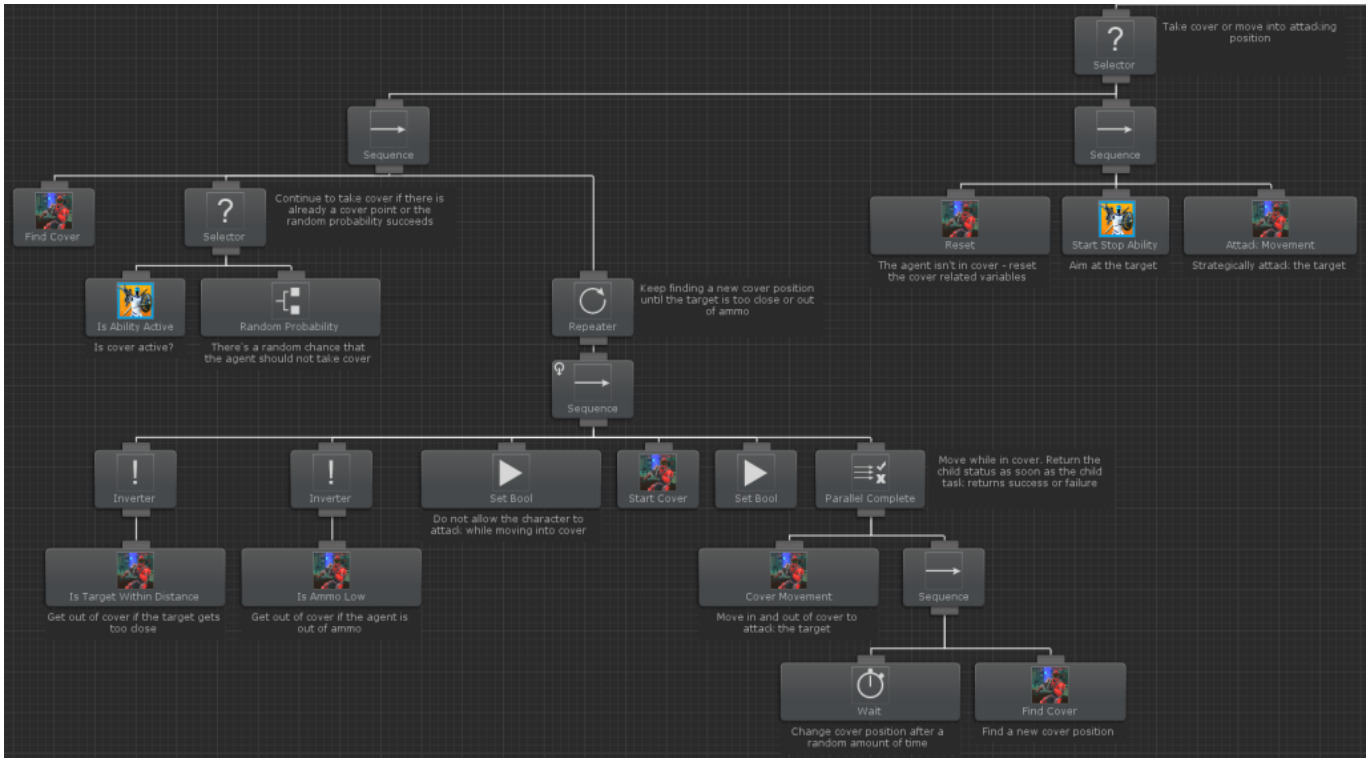
The Attack branch does the actual attacking of the target:



The Attack branch uses a Self conditional abort to reevaluate the Is Alive conditional task every tick. This will stop the agent from attacking if the target is no longer alive. If the target is alive three branches run at the same time:

- The branch that moves the agent into attack or cover position.
- The branch that determines the best weapon.
- The branch that does the actual attack.

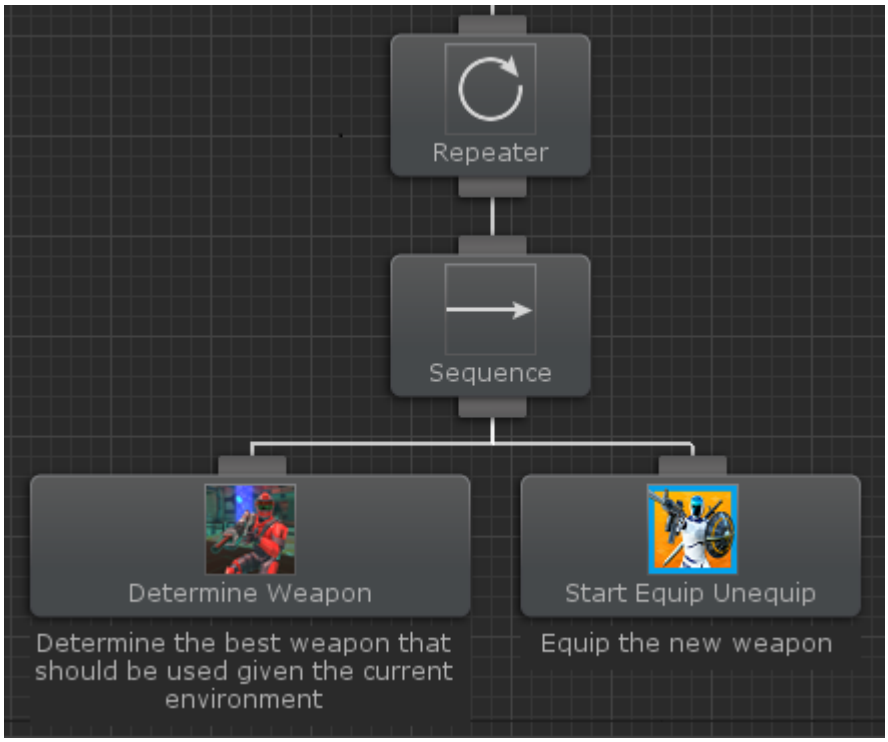
Let's take a look at these branches one at a time. The first branch is the movement branch:



