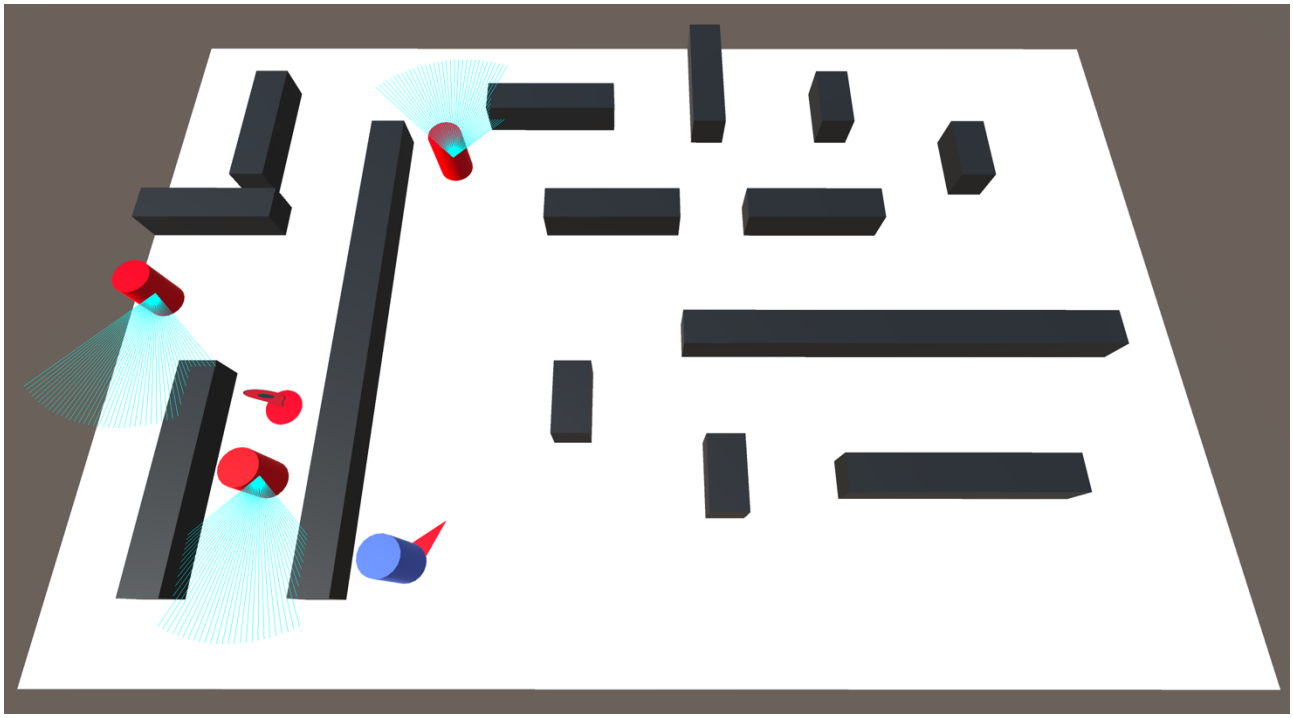
Artificial Intelligence for Video Games
-
Project

Student: *Nicola Sangalli* Student number: *964954* Academic Year: *2021/2022*

Index

1) The project in a nutshell

The aim of this project is to implement an agent capable of reaching a target through a map, avoiding obstacles. The obstacles number, size and position are generated randomly. If the agent enters in the cone of vision of a hidden sentinel, he searches for a cover position outside the sentinel's vision and then he tries to resume the previous path to the target (only if the target isn't in the sentinel's area). When the agent reaches the target, this is moved to another position in the map. If there isn't a path between the agent and the target (a path without entering a visible sentinel's vision) the agent stops moving.



The agent (in blue), the sentinels (in red) and the obstacles (in black). In this case the agent enters in stuck state because he cannot reach the target without entering the sentinels' cone of vision.

2) Map and graph generation

The data structure *mapArray* is a bidimensional array that represents the whole map. The possible values of a cell of *mapArray* are:

