AI Cover System for Videogames

Dell'Acqua Matteo 09488A

January 17, 2023

Summary

1 Introduction

The objective of this project is to build an AI capable of engaging the player in a firefight using the environment to its advantage by searching for available covers from which to attack.

The project is divided in two components:

- **Cover master**: handles the cover system and stores information on which agent is using which cover;
- **Cover agent**: handles a single agent and its interactions with the environment and the player.

1.1 Delegates

In order to abstract from the movement implementation in the game, both components will make use of a delegate class that must be implemented by the user in order to link the decision-making of the project to the movement component.

In particular it requires the user to specify how to transform a cover point to a real point in the world terrain, how to move to a location, determine if a location is reachable from a specified point, etc.

1.2 Implementation

The project is implemented in Unity using C# scripts. Graphics and UI are ignored as both are not strictly part of the project.

2 Defining Covers

The first decision to make is how to define a cover.

2.1 Cover types

For this project a cover is generally defined as a point in the game world that satisfies certain criteria. In particular two types of cover are defined:

- Wall covers;
- "Sandbag" covers.

2.1.1 Wall Covers

A wall cover is defined as a point in the game world that completely covers the agent from the player but allows it to hit the player with a small movement to the side, as shown in Figure 1.

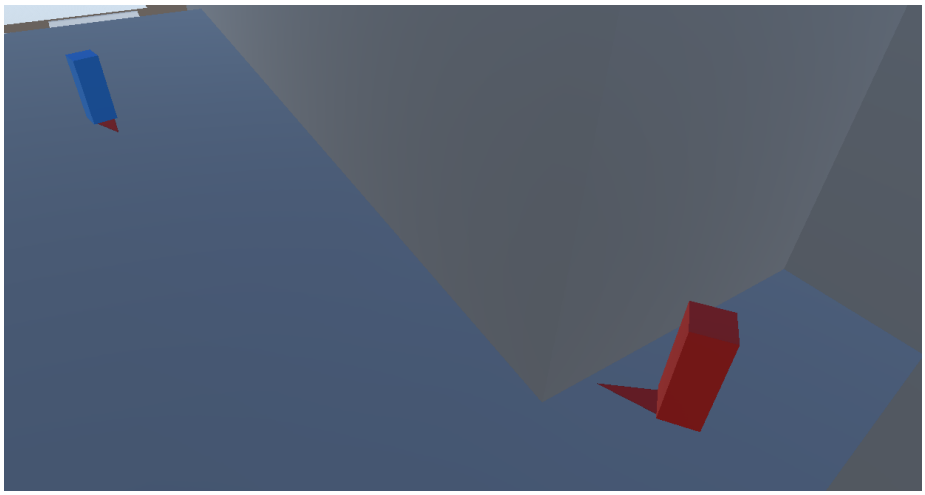

Figure 1: Example of a wall cover

2.1.2 Sandbag Covers

A sandbag cover, as the name suggests, is a cover meant to mimic the type of cover created with sandbags by the military. In particular it is defined as a point in the game world that completely covers the agent when crouched but allows it to attack the player when standing, as shown in Figure 2.

3 Cover Master

The first component of the project is the master part of the cover system.

3.1 Cover points generation

During the initialization phase its function is to generate all the possible cover points in the map and store them. In order to do this it requires the user to specify two parameters:

- Distance between cover points;
- List of Vector3 representing the agents spawn points (spawn mechanics are NOT handled by the cover system. This is only to avoid generating points for unreachable or uninteresting areas).

3.1.1 Distance between cover points

A float specifying the distance between each cover point. This should be set to the highest value that guarantees an acceptable result. The higher the number, the more points will be calculated, thus increasing the performance cost of