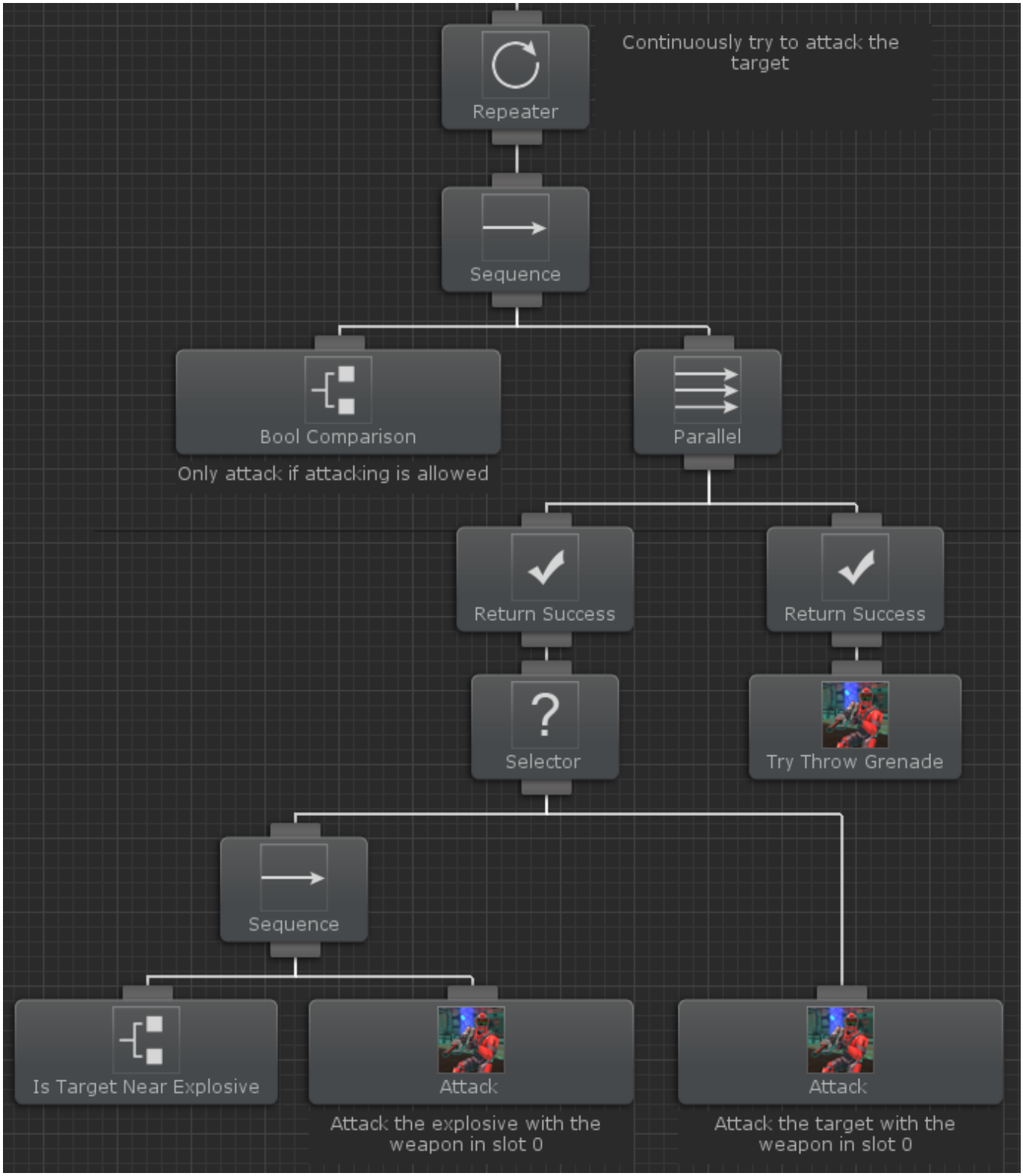

A Selector is used so one of two branches will run: either a cover branch or a regular attack movement branch. The Find Cover task will return Success if the agent is near a Cover Point and should move into cover. There is a chance that the agent is already in cover from a lower priority task so the Is Ability Active task will return Success if the Cover ability is active. If the Cover ability is not active then there's a chance that the agent will not take cover even though the Find Cover task returns Success. This happens with the Random Probability task. If either the Is Ability Active or the Random Probability task returns

Success then the Cover branch will run.

If the cover branch should not run then the agent will run the Attack Movement task. Before the Attack Movement task is run the Reset task is used to reset the cover variables. The Start Stop Ability task will also ensure the agent is aiming at the target. The Attack Movement task will move the agent into attacking position. While the agent is moving into position the Determine Weapon branch also runs:

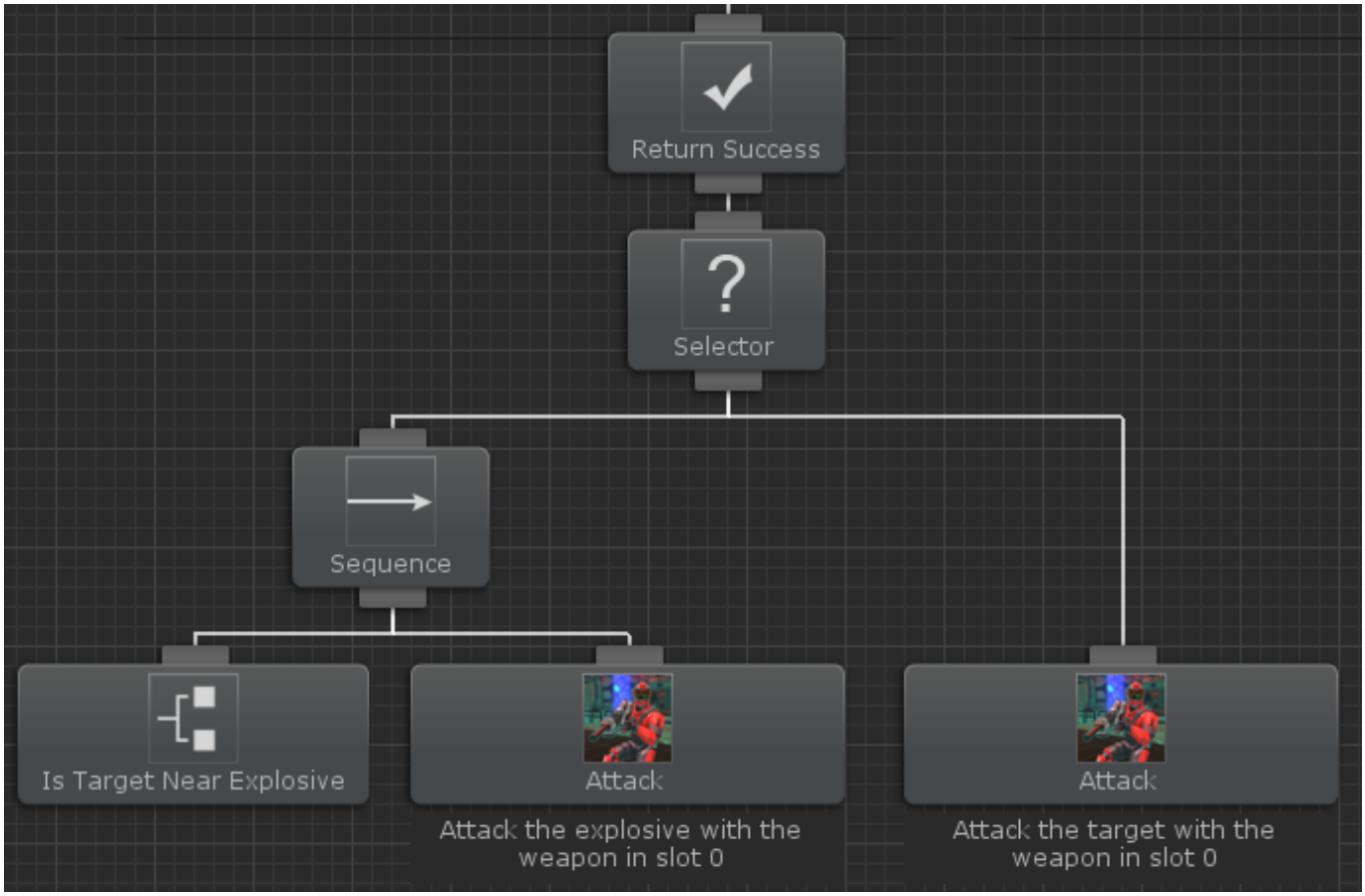



The Determine Weapon branch is parented by a Repeater to allow the agent to determine the best weapon for the situation every tick. The Determine Weapon task will set which Item Set should be equipped and the Start Equip Unequip task will equip that item. In addition to determining which Item should be equipped, the agent is also doing the actual attack:

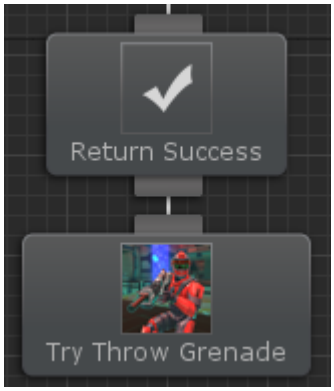



Similar to the Determine Weapon branch, the attack sub-branch will use a Repeater to continuously try to attack. Before the agent can actually attack though with the Attack task it must first check the Can Attack bool variable. This variable is set by the Cover branch and will set it to false if the agent is moving in or is between cover positions.

