Table of Contents

Defining Nodes Documentation

Nodes are relevant Game Objects that permit a variety of specialized game functions. In the case for Squad and Agent behaviors, they further expand and exercise behavior functionality. In this documentation, different necessary nodes are defined to be used by specific behaviors on an individual and group level basis.

Template

Description: Here a brief description about the Node and its purpose will be made
Implementation Summary: Here a summary about the node is made, based on the more detailed descriptions of its components preceding this sub-heading.
Results from Testing:
Known Issues:
Considerations:

AINode Interface & AINodeManager Interface

Defined Interfaces for Node Classes and NodeManagerClasses. The Node Classes act as sensors to update their resulting data to the data collection bank of their respective NodeManager. Upon receiving the data, at intervals, the NodeManager will filter and Update information to the Global BlackBoard for other systems and Agents to use. These Files can be found in the Core folder in the Scripts.AINodes directory in Assets. Other relevant folders, including Nodes and NodeManagers can be found too, along with their base classes that implement their respective interfaces.

Additionally, it may be important for Nodes to specify what types of defined Agents that can utilize them.

PatrolPoint Node

Description: Patrol Points are simply areas that the Agents navigate to investigate, and are chained together with other Patrol Points throughout the map.

Comments:

- When first starting the GameProject and defying the patrol Behavior, PatrolPoints weren't created as nodes, but were their own class.
- There can be further features added to enhance the Patrolling experience. For example, when investigating, rather than just simply ticking off the Patrol Point's public bool parameter "investigated" to be true, also have the agents scan the surrounding area based on some angle parameter.
- Furthermore, need to fix lingering issues with Patrolling Behavior, such as creating a stopping distance from patrol points, based on their radius, so that Agents navigating to the same PatrolPoint aren't colliding into each other. This is a similar issue with Following Behavior in which Agents are colliding with other Agents they're assigned to follow.

Cover Node

Description: This node is a direct copy of cover nodes defined in Monolith's game F.E.A.R. (2005). The cover node has five parameters that determine whether an AI will use a node and what animations will play while at it.

1. **FOV**(field of view): the angle around the direction a node is facing. If the player is within the FOV, then the node can be valid.
2. **Radius or Region**: the area the AI must be within to consider using the node.
3. **BoundryRadius or BoundryRegion**: the area the player must be within for the node to be usable.
4. **ThreatRadius or ThreatRegion**: the area that if the player enters, then the AI will retreat.
5. SmartObject: determines the animations the AI can use while at the node. A child node [GameObject] of the Cover Node [GameObject]

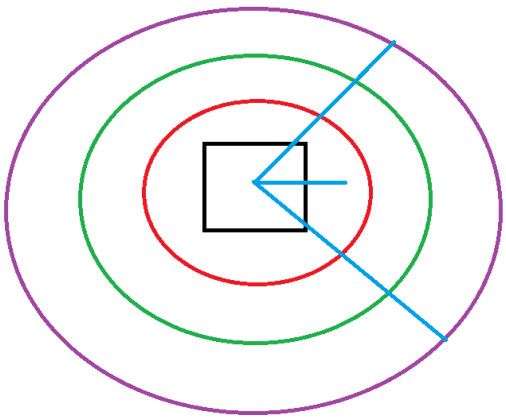

***Fig 1**. The figure above displays a CoverNode and its parameters. The BlackBox represents the CoverNode; the blue representative of the FOV; The green region represents the Region the agent*

needs to be within to consider using the node; the purple radius is the Boundry; and lastly the Red is the Threat Radius/Region that if the player enters, then the agent at the node will retreat.