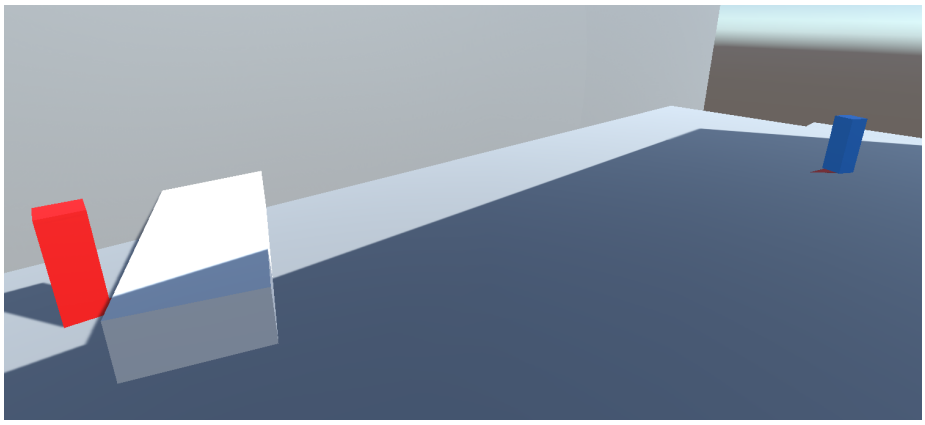

Figure 2: Example of a sandbag cover

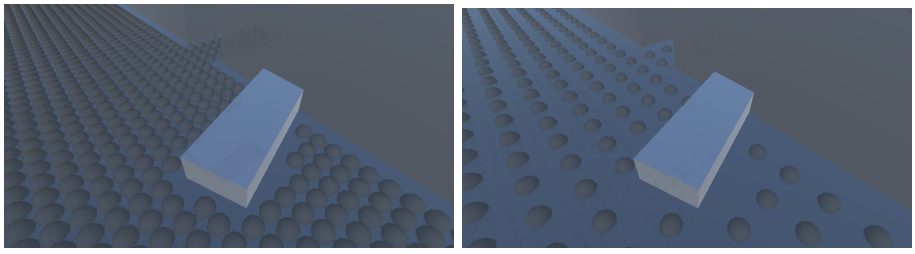

Figure 3: Example of point generation with distance set to 1 (left) and 2 (right)

finding a suitable cover. Figure 3 shows an example of point generation with different distance values.

3.1.2 Spawn points

A list of Vector3 to limit point generation to the interesting areas. The only use of this list is to specify the starting points from which the system will generate all the other points. It will not actually serve as a spawner for the agents.

3.2 Cover calculation

Every time an agent requests for a cover, the master will take the agent's position and heights and the player position to give back to the agent its new cover (if found, null otherwise).

Priority is given to sandbag covers, in particular to those that cover more of the agent without blocking it from shooting while standing. In general, the steps are as follow:

1. Iterate through all the points between a certain distance from the player

and add them as possible sandbag covers or possible wall covers if not already occupied by another agent;

2. If there is at least 1 possible sandbag cover, return the best possible sandbag cover (where the value of a sandbag cover is determined by how much of the agent is covered without impairing its ability to attack the player while standing);

3. If there is at least 1 possible wall cover return the best possible wall cover (where the value of a wall cover is determined by the distance from the cover point to the covering wall, the less the distance, the higher the value).

4. If no possible cover is found, return null to notify the agent that no cover is available.

3.3 Cache

In order to avoid continuous calculations, a cache is used to store agent-player positions from which no cover is available. To do this efficiently, the user has to specify an additional parameter called *distanceBetweenCachePoints*. This parameter controls the cell size of the cache grid. When a cover is requested, a first check is performed on the cache, rounding the position to the cache grid. If the agent-player position is in the cache, then no cover is available and no other calculation is needed. When no possible covers are detected from a request, the agent-player position is added to the cache. The parameter's value should be kept as high as possible while still maintaining acceptable results to reduce the number of cache cells.

3.4 Delegates

The master part of the system requires the user to implement the 3 abstract methods contained in the abstract class *CoverDelegates* in order for the system to work:

- *public Vector3 GetPositionOnTerrain(Vector3 position)*;
- *public bool IsReachable(Vector3 startPosition, Vector3 endPosition)*;
- *public bool IsOnTerrain(Vector3 position).*

For the project, a Navmesh-based implementation is provided.

4 Cover Agents

The second part of the system directly controls a single agent. It is implemented as an Hierarchical Finite State Machine (HFSM) and determines how the agent interacts with the environment and with the player. With regards to parameters, the user has to specify the following:

- *AITimeFrame*: how long between AI updates;
- *rangedAttackRange*: range from which the agent should start attacking the player with ranged attacks;
- *meleeAttackRange*: range from which the agent should start attacking the player with melee attacks;
- *meleeChargeRange*: range from which the agent should charge the player to get in *meleeAttackRange*;
- *gunHeightComparedToPivot*: height from which the agent shoots while standing from the pivot;
- *gunCrouchHeightComparedToPivot*: height from which the agent shoots while crouching from the pivot;
- *pivotHeight*: height of the pivot of the agent from the terrain.

4.1 HFSM Design

As shown in Figure 4, a first level of FSMs are an *Alive State* and a *Dead State*. If the agent is in the *Alive State*, then another FSM takes over to control how to react based on the distance from the player. If the player is less than a certain distance from the agent, the *Melee FSM* will have the agent charge the player to attack it in melee. If the player is more than a certain distance from the agent, the *Move Towards Player* state will have the agent walk towards the player. Otherwise the *Ranged FSM* takes control and handles the ranged part of the firefight. In order to do this 3 states have been defined inside the *Ranged FSM*:

- *Not in Cover*
- *In Cover*
- *Going to Cover*

The *Not in Cover* and *In Cover* states are actually other nested FSMs. The *Not in Cover* FSM tries to attack the player with a ranged attack, otherwise it will move towards it. The *In Cover* FSM runs a cycle of exiting from cover, attacking, returning to cover and staying in cover.