The Parallel task is used so the agent can attack or throw a grenade independently of one another. On the left side of the parallel task the agent will perform the attack with the equipped weapon:



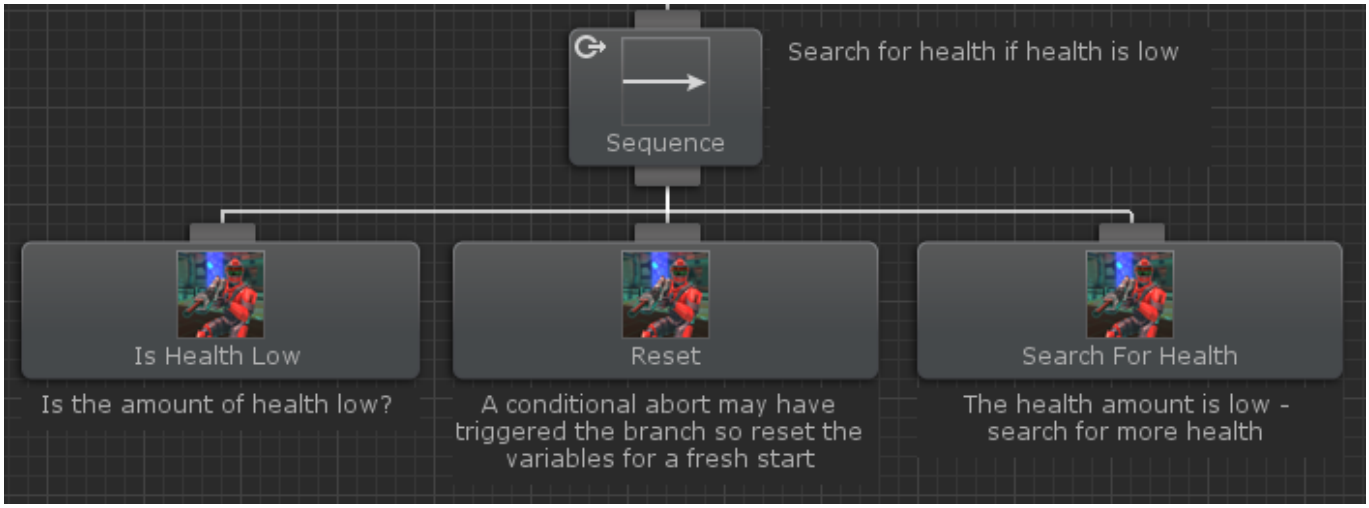

The Return Success task ensures the Parallel task will keep on running even if the attack fails. A Selector is then placed to allow the agent to attack an explosive barrel if the target is nearby. This is checked with the Is Target Near Explosive task. If the target is near an explosive then the Attack task will fire at the explosive, otherwise the Attack task will fire at the target.




The right branch under the Parallel task will throw a grenade if it should be thrown. There are many conditions which would prevent a grenade from being thrown so the grenade is not thrown more often than it is thrown. The Try Throw Grenade task is parented to the Return Success task to keep the branch active.

## Search for Health

The Search For Health branch is the next highest priority branch:

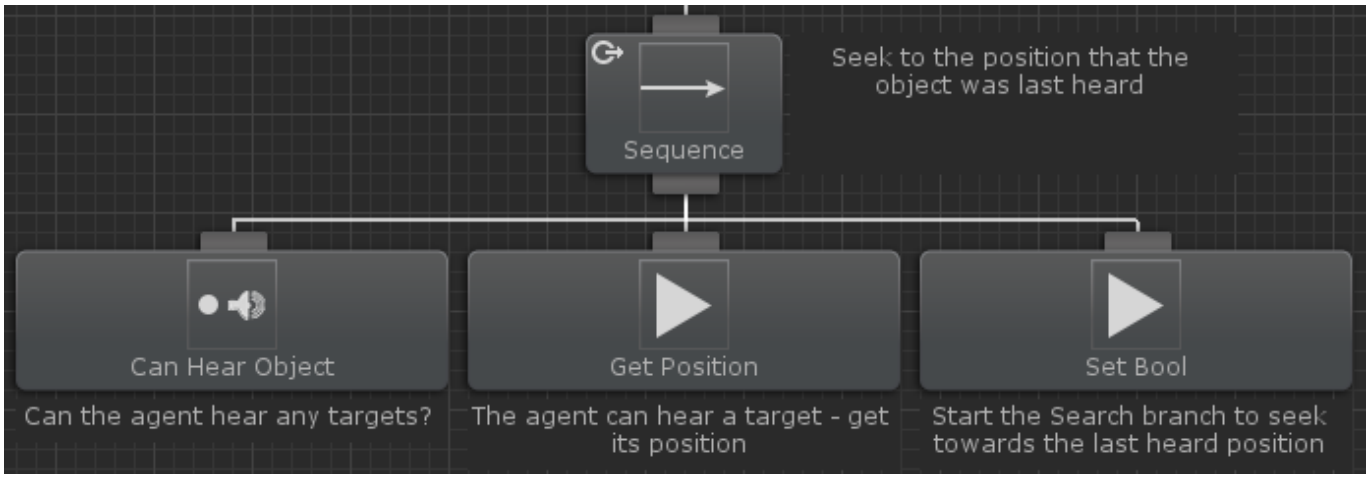



The Search For Health branch is a small branch which will search for health if the agent's health is low. This branch uses a Lower Priority conditional abort so it can abort any branch to the right of it.

The Is Health Low conditional task will return success if the health is lower than the specified amount and the agent does not have a target. This ensures the agent doesn't always start searching for health while they are attacking a target and their health is getting low. The Reset branch will then reset the variables to ensure they have a fresh start after the branch has ended. The Search For Health task will then move the character to a position where there is a health pack. As soon as the agent finds the health pack the task will return Success.

## Can Hear Target

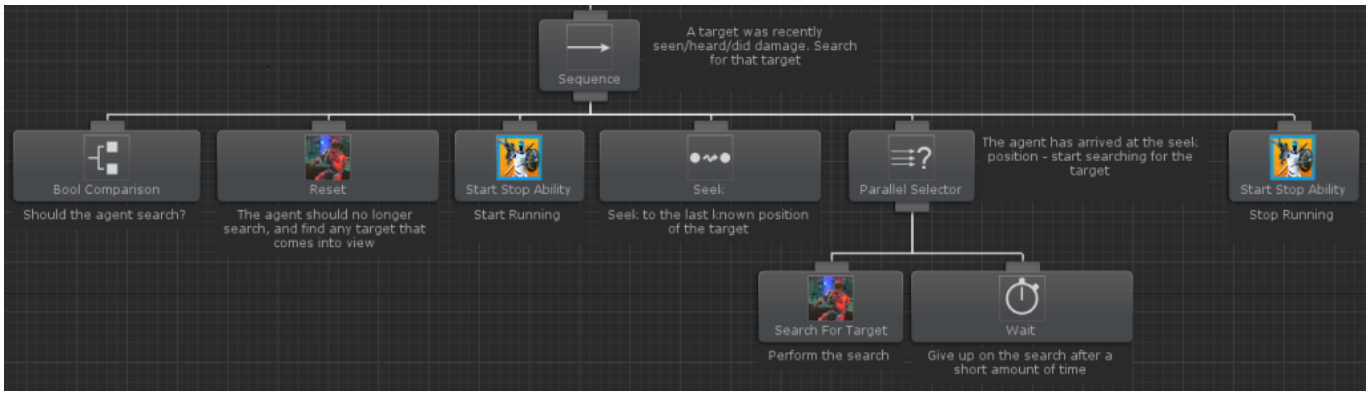
The Can Hear Target branch is a small branch which will detect if the agent hears a sound from an enemy:




The Can Hear Target branch uses a Lower Priority conditional abort so it'll interrupt any branch to the right of it. If the agent does hear an enemy with the Can Hear Object task then the position of that heard object is saved and the Search bool is set. By setting the Search bool to true the agent will run the Search For Lost Target branch.

