CLAUSTROPHOBIC NPC

Katia Cracco

PURPOSE	This document should explain the implementation choices, as well as the behaviors and techniques applied to obtain the required result.
CREATION DATE	03/08/2021
CURRENT OWNER	Katia Cracco
LAST MODIFICATION	18/01/2022
SOFTWARE	1. Development: Unity 2020.3.25f1 - JetBrains Rider 2021.3.2 2. Organization: Pages version 11.2 - GitHub Desktop version 2.9.6 3. Environment: macOS Monterey version 12.1
DATA STORAGE AND BACKUP	The data are saved on a git repository and on an external hard disk.

REVISION HISTORY	
WHEN	**WHAT**
03/08/2021	Create a map with some obstacles
04/08/2021	Insert the agent and the destination
10/08/2021	Study NavMesh Component
13/08/2021	Move the agent with NavMesh
07/09/2021	Create a decision tree
28/12/2021	Generate a new map with random obstacles
29/12/2021	Create a grid
03/01/2022	Apply A* algorithm
07/01/2022	Unify decision tree and pathfinding
17/01/2022	General overhaul and cleanup of unused files
18/01/2022	Documentation

Project specifications

You have a map with a series of randomly placed obstacles and there is an NPC that has to traverse the map.
The path is calculated with A* and the NPC starts walking on it one step at a time.
However, the NPC is claustrophobic, so before taking each step, evaluate with a DT if there are too many obstacles around the point where it would be located (you can choose the policy).
The passage is marked as "invalid" and a new path to the destination is calculated.
Therefore, what the player sees is an NPC that tries to go to the destination, but at certain points it "changes its mind" and decides for an alternative route.

Overview

The first thing inserted into the scene was a plane on which obstacles have been randomly generated. A layer was defined to enclose these obstacles, so that later the agent would be able to recognize and avoid them. At this point, a quad was added on a corner of the map to symbolise the destination to reach and on the opposite corner a capsule was added to represent the agent. Then, the plane was mapped with a grid, so that each tile of the map could correspond to a node of the grid. Using A*, a path was generated starting from the agent node and reaching the destination node. Finally, the agent can start following that route, stopping at each step to check that the next one is valid, that is when there are no obstacles within the 0.7 range. If the condition is valid, then the agent can continue on this way, otherwise it must calculate a new path starting from its current position and reaching the destination. The algorithm ends once the agent is over the target quad.

Designing AI model

In this project, the AI is developed in order to guide the agent in crossing the map.
The two main AI mechanisms that have been used are A* and a Decision Tree. Starting from a basic operation of A*, a pathfinding algorithm, the NPC has been made “sentient” with the addition of a Decision Tree.
In fact, at the base of a good AI for a videogame, there are no complex algorithms, since all the complexity actually lies in the player’s head. As a matter of fact, quite elementary algorithms are usually used and then some randomity is added, giving the player the illusion that the NPC is thinking.
In the case of this project, on the other hand, randomity was not used, but instead the basic algorithm of A* was taken and to reach the destination was made more complicated thanks to a Decision Tree. Therefore, at each step found by A*, the tree is walked so it can be determined whether to continue on that path or, if there are too many obstacles around, recalculate a new path. The player, thus, has the perception of a thinking NPC that realises it is going towards too many obstacles and, being claustrophobic, changes direction.

Map and Components

The map in the scene is a plane, described by different types of coordinates.
First of all, the coordinates given by a Vector3 are used, which refer to the map’s position and its points in the world scene. The map is centred at the position (0,0,0), so it will extend from (-5,0,-5) to (5,0,5).
Then, on this map a grid of the same size was also applied, to create a node on each tile of it. So, when referring to the grid, tiles or nodes, the coordinates in 2 dimensions are used, ranging from (0,0) to (9,9).