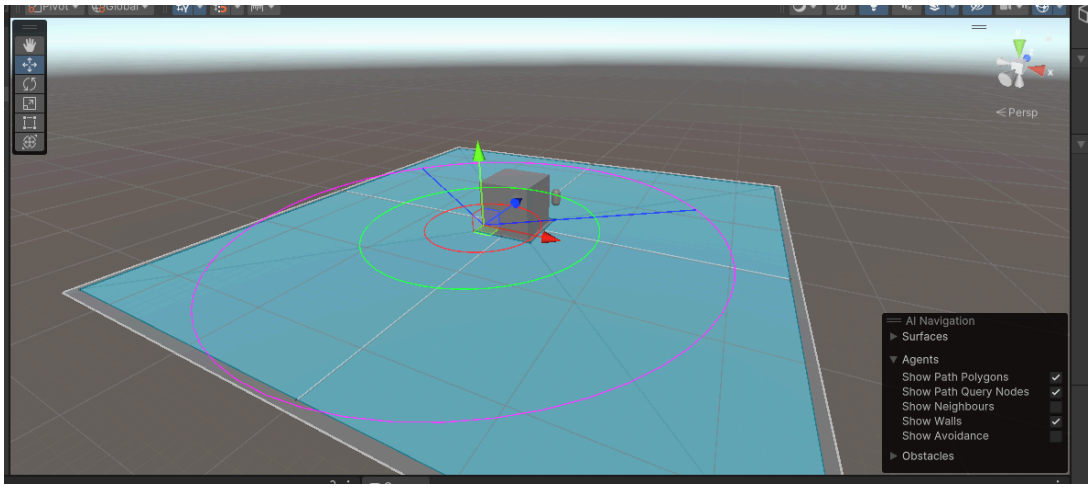

Fig 2. Cover Node in the GameScene

FOV

The field of view is a very similar implementation like the FOV for the Player Sensor (Scripts >> Sensors) which is just the range of angles (0 to 360 degrees). Also, the distance of the direction the cover node will be facing must be manually set.

Presumably, FOV and Boundry Radius/BoundryRegion go hand in hand, as the BoundryRadius/BoundryRegion sets the distance of how far the FOV reaches.

Radius and Region

The radius is the maximum distance from the node an Agent can be for it to consider using the node. Alternatively, developers can specify a Region that the node must be within to be used, and the Agent will use this instead of distance to determine usability.

For the purposes of visually seeing if the Agent considers using cover nodes through some Editor Script similar to Player sensor, we will use Regions, as well as for setting the Boundaries of Nodes and determining when the Agent's life is threatened when the player comes close to the node.

ThreatRadius and ThreatRegion

Either the player comes within the ThreatRadius or the ThreatRegion of the node, and the node will become invalid, and any Agent standing on it will attempt to retreat.

BoundryRadius and BoundryRegion

The BoundryRadisu is the distance the player must be from the node for it to be considered valid by other Agents. Alternatively, a BoundryRegion can be used, in which if the player is within region specified by BoundryRegion the node will be considered used.

Again, presumably this goes hand-in-hand with the FOV, and this implementation makes sense, since it wouldn't make sense to take cover at some cover node if an enemy isn't detected within that region/radius of the cover node. Also, having a direction for the node's FOV will prevent Agents taking cover at nodes in which the player can visually see their behinds.

SmartObject

There will be a child SmartObect inside the Cover Node that determines which animation the AI can play while using the node. In F.E.A.R., there are four SmartObjects regarding cover that Agents can use: CoverDuck, CoverDuckPeek,CoverDuckStand; and CoverStep. The CoverDuck, CoverDuckPeek, and CoverDuckStand smart objects allow Agents to duck behind cover, peek ove cover while ducking, and stand higher while still ducking over cover, respectively. These three CoverDuck SmartObjects permit the Agent to shoot over cover while crouching if the agent were to see the player behind waist-high to chest-high level cover. Meanwhile the CoverStep SmartObject has the Agent step to the side and shoot around a corner [if the player is visible].

For the purposes of this Game Project, we will primarily focus on implementing the **CoverStep Smart Object**. In one of F.E.A.R. 's development document about A.I. combat, where most of the information about their nodes has been pulled from, the CoverStep Smart Object has two directions it considers: the direction the AI will step towards, indicated by a red arrow; and the direction the CoverStep Node [SmartObject] faces, indicated by a blue arrow.

The direction the CoverStep Node/SmartObject faces is the direction that the Agent will face when using the Node. Also, think of the SmartObject nodes as children of whatever Cover Node they are found in. Figure 3 demonstrates a rough diagram about how the CoverStep Node/Smart Object will appear in the GameScene:

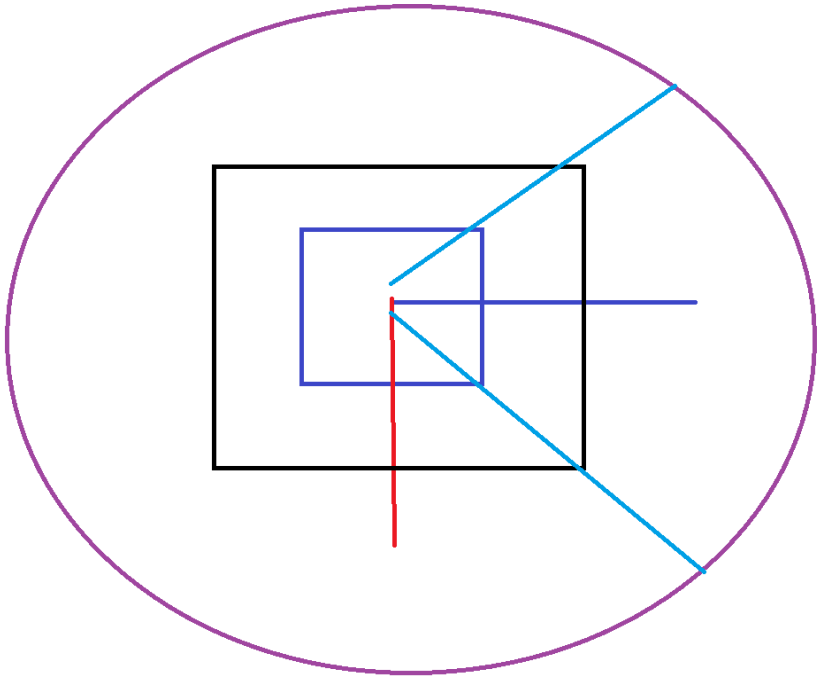

Fig 3. *The above figure features a rough sketch of a CoverStepNode, represented by the innermost blue box, inside a CoverNode represented by the outer black box. The blue line from the CoverStepNode represents the direction the Agent will face and the red line represents the direction the Agent steps to, if it were to use the CoverStepNode SmartObject. The FoV of the CoverNode is independent from the lines of the CoverStepNode, which permits a variety of customization.*

Implementation Summary:

Similar to PatrolPoint Nodes, Cover Nodes will be made as a prefab with a transform position of Vector3(0, 0, 0) initially, as developers can change this however they please. Very similar to the Player Sensor, the BoundryRegion/BoundryRadius and the FOV will be implemented in a similar fashion, and the Cover Node will have a UpdateToManager() method in which it will update itself, UpdateToManager(this), if it's Valid.

A Cover Node class must have the following instances:

- Private bool instance that determines whether the Node is Valid or not
- Public float instance that can be modified to adjust the FOV
- Public float instance to adjust the Radius/Region for any AI to consider using the Cover Node
- Public float instance to adjust the BoundryRadius/BoundryRegion
- Public float instance to adjust the ThreatRadius or ThreatRegion
- A private list, List<ReGoapAgent<string, object>>, of nearby Agents within Radius, only if the Cover Node is valid, otherwise this List is empty.
- Layer Mask to detect specific targets, such as the player, agents, and obstacles
- Etc.

At intervals in an infinite loop, the Cover Node will check within its FOV and BoundryRadius for the player, and if the player is detected, then the code will enter a sub infinite loop. In the inner loop, enact the same FOV check again, as the node will become invalidated if the Player is no longer within the FOV and BoundryRadius. Additionally, the ThreatRadius will check for the player too, as the player entering will also invalidate the node. If the FOV check is false or the ThreatRadius check is true, then the node is invalidated, otherwise, update the list of nearby agents, and this list must always be cleared before updating. Here is a quick mock up:

while(true){

- Some Time Delay
- FOV Check Routine
    - If Valid
        - while(true) {
        - FOV Check Routine
        - Threat Check Routine
        - If !Valid {Break;}
        - Else {Check For Nearby Agents}
          }

}

To manage Cover Nodes, there should be a Cover Node Manager that holds instances of every Cover Node in the current GameScene. Similar to PatrolPoint Manager, it can have a public list of CoverNodes that can be filled in the Inspector Window of the Cover Node Manager component. Unlike the Patrol Point Manager, in terms of functionality, the Cover Node Manager acts more as a Memory Bank, while its Cover Nodes are the sensors. The Cover Nodes try to update themselves to the Cover Node Manager with an associated bool value on whether or not they're valid. The CoverNodeManager is then responsible for filtering out what CoverNodes are Valid, then updates them to the Global BlackBoard in the GameScene.

Further adding to CoverNode functionality is the SmartObject child to the COver Node that the Agent occupying the Node can use. Lastly, agents should be able to make reservations when heading to a Cover Node, so that other Agents do not plan or fail to navigate to a Cover Node that is already reserved, and, or occupied. In Conclusion, Cover Nodes are responsible for the following:

- Detecting nearby Agents, so that other Agents can detect what Valid CoverNodes are nearby through some Sensor class (i.e. CoverSensor)
- Validating and Invalidating itself based on checks regarding player detection in FOV and ThreatRadius

- Permitting Specific Actions/Animations through the use of SmartObjects available to any agent occupying its space
- Responsible for holding reservations made by Agents, so that other Agents do not consider the CoverNode in their plans, or other Agents fail to navigate to it.
- [Added Feature] An Agent could occupy the space and adjust the parameters based on their Health Status, and upon leaving the space or being deceased, the CoverNode's parameters will reset.