## Search For Lost Target

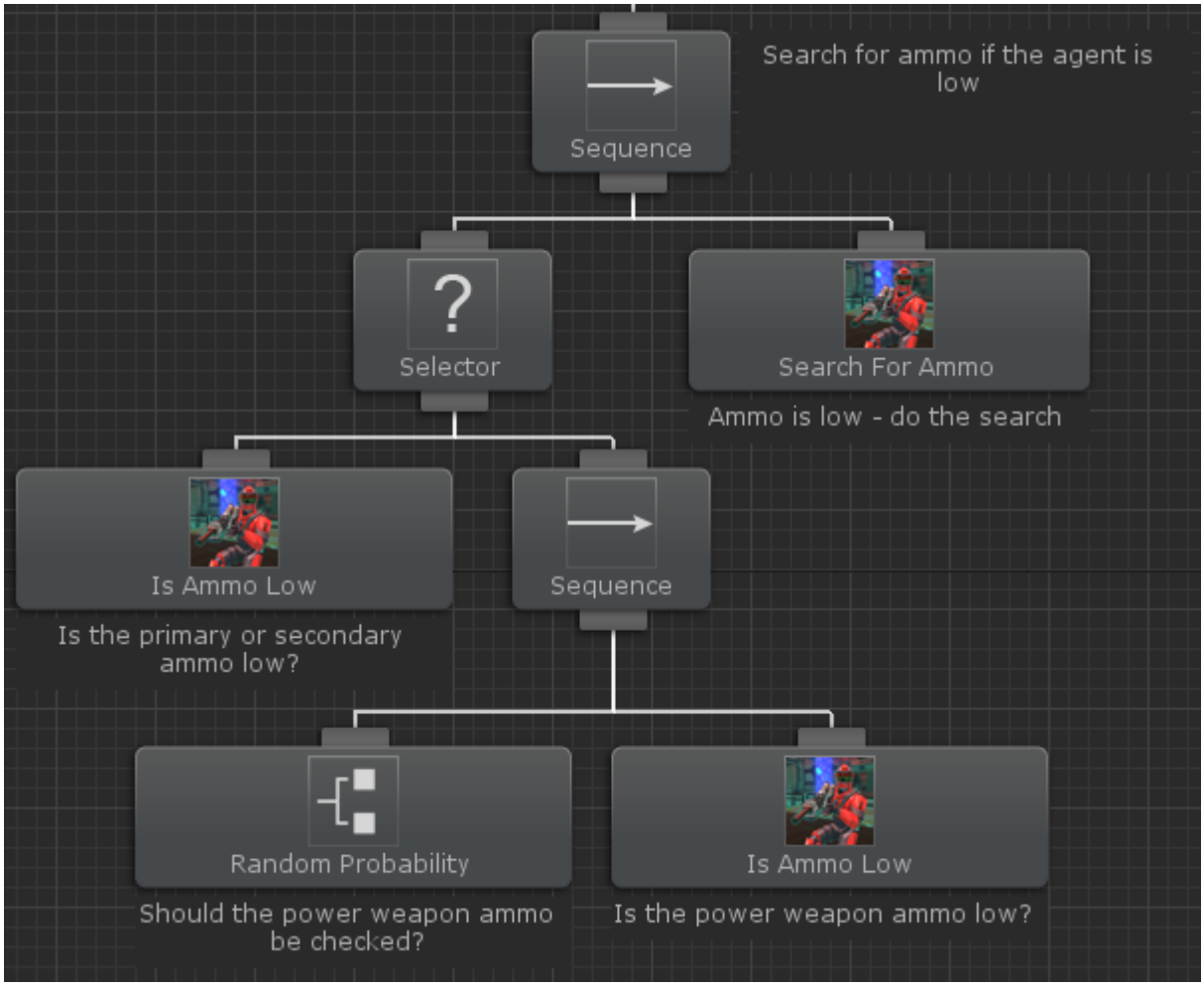
The Search For Lost Target branch will run when the agent saw/heard/took damage by a target and that target is no longer in the agent's sights:




If a target was recently lost the Search bool variable will be set to true and the Bool Comparison task will return Success. The Reset task will reset the Ignore Target and Search variables so the agent will find any targets that come into sight. The Start Stop Ability task will activate the Speed Change ability which will have the agent run while searching for the lost target. The agent will then Seek towards the last known target position. As soon as the agent has arrived at that position the Search For Target task will perform the actual search. This task is parented by a Parallel Selector so the Wait task can also run at the same time. This will prevent the agent from searching too long – as soon as the Wait task returns Success the Search For Target task will also stop. Finally after the agent is done searching the Start Stop Ability task will deactivate the Speed Change task.

## Is Ammo Low

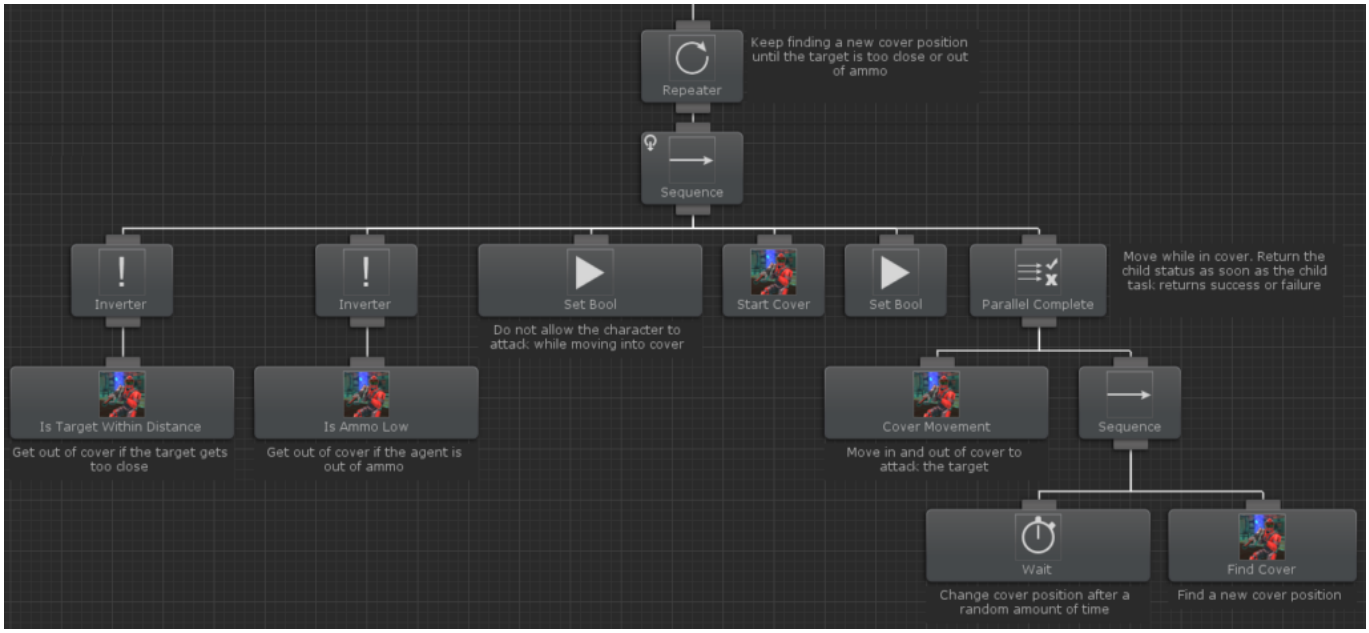
The Is Ammo Low branch will search for ammo if the agent is close to running out:




Before the agent can search for ammo they first have to know what type of ammo to search for. The first conditional task is parented by a Selector so the agent can either search for non-power weapon or power weapon ammo. If the first Is Ammo Low condition returns Failure then the branch with the Random Probability will run. The agent shouldn't always search for power weapon ammo so the Random Probability was added to introduce some randomness. If either Is Ammo Low tasks return Success then the Search For Ammo task will run and this task will return Success as soon as the agent has picked up more ammo.