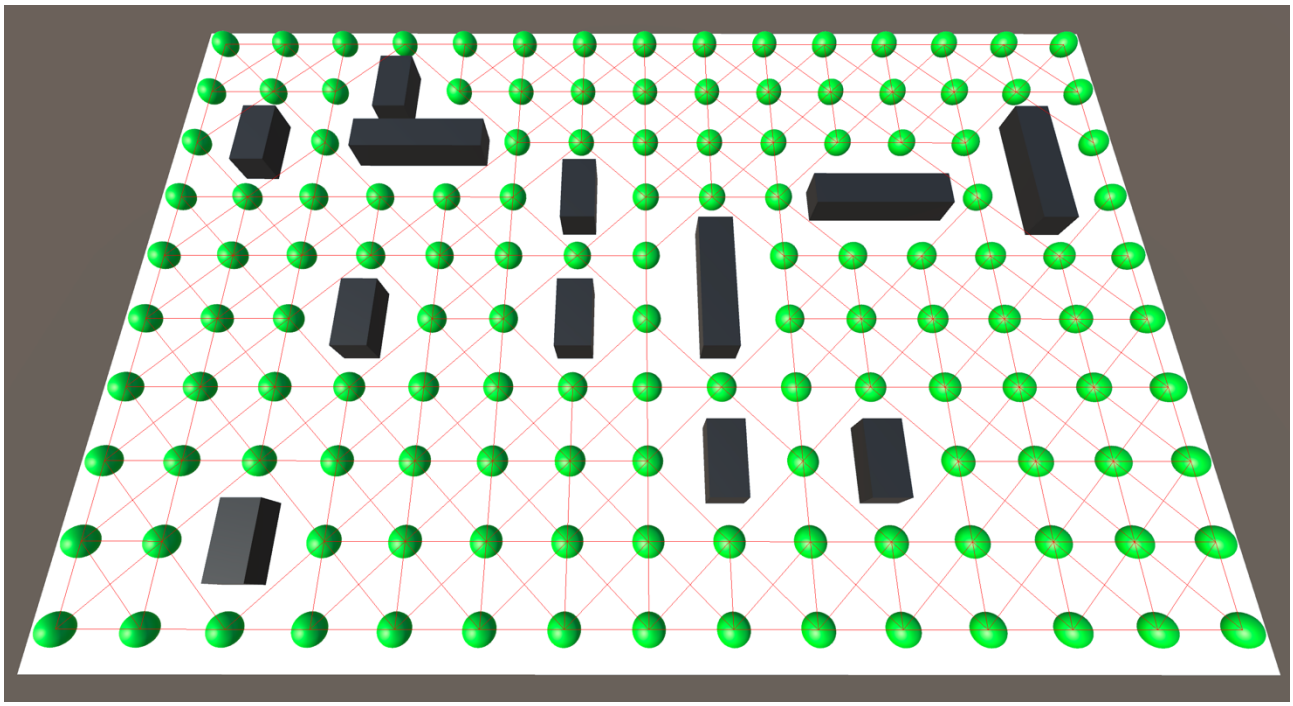
- *0* → the cell is empty;
- *1* → current agent's position;
- *2* → position of an obstacle;
- *3* → sentinel's position;
- *4* → position of the target;
- *-1* → position not available anymore;

When the execution begins, the program must generate the obstacles. First of all, *mapArray* is populated with value *1* (with a certain *obstacleProb* probability). After that *mapArray* is scanned for row and for column: contiguous values of *1* are used to calculate the size and position of the obstacles before instantiating them. All the *1* values are converted to *2*.

The graph, used for pathfinding, is generated scanning *mapArray*: if a cell is empty (*0*) a node is created. The neighbors' position (*N, NE, E, SE, S, SW, W, NW*) are used to generate the edges.

The edge weight of a neighbor in the *N* or *E* or *S* or *W* position is equal to *1*, otherwise it's equal to $\sqrt{2}$.

I've used the *Graph* library seen during the course. I've added three new attributes to the *Node* class: the integers *i* and *j* (used to localize the node in the *mapArray* data structure) and the boolean *inSentinelView* (used by a sentinel to mark the nodes in his cone of vision). In the *Graph* class I've added a function to remove a specific node from the graph and a function to retrieve a node by its *GameObject* instance.



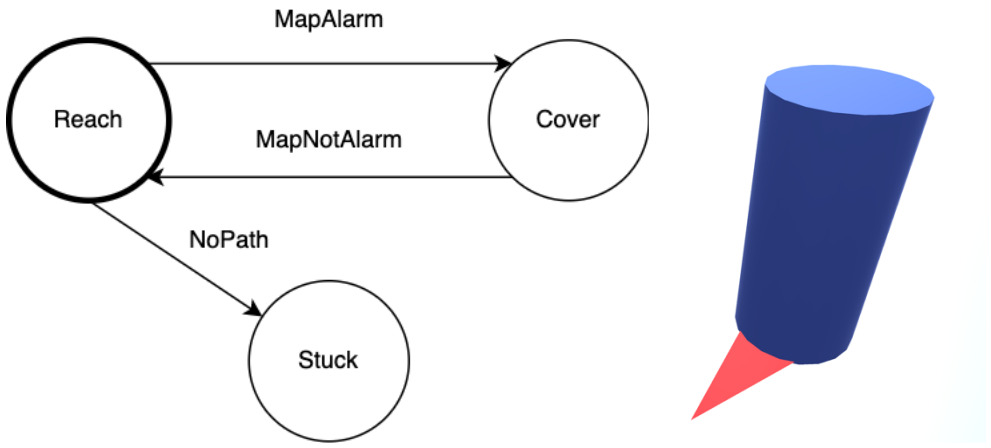
Graphic representation of the graph

The last step of map generation is to instantiate the sentinels, the agent and the target (also updating *mapArray)* trying to maintain a certain distance between them.

3) Finite state machines

The behaviors of the 3 entities involved (agent, sentinels, map) are implemented using the *FSM* technique.
The schemes of the finite state machines are presented below (the bold circle means the starting state).