Meanwhile, the CoverNodeManager's responsibilities are:

- Updating Valid CoverNodes to the Global BlackBoard in the GameScene, and overwriting pre existing ones with new ones, even if no CoverNodes are validated.

Results from Testing:
Known Issues:
Considerations:

Inside the F.E.A.R. document about AI combat, it reads that the "BoundryRadius value can be used to [affect] how close enemies will try to get to the player during combat", since the BoundryRadius affects how far from the player the node must be before it is considered valid". The significance of this is that Agents will navigate to cover that is more closer to the Player with a lesser BoundryRadius. This is further enforced if the [Consideration] Radius were the same or larger, thus demonstrating a more suicidal aggressive behavior, and that is especially true if the Threat Radius were minimized, thus decreasing likelihood for Agents to retreat from their cover unless the Player gets really closer.

A Game Manager that can control Aggressiveness may want to utilize these functionalities of CoverNodes. The Greater the BoundryDistance and the Greater the ThreatRadius, then the more cautious the Agent will play. Meanwhile the lesser the Boundry Distance, and the Lesser the Threat Radius, and the Greater the [Consideration] Radius, then the more Aggressive the Agent is.

Besides adjusting these CoverNode parameters with a Game Manager, **perhaps the Agent itself can**. For example, if an Agent is badly Damaged, then they may adjust the parameters accordinging to their health. Additionally, if the Agent were to die or leave the space, then the CoverNode will reset itself to its default settings.

The F.E.A.R. CoverNode also has a Dependency property, in which the AI must visit first before visiting this cover node. ALso, the CoverNode has a IgnoreDir bool property that if ticked, then the AI will ignore the FOV Player detection. Also there is a bool property called ThrowGrenades in which the Agent will throw grenades from this CoverNode if this is ticked.

Ambush Node

Description: Similar to cover nodes, the ambush node is a direct copy from the one in F.E.A.R. Reportedly, Ambush Nodes in F.E.A.R. have two uses:

1. Act as locations that developers want Agents to navigate/coordinate to and fight from, but to NOT use any CoverNode animations.
2. Locations behind corners or other cover, especially good ambush points, that developers want Agents to wait for the player to come into view before firing.

Implementation Summary:

The Implementation of the AmbushNode is the same as the CoverNode class with the only difference being the new two float parameters: MinExpiration and MaxExpiration. These are the minimum and maximum amounts of time (in seconds) that an Agent will wait at these nodes while waiting for the Player to appear. The default values are 3 and 20 seconds, so the Agent occupying the node will wait for at least 3 seconds for the Player to come into view before deciding on whether to move. If the player has still not appeared after 20 seconds, then the Agent will move to a new location.

Since the Ambush Node is implemented the same way as the CoverNode, Agents will only consider it if it's Valid, meaning the Player is within its FOV and Boundry radius. With the purpose of this being an Ambush Node, if set at around a corner, the FOV can ignore obstacle obstructions. This means that if an Agent that considers a Valid Ambush Node set around some corner, then the Agent is technically cheating knowing the Player could show up around the corner.

The described example can be akin to the Agent hearing the Player nearby, but if nothing happens after 20 seconds then it will move given a different behavior goal. Or it could move between the 3 and 20 second mark depending on changes in the GameScene, like the Player no longer being detected by the AmbushNode, or if the Agent is responding to an Order via some Goal of higher priority, etc.

Results from Testing:
Known Issues:
Considerations:

- If an Agent has nowhere else to go, meaning no valid cover that isn't present, reserved, or occupied, then perhaps they can go to a nearby Valid Ambush Node instead. Recalling

the first use of Ambush Nodes in the description, they can act as locations where Agents can fight from without using any CoverNode animations. This allow developers to stack Ambush Nodes relative to Cover Nodes, so one Agent may shoot from cover and another shoots closeby. A word of caution to developers, **DO NOT** place ambush Nodes relative to Cover Nodes in which Agents could accidentally commit **friendly fire**.

- Should probably have a **Reactivation time** like the Ambush Node in F.E.A.R. which is the number of seconds before any other Agent can use the node. This prevents other Agents from immediately using the Ambush Node right after, let's say if an agent occupying the space just dies. That would be pretty unconvincing behavior.

SmartObjects & Interactions

Considering CoverNodes, the related SmartObjects, for example, CoverStep possess instantaneous interactions that animate the Agent that performs the interaction. If the Agent performs an interaction, then it must pass an instance of itself to the interaction, so that the interaction can call the Animator to animate the interaction's held Animate State (via a string), and this Animate State handles itself with the Agent performing the interaction.