## Cover

The Cover branch is used in two locations: the first is within the Attack branch and the other in an independent branch reevanuated with a Lower Priority abort type. The Cover branch looks like:



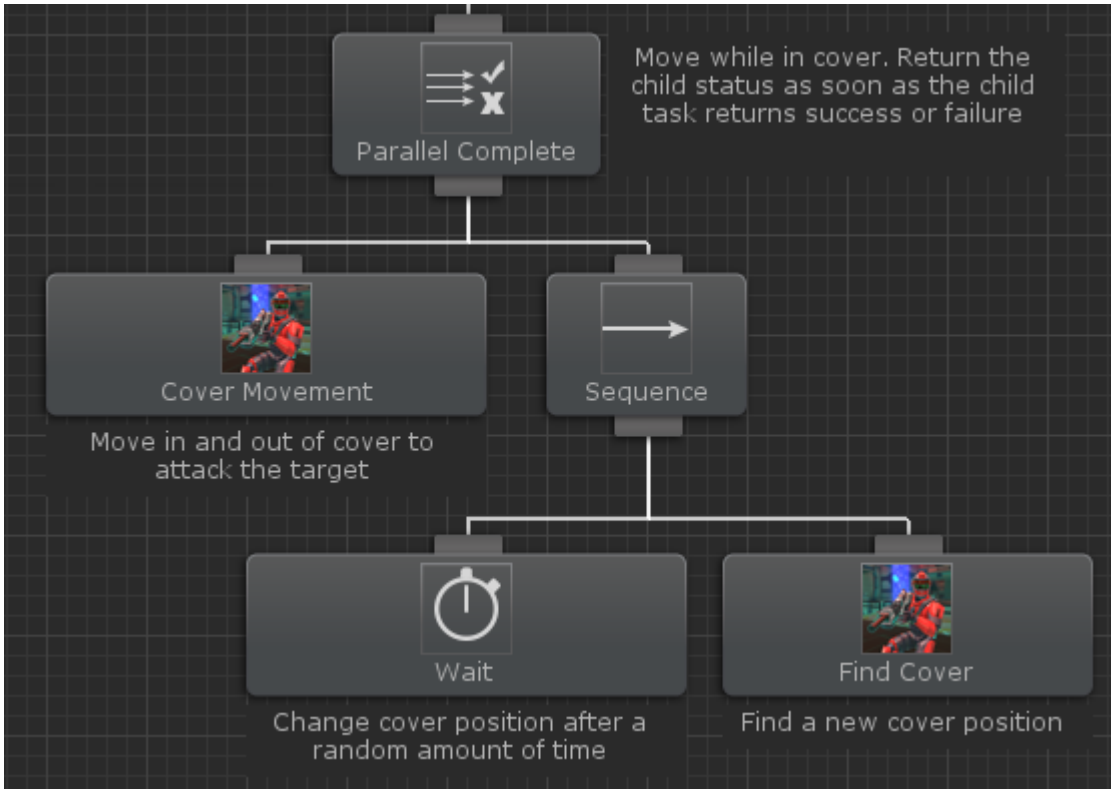

When an agent is in cover the Cover Point can specify a linked Cover Point which allows the agent to move between cover positions. The Repeater at the top of the Cover branch will do the actual switch between cover positions. Just below the Repeater is a Sequence task which has the Self abort type. This allows the agent to get out of cover if either of the first two conditions return Success.

The Is Target Within Distance task is the first condition that is checked for the Cover branch. It is parented by an Inverter so the agent stops taking cover if the target comes too close to the agent. The second conditional task is the Is Ammo Low task. If the agent is low on ammo then they should get out of cover.

Assuming these two conditional tasks return Failure (and thus return Success with the Inverter), the Set Bool task will run which sets the Can Attack bool to false. This prevents the agent from attacking while moving into cover. The Start Cover task will then run to move the agent to the Cover Point.

As mentioned at the start of this section, the Cover branch can be called from two different locations. The first is within the Attack branch and the second is in a lower priority branch just to the left of the branch which searches for the target. The agent may be taking cover in hopes that a target comes within their sights. If a target does come within sight the Can See Target branch will abort the current Cover branch and start running the Cover branch that is within the Attack branch. If the agent is already in cover the Start Cover task will move them to a new cover location only if the Find Cover task returned a new location. After the agent is in cover the Set Bool task will reset the Can Attack variable so the agent can attack.

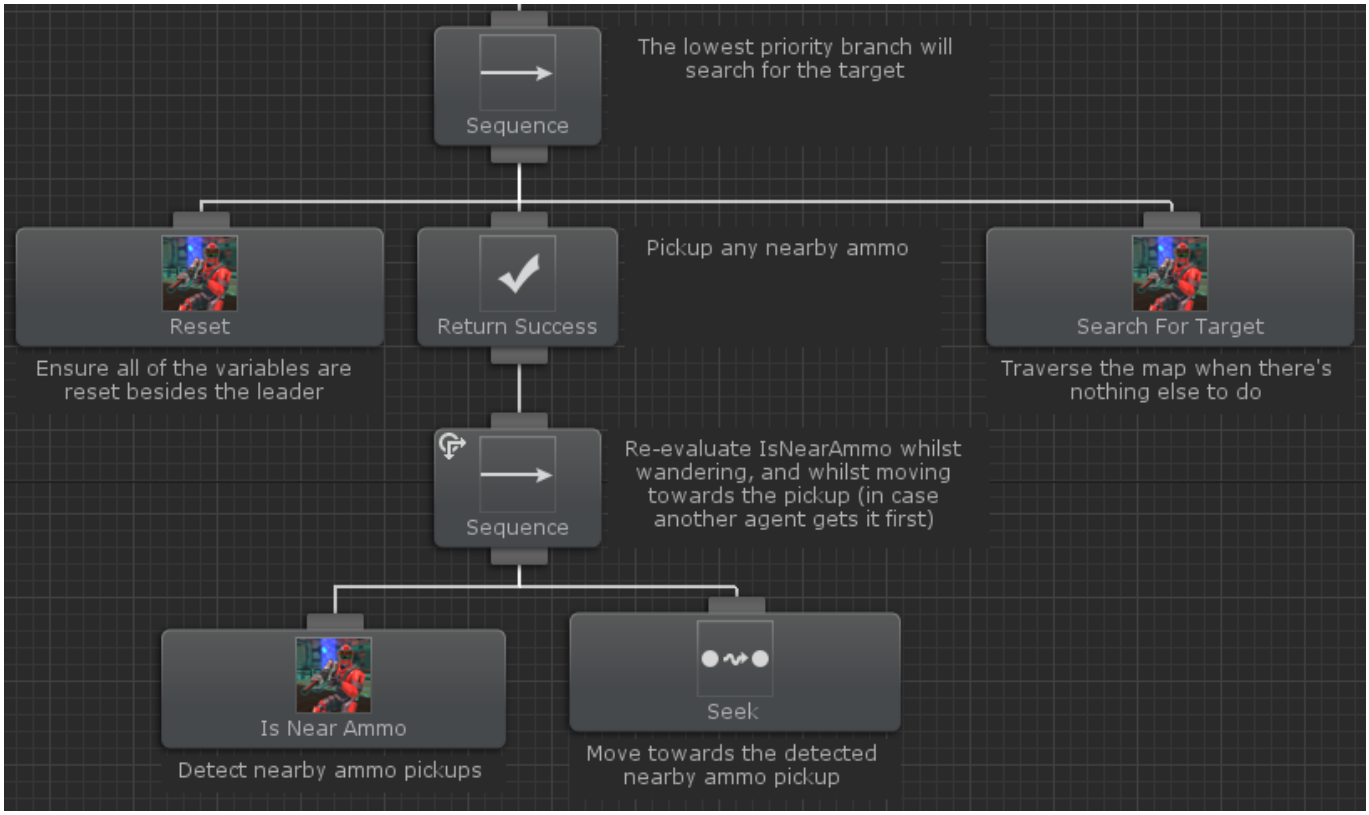
As soon as the agent is in cover they should aim in and out of cover to be able to attack the target. In addition, a separate timer should start to determine when the agent should change cover points. The last branch accomplishes both of those goals:




The Parallel Complete task is similar to the Parallel Selector task except it will stop as soon as the child returns Success or Failure. The Parallel Selector only stops if the child returns Success. The purpose of this task is to stop running both children when either child finishes executing. Cover Movement will decide when the agent should take cover versus aim out of cover and attack the target. If the target is no longer in sight then Cover Movement will return Failure. The other branch will wait a specified amount of time before selecting a new cover point with Find Cover. Find Cover will return Success if a new cover point has been found. In both of these cases the other branch should stop running if one of the branches returns a Success or Failure status.