Agent



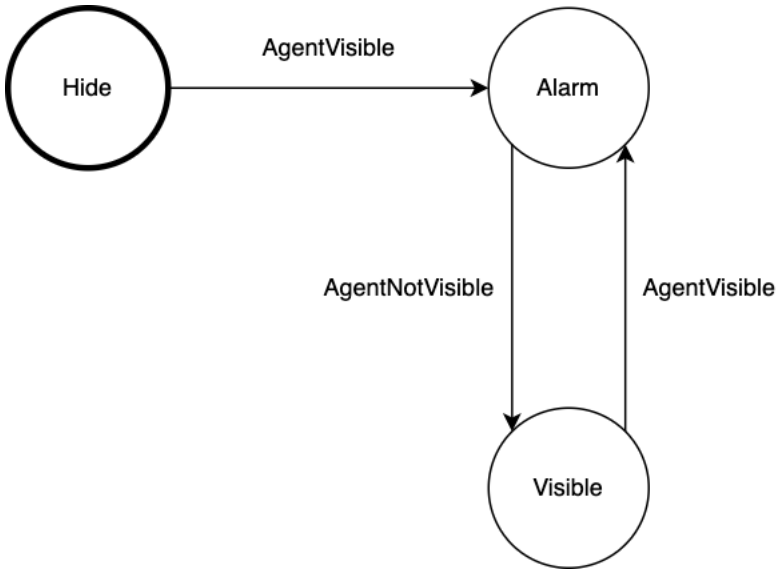
States:

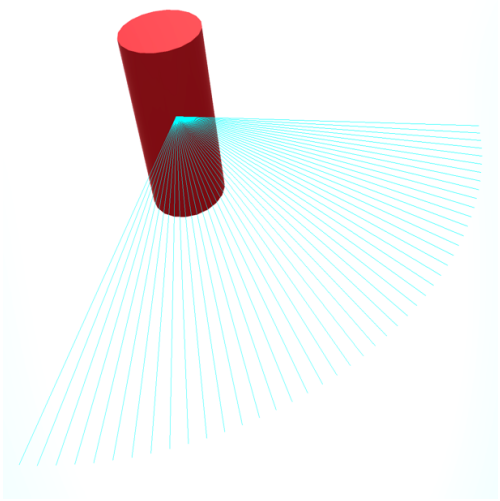
- *Reach* → the agent calculates the path from his current position to the target using the *A* algorithm* with *Manhattan estimator*. I've decided to use this estimator because the map/graph has a regular structure like a chess board. When the path is calculated, the agent follows the trail to the target using the *kinematic movement algorithm* seen during the course (I'm not interested about physics simulation in this case). If the target is reached, his position is moved randomly to a free position in the map and the agent recalculates the path;
- *Cover* → the agent searches the nearest free position in the map, outside the sentinel's vision. If there are more than one possible position, the agent chooses one of them randomly. Now the path from current position to cover position is calculated and, like in the *Reach* state, the agent moves to the target using the *k-movement algorithm*. Before leaving this state, the graph is updated: all nodes and related edges inside the sentinel's vision are removed (the agent couldn't use these nodes anymore);
- *Stuck* → the following message is displayed: "*The target is not reachable avoiding the sentinels!*";

Transitions:

- *MapAlarm* → it's true when the map active sentinels are > 0, false otherwise;
- *MapNotAlarm* → *MapAlarm* complementary;
- *NoPath* → it's true when there isn't a path between the agent's current position and the target, false otherwise;

Sentinel





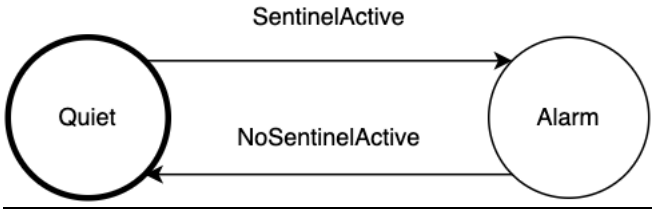
States:

- *Hide* → when the sentinel goes from this state to the *Alarm* state, he shows himself;
- *Alarm* → the sentinel marks all the nodes in his cone of vision (the same nodes that the agent removes later from the graph) and informs the map that he's in *Alarm* state;
- *Visible* → the sentinel informs the map that he is no longer in *Alarm* state;

Transitions:

- *AgentVisible* → it's true when the agent is inside the sentinel's cone of vision or he's touching him, false otherwise;
- *AgentNotVisible* → *AgentVisible* complementary;

Map



States:

- *Quiet* → the directional light of the map is turned on;
- *Alarm* → the directional light of the map is turned off;

Transitions:

- *SentinelActive* → it's true if the current number of sentinels in alarm state is > 0, false otherwise;
- *SentinelNotActive* → *SentinelActive* complementary;

4) Project execution

To run the project, we need to load the *MainScene* file located in *Scenes/* folder.

For the *Map* game object in the hierarchy, we can change the number of sentinels to instantiate (from 0 to 3).

For the *Sentinel* prefab (located in *Prefabs/* folder) we can change the following parameters:

- *FOV* → field of view in degrees (from 90° to 120°);
- *Distance of view* → (from 1 to 4);

The sentinels' cones of vision are implemented visually using the function *Debug.DrawRay()*, so we must activate the *Gizmos* to see them.