4.2 HFSM Implementation

In order to implement the HFSM an *FSMState* and an *FSMTransition* abstract classes have been defined. With regards to nested FSMs, an additional class *FSM* has been subclassed from *FSMState*.

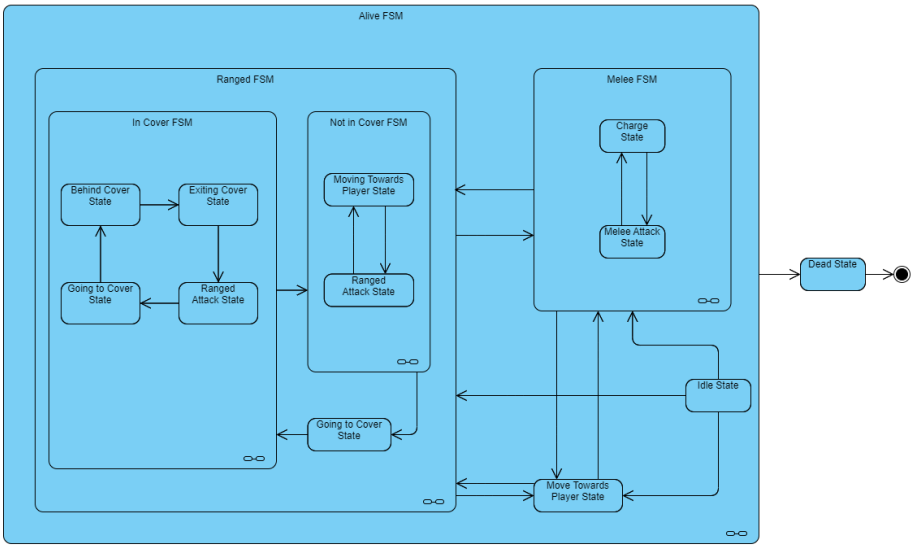

Figure 4: Diagram representing the structure of the HFSM

4.2.1 FSMState

In order to implement an *FSMState*, the following methods must be implemented:

- *public void OnEntry(CoverAgent agent)*;
- *public void OnExit(CoverAgent agent)*;
- *public void OnStay(CoverAgent agent)*;
- *public void OnInterrupt(CoverAgent agent)*.

For more information on *OnInterrupt* see 4.2.4.

4.2.2 FSM

An *FSM* is a subclass of *FSMState*. It has no abstract methods, but they can be overridden to add required functions with *OnEntry* and *OnExit*. *OnInterrupt* and *OnStay* should not be overridden or, at least, always be sure to also call the base method. The *OnStay* method is already overridden to update the FSM. State and transition definition should happen in the constructor using the *AddLink* method of *FSMState*s instances with the following signature: *public void AddLink(FSMTransition transition, FSMState nextState)* and initializing the *currentState* field to the starting state of the FSM.

In general, if a transition fires, the update algorithm follows the following steps:

1. Call the *OnExit* method of the current state;
2. Call the *TransitionActions* method of the fired transition;
3. Change the current state to the new state;
4. Call the *OnEntry* method of the new state.

4.2.3 FSMTransition

In order to implement an *FSMTransition*, the following methods must be implemented:

- *public bool FireTransition(CoverAgent agent)*;
- *public void TransitionActions(CoverAgent agent)*.

4.2.4 Interrupt nested FSMs

With HFSMs, an additional issue has to be considered: correctly exiting from nested FSMs. To ensure that exiting from an FSM won't leave inconsistencies, an additional step is performed before the algorithm presented in 4.2.2: if the current state is an *FSM*, call its *OnInterrupt* method.

The *OnInterrupt* method is already overridden in the *FSM* class to propagate the call, while it must be defined for every leaf *FSMState*. It should contain all the instructions to safely interrupt execution and leave the state.

4.3 Delegates

The agent part of the system requires the user to implement the 9 abstract methods contained in the abstract class *CoverAgentDelegates* in order for the system to work:

- *public void MoveTo(Vector3 targetPosition)*;
- *public void Crouch()*;
- *public void Stand()*;
- *public void Charge()*;
- *public void RangedAttack(Gameobject target)*;
- *public void MeleeAttack(Gameobject target)*;
- *public void Die()*;
- *public bool IsDead()*;
- *public float DistanceBetween(Vector3 startPosition, Vector3 endPosition)*.

For the project, as before, a basic Navmesh-based implementation is provided.

5 What could be improved

While the project works, there are different opportunities for improvement:

- Optimize point generation by eliminating points that can be considered out in the open;
- Optimize cover calculation in order to reduce the number of raycasts (cause of the in-game stutters). Maybe a second cache to memorize if the player can be hit from a point could be stored;
- Allow the environment to change at runtime (and avoid recomputing all the cover points).