## Search For Target

The branch with the lowest priority is the Search For Target branch:

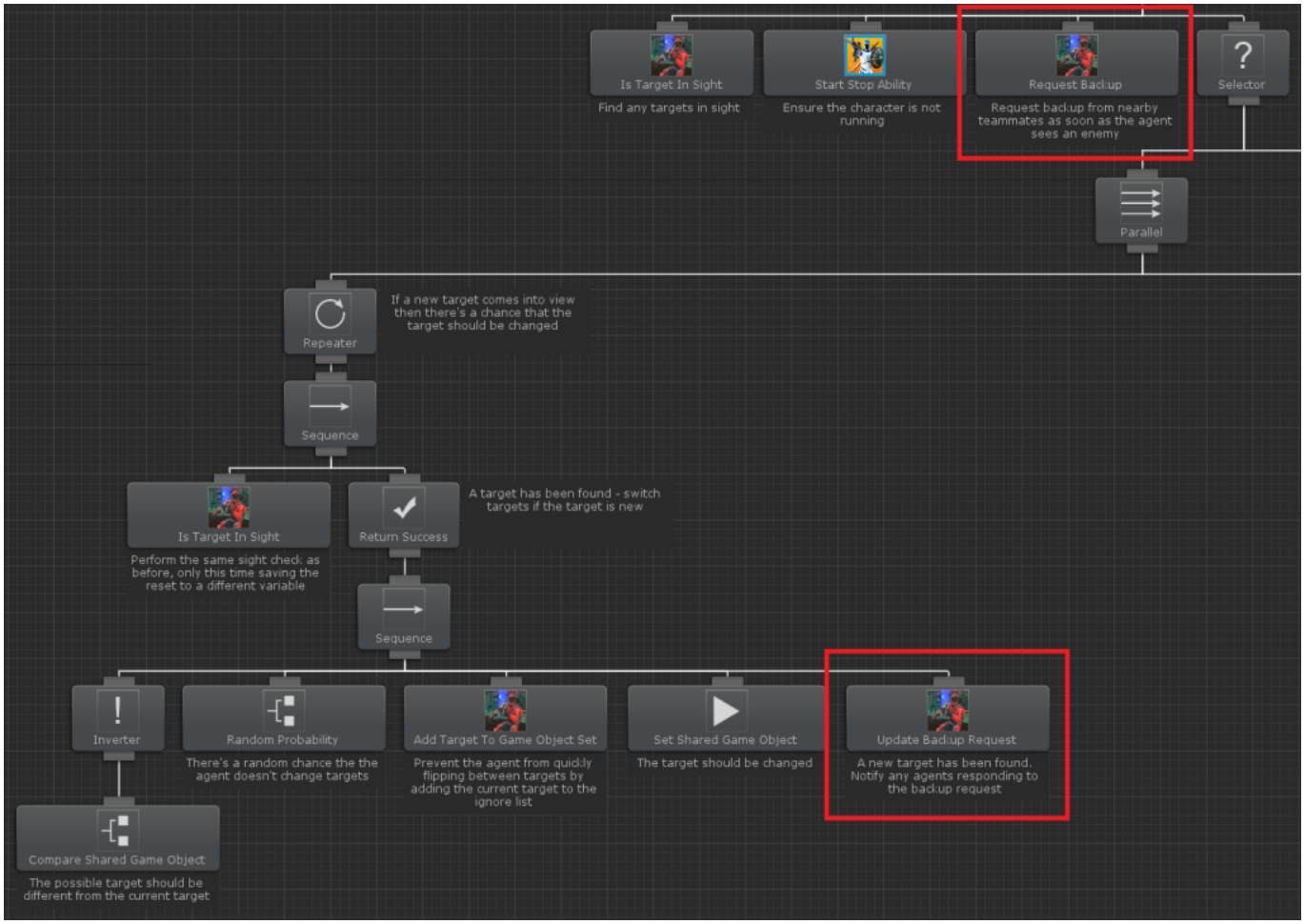



If no other branches can run then the Search For Target branch will run. This branch does not require any conditions in order to run. The first task that this branch runs is the Reset task. By running this task it ensures that all of the variables have been reset to their starting values. There is a small sub-branch which will trigger if the agent is near ammo: the Is Near Ammo task is reevaluated using a Both conditional abort. This means that if the Search For Target task is active the Is Near Ammo task can return success so the agent will abort searching for a target and instead Seek to the ammo location. As soon as the agent picks up ammo the Search For Target task will resume until it is aborted again by a higher priority branch.

## Team Tree

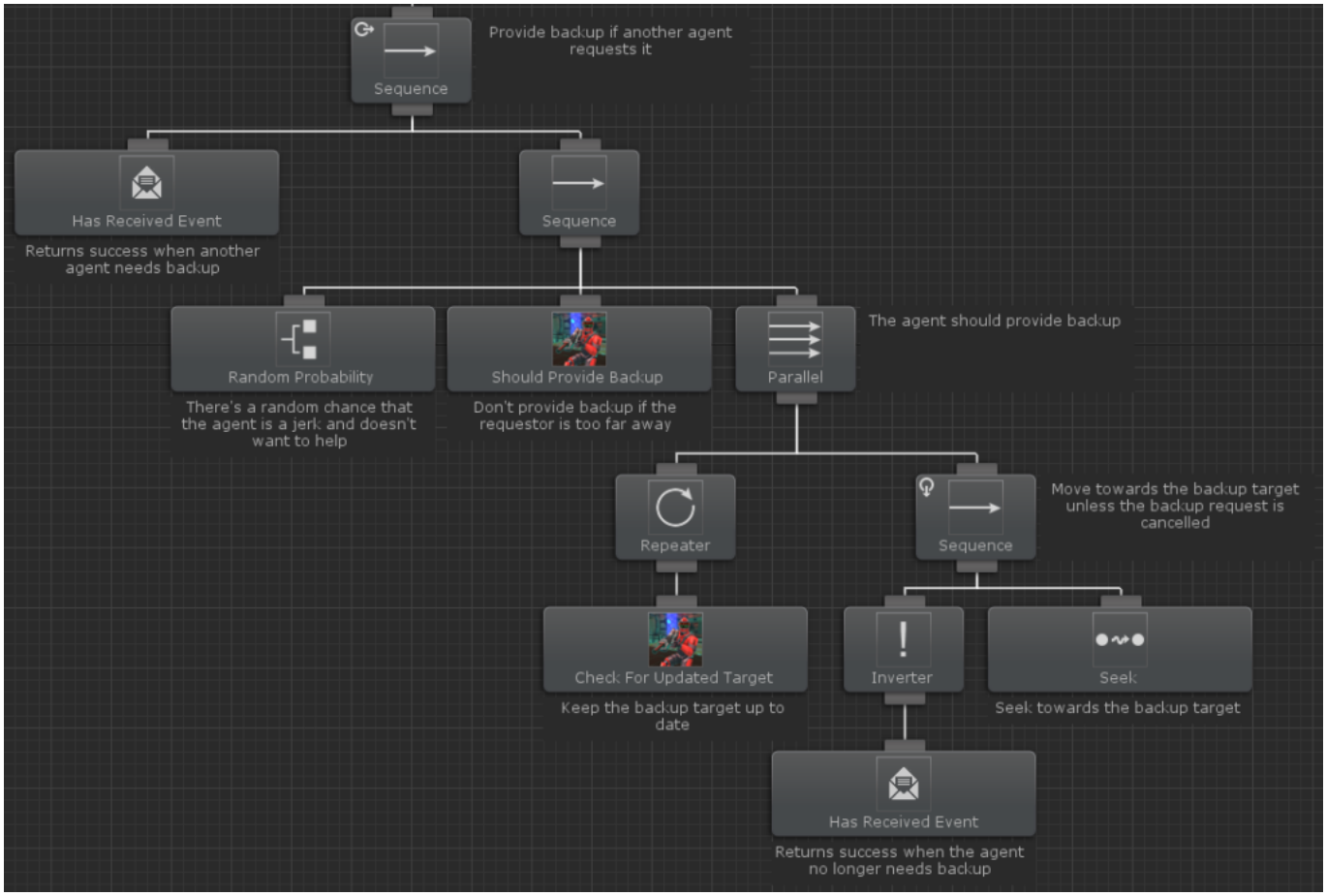
The Team behavior tree has the same structure as the Solo behavior tree. The main difference between the two is that the Team behavior tree allows agents to request and respond to backup as well as create a formation when searching for the target. The Team behavior tree looks like:

Since this tree is so similar to the Solo behavior tree we'll just go over the differences. The first difference is within the Can See Target branch:




As indicated by the highlighted Request Backup task at the top, when the agent can see a target the agent will request for backup. This will allow the agent's teammates to respond and come to help. The Request Backup task notifies the agent's teammates by sending the "RequestBackup" event using the Behavior Designer event system. If the Is Target In Sight check returns a new target then the Update Backup Request task will run to notify all of the interested teammates that there is a new target. This task is highlighted at the bottom of the above image.

Immediately following the Search For Lost Target branch is the new team branch which responds to backup requests:



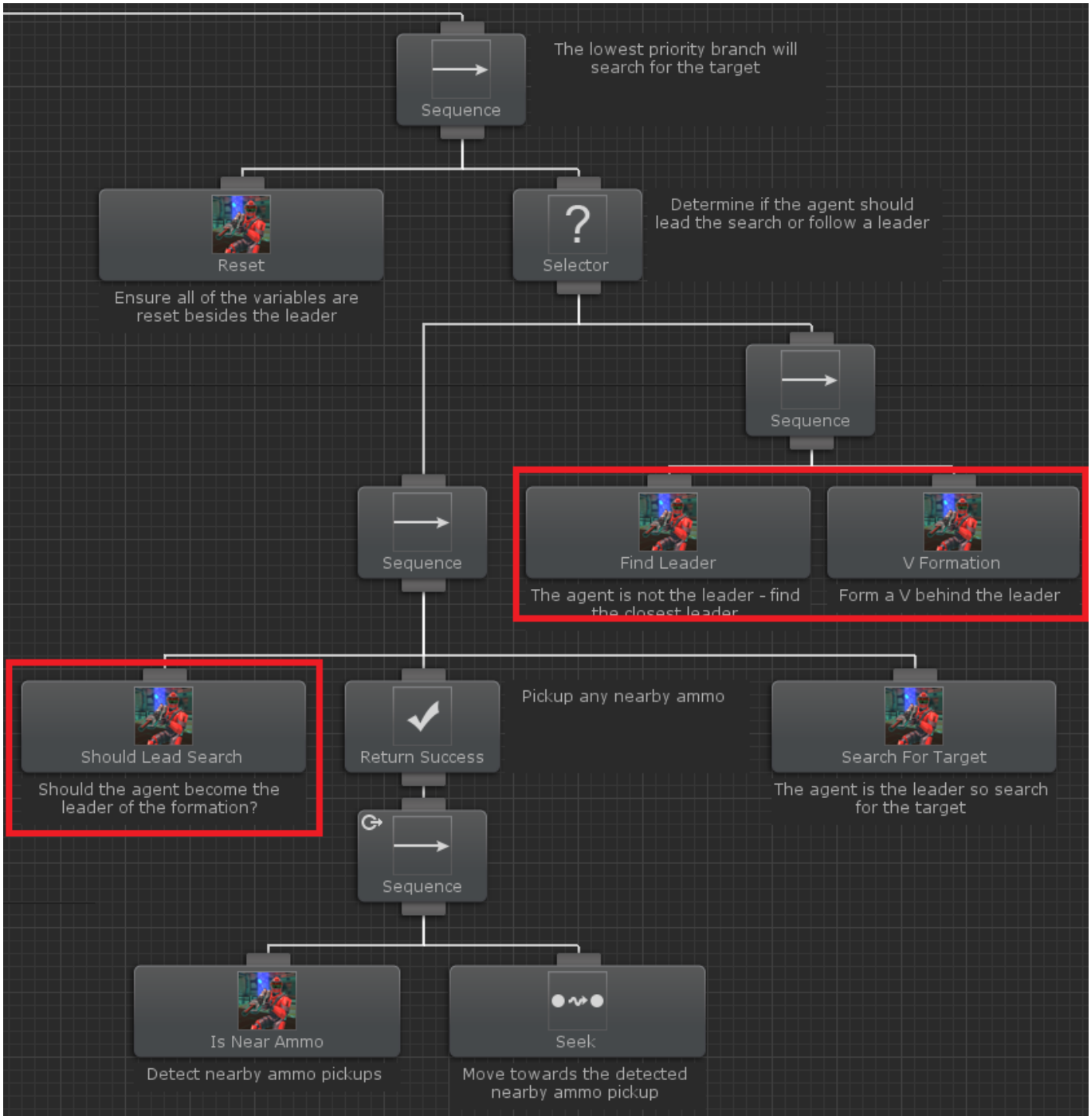

The Has Received Event task will receive events sent by the Request Backup task. The Has Received Event task is being reevaluated using a Lower Priority conditional abort so it will be triggered when the agent is not running a branch with a higher priority than the backup branch. If the agent has received a backup request event they can decide to ignore with the Random Probability task. Notice there is a Sequence task directly underneath the top Sequence task. This will ensure the Lower Priority conditional abort is only triggered by the Has Received Event task.

After the Random Probability task has run the Should Provide Backup task will run. This task determines if the agent should provide backup. One reason the agent won't want to provide backup is if the backup requester is too far away. Assuming the agent does want to provide backup a Parallel task will run which runs two branches:

The first branch, on the left, will continuously check to ensure the backup target is up to date. Within the Can See Target branch from earlier the agent may update the target with the Update Backup Request task so the agent responding to backup should ensure that they are always responding to the correct backup target.

The second branch that runs in parallel is the branch that will actually move the agent to the backup target with the Seek task. At any point the backup request may be cancelled (such as if the target dies) so a Self conditional abort is used to continuously reevaluate the Has Received Event task. This task monitors the "CancelBackupRequest" event rather than the "RequestBackup" event like the previous Has Received Event task monitored.

The last difference between the Solo and Team behavior trees is the lowest priority branch, the Search For Target branch:




The team-specific tasks are indicated in red. When an agent goes to search for the target they can either start their own search or join an existing search. If they want to start their own search then the Should Lead Search task will return Success. The Should Lead Search task will communicate with the Team Manager component to determine if the agent should start its own search. If the agent should not start its own search then the Find Leader and V Formation tasks will run. Find Leader will also communicate with the Team Manager only this time it'll return the leading agent that the current agent should create a formation with. As soon as this leading agent has been returned the V Formation task will then have the current agent follow the leading agent in a formation.

## Deathmatch Agent

The Deathmatch AI Agent inherits from the AI Agent component and adds the ability to detect any targets as well as keep a list of available weapons. Any weapon that the agent can use should be added to this Available Weapons list. The behavior tree tasks will use this

list to determine which weapon is best for the current situation.