On the map, obstacles have been generated. The Fisher-Yates shuffle algorithm was used to make them casual. This algorithm generates a random permutation of a finite sequence. Based on where in the sequence it starts, that is, based on the seed value passed to the algorithm, a particular set of pseudorandom values is selected. Therefore, if you keep the same seed, the series of values generated will always be the same. The sequence mixed by the algorithm was that of the grid’s coordinates, of which the first 10 were taken to generate the obstacles.
It was chosen to use this pseudorandom algorithm and not a random one, because it uses randomly chosen elements that make the scene arbitrary and natural, but at the same time ensures that the same “random” pattern is played when desired. In this way, it was easier to determine if the whole project worked, being able to test it with preset map configurations.
Some seed have been tried:

- 10, 50, 70 - the destination is reachable; the agent walks the path changing direction sometimes
- 30, 85 - the destination is reachable; the agent follows the path but encounters many obstacles: he changes the route several times but always gets stuck, unable to reach the destination

Game objects

❖ Ground

It is a simple GameObject with a green material applied and no scripts attached. The ground is perfectly flat, which does not make it very realistic, but I have not made any changes as the realisation of the terrain was not the main focus of this project.

❖ Agent

It is a GameObject that represents an NPC that has to move around the map to reach the destination. It is a blue capsule that slightly levitates on the floor. Its AI is defined in the *pathfinding.cs* script: the agent must reach the destination via a path calculated with A*. It can be recalculated if the agent is about to find himself surrounded by too many obstacles. Hence, the player sees the NPC go towards the destination and, eventually, change direction in certain points.

❖ Target

It is a simple GameObject with a yellow material applied and no scripts attached. It only serves to represent the destination, which does not move. The program ends when the agent arrives on this flattened cube.

❖ AI

It is an empty GameObject used as a reference for *grid.cs* and *pathfinding.cs* scripts.
First, it generates 10 obstacles, positioned differently based on the value assigned to the seed variable. Then, the map is associated with a grid, so that each tile corresponds to a node which is defined as walkable or not. At this point, a path is calculated from the agent's location to the destination, avoiding not walkable nodes. In the end, the agent walks the path, changing direction in the vicinity of many obstacles.

❖ Main Camera

It is fixed high up on the scene, so that all the obstacles and the path taken by the NPC can be seen.

Scripts

❖ Node

This script define the Node object that has four attributes:

• *walkable* determines whether the node is available or not. A node is not walkable if there is an obstacle in the corresponding position or if the agent finds obstacles within a certain radius from that node. In all other cases, the node is walkable.
• *worldPosition* is a vector in 3 dimensions indicating the node's position in the world scene. The position refers to the centre of the node, hence, since the origin point of the reference system is positioned in the center of the grid, a node can move from -4.5 to +4.5 in the x and z axis, while y is always 0.
• *gridX* indicates the x-coordinate in the grid.
• *gridY* indicates the y-coordinate in the grid.

There is also a *fCost* accessor that returns the sum of *gCost* (node's distance from starting node) and *hCost* (node's distance from end node).

❖ Grid

This script takes as reference:

• the *unwalkable layer*, which contains the obstacles;
• the *obstacle prefab*, used to generate the obstacles directly from the code;
• the *map size;*
• the *seed* value, which indicates the point in the sequence where a particular series of pseudorandom values begins.

In this script there are two main methods: *CreateObstacles()* and *CreateGrid()*.
First of all, the *CreateObstacle* method sections the map into a grid of tiles (size 1x1) and creates a list (*shuffleTileCoords*) of coordinates of the centre of these tiles. This list is mixed with the *ShuffleArray* method, which is an implementation of the Fisher-Yates shuffle algorithm. Then, the method generates 10 obstacles, placing them on the coordinates given by the *shuffledTileCoords* list. These obstacles are included in the obstacles list. Note that in this method there are some commented lines: they will be explained in the "Conclusion and Possible Improvements" section.
At the end, the *CreateGrid* method initialises a node for each grid's tile. Checking the list of obstacles, it determines for each node whether it is walkable or not, in the event that an obstacle is present on that tile.

In this script other methods are also defined, which will be called by the *pathfinding* script.
One method is *GetNeighbours*, which takes a node as input and returns the adjacent nodes.
Another one is *NodeFromWorldPoint*, which transforms the position in the world scene into the grid's coordinates. This means that it maps values between -5 and +5 into values between 0 and 10. There is also the *CoordToPosition* method that works exactly the opposite of this.
The last one is *ModifyNode*, which modifies a node to be not walkable when the agent is close to it but there are obstacles too close.

There is also the *OnDrawGizmos* function that is called every frame. It creates colored cubes on each node:

‣ red, if the node is not walkable
‣ black, if the node is on the path
  Note that if the agent has to recalculate the path, only the last one will be colored.
‣ white, otherwise

When the scripts are executed, these colored cubes are not visible in the Game but only in the Scene. In this way, the player does not notice anything but it is possible to control how the nodes are catalogued.