## Inspected Fields

#### Pause on Death

When the agent dies should the behavior tree be paused instead of disabled?

#### Add to Team

Should the DeathmatchAgent add the agent to a team?

#### Team Index

If *Add To Team* is enabled, specifies the team that the agent should be added to.

#### Max Collider Count

The maximum number of colliders that the character can detect.

#### Look Transform

Optionally specify a transform to determine where to check the line of sight from.

#### Target Bone

If the sight check locates a humanoid, specify the bone that the agent should look at.

#### Distance Score Weight

The amount of weight to apply to the distance when determining which target to attack score. The higher the value the more influence the distance has.

#### Angle Score Weight

The amount of weight to apply to the angle when determining which target to attack score. The higher the value the more influence the angle has.

#### Available Weapons

All possible weapons that the agent can carry. Each weapon has the following fields:

- *Item Definition*: The Ultimate Character Controller ItemDefinition.
- *Class*: Specifies the type of weapon.
- *Use Likelihood*: A curve describing how likely an item can be used at any distance. The higher the value the more likely that item will be used.
- *Min Use Distance*: The minimum distance that the item can be used.
- *Max Use Distance*: The maximum distance that the item can be used.
- *Group Damage*: Can the weapon damage multiple targets in one hit?

# Cover

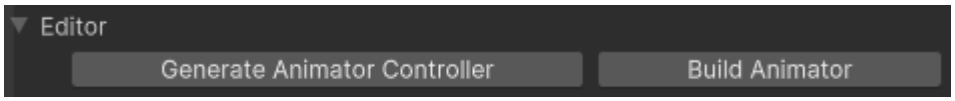
The Cover ability allows the character to take standing and crouching cover. When the character aims they will pop out of cover and aim in the direction of the crosshairs. The Cover ability is a subclass of the Detect Object Ability Base ability allowing it to use that class in order to detect when the character can take cover.

## Setup

1. Select the + button in the ability list under the "Abilities" foldout of the Ultimate Character Locomotion component.
2. Add the Cover ability. The Cover ability should be positioned toward the top of the ability list.
3. Use the *Object Detection* field to specify how the character should detect a cover point. If the character is an AI agent the *PredeterminedMoveTowardsLocation* property can be set.
4. Add the Move Towards Location to the cover object indicating where the character should take cover from.
5. Adjust the *Exposed Distance* to determine which side the character can aim from. The x value indicates the local horizontal offset from the center of the character that the character can aim from. The z value indicates the length of the raycast to detect if an obstructing object was hit. The *Debug Exposed Distance Ray* field can be enabled to visualize the raycast.

## Animator States

The states from the Deathmatch AI Kit demo animator can easily be added to your own character by clicking on the "Build Animator" button under the Editor foldout of the Cover ability. This will add the states to the animator that your character is using.




## Inspected Fields

#### Debug Exposed Distance Ray

Should a debug ray be drawn for the exposed distance?

#### Move Speed

The speed at which the character moves towards the cover location.

#### Depth Offset

The offset between the cover object and the character.

#### Exposed Distance

The distance when determining which side to aim from.

#### Can Take Crouching Cover

Can the character take crouching cover?

#### Crouch Height Parameter

The value to set the Height Animator parameter value to when crouching.

#### Min Auto Cover Switch Duration

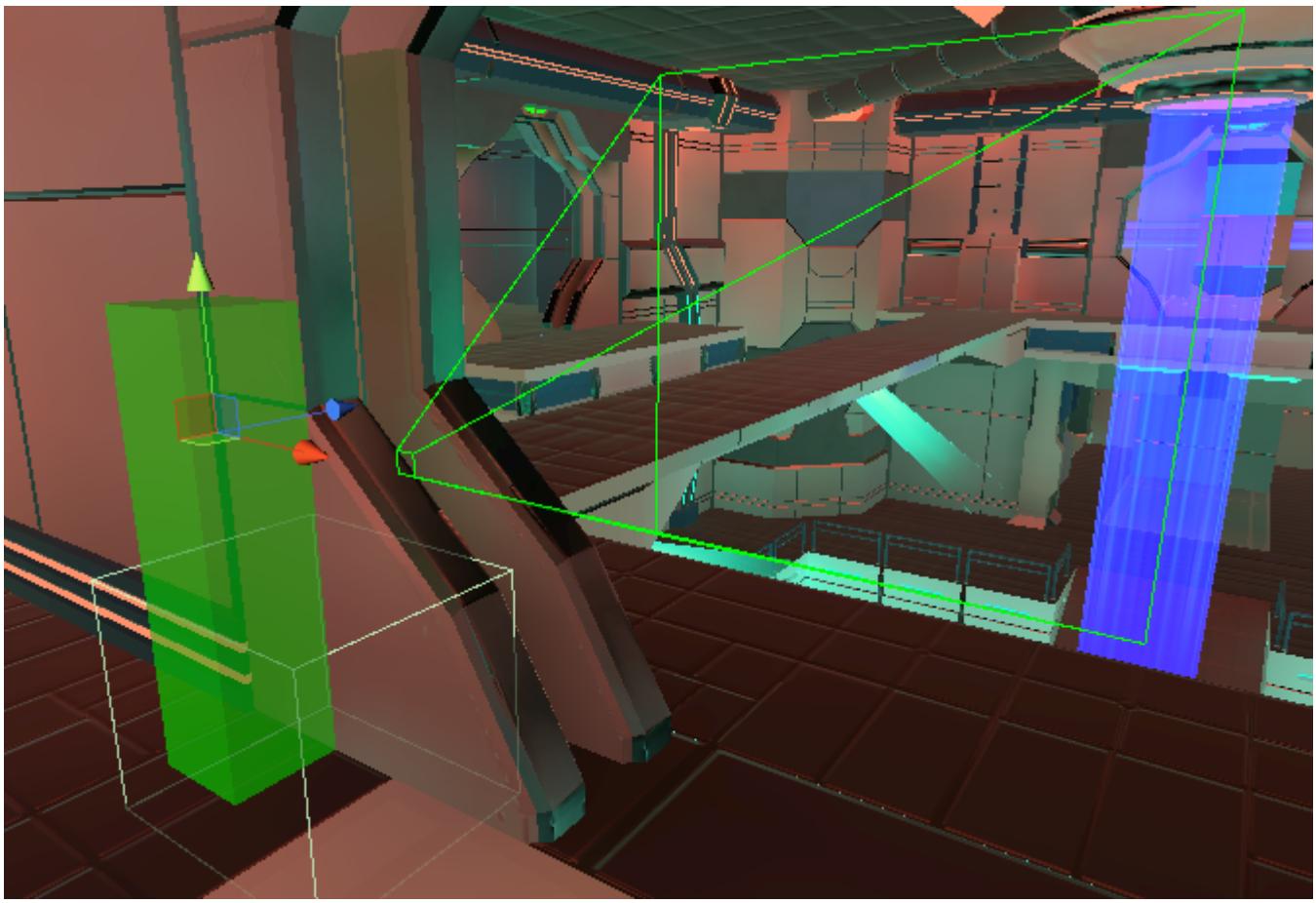
The minimum duration that the character can automatically switch between crouching cover and standing cover based on the cover object.

#### Exposed State Name

The name of the state when the character is aiming while exposed from the cover position.

## Cover Points

When the agent need to find a new cover position they'll look through all of the Cover Points positioned within the scene and choose the best one. These cover points are positioned manually to ensure they are in a good cover position. The Cover Point component will show a uses helper gizmos to make this positioning easier. In the screenshot below, the mostly-solid green block represents the location that the agent can take cover. The camera frustum just to the right of that block represents the area that the agent can shoot from cover. When the agent is within the box outline on the floor the cover point is considered occupied.



If a Cover Point is positioned too far or too close from a wall then the mostly-solid box will turn a red color. This means that the position should be corrected otherwise the agent will not correctly take cover at that location.