❖ Pathfinding

This script takes as reference:
• the *agent Transform* and the *target Transform*, which are the transforms of their GameObjects
• the *ray* value, which is the ray used by the agent to detect the obstacles.
In this script there are two main parts: the *FindPath* method and the building of the Decision Tree.
In the *FindPath* method, two lists are defined:
‣ *open*, it is the list of nodes to be evaluated
‣ *closed*, it is the list of nodes already evaluated
Starting from the nodes in the open list, the method selects the node with the lowest *fCost*, removes it from the open list and inserts it in the closed list. At this point, it checks, for all the neighbours of the node, if the cost to reach a neighbour, passing through the current node, is less than the neighbour's current cost or not. If so, it updates the neighbour's costs and marks the current node as its parent: the *gCost* would be the new cost for reaching the neighbour through the current node and the *hCost* would be the distance between the neighbour and the target node, calculated with the *GetDistance* method.
Once the current node corresponds with the target node, the *RetracePath* method proceeds to reconstruct the path, starting from the end node and tracing all the kinship up to the starting node, and then reversing the path.
For what concerns the building of the Decision Tree, the first thing was to define the actions (*TakeStep* and *RecalculatePath*), the decision (*StepValidation*) and to link them. Then the decision was set as the root node for the Decision Tree.
Lastly, *dt.walk* was started in *FixedUpdate*, so that the tree was traversed, analysing the decision and then performing one of the two actions, at every step until the agent arrived at the target.
The *TakeStep* action takes the next node in the path and adjusts the height of that node and of the target node, so that all elements have the same height. Then, the agent looks at the destination and takes a step towards it.
The *RecalculatePath* action sets the node of the next step as not walkable and then recalculates a new path from the actual agent's position to the target position.
The *StepValidation* decision checks if within a 0.7 ray the agent encounters obstacles. If so, the step is marked as not valid.

❖ DecisionTree

This script is the one presented in a lecture.
In this script, an interface that walks the node is defined for all the nodes in the tree, so it is used both to define a Decision Node and an Action Node. Then, a delegate *DTCall* is defined, which will hook external code to both run decisions on single nodes and to perform actions.
In the *DTDecision* class, there is a *DTCall Selector* which is the method to call to make the decision. The return value of the latter is compared with a dictionary that links the return value to the corresponding node, through a key object. Then, a decision node is created and a method for adding a link to an external state is defined. Finally, to walk the node, the selector is called: if there is a corresponding link for the return value, *Walk()* on that link is performed, otherwise it means that there is no action to be performed.
In the *DTAction* class, there is a *DTCall Action* which is the method to call to perform the action, and then an action node is created. In the end, the node must be walked, as the final stage of the decision process, so that the action is performed.
There is a third class, the *DecisionTree* class, that holds the decision structure. So, in this class, a decision tree is defined, with the root node as its starting point. Also a *walk* method is defined: it walks the structure and calls the resulting action that will be performed.

Conclusion and Possible Improvements

Developing this project, some difficulties were encountered in the section where the obstacles were classified as not walkable. In fact, initially this part was implemented using *CheckSphere* on the *unwalkableMask*, but in this way also the tiles adjacent to the obstacle were recognized as non walkable. This happened because keeping a radius of 0.5 already led to the edge of these tiles and so the result was that the walkable part of the map was considerably reduced for no good reason. Then, it was decided to change the approach, keeping a list of the coordinates of the obstacles and checking if the coordinates of the node to be evaluated as walkable were in it.
In some cases, the NPC fails to reach the destination, not because it is physically limited by obstacles, but because the claustrophobia policy does not allow it to pass certain points. In these cases, the NPC continues to look for an alternative path. The implementation could be improved by adding a condition that if the destination has not yet been reached after a certain number of attempts, the agent stops searching for a way. However, to make this improvement there are many combinations to consider, and finding a policy that suits them all is very difficult.
Another change that could be made to the code would be to generate obstacles in a completely random way. In the *grid* script, in the code of the *CreateObstacles* method, there are some commented lines that implement the actual random generation. However, it was chosen to keep the pseudorandom implementation as, otherwise, situations in which the agent would not be able to reach the destination due to the arrangement of the obstacles would often arise. Instead, with the pseudorandom generation, based on the seed values tested, the termination of the algorithm can be seen.