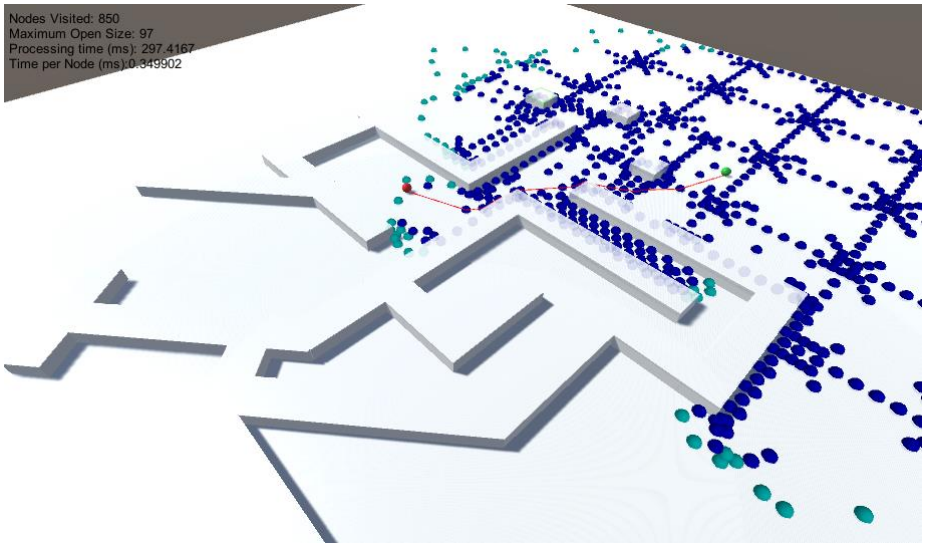
Inteligência Artificial para Jogos

2016-2017

Laboratory Guide

Lab 3 – A* Pathfinding with Navigation Meshes



1. Explore the provided scene and the source code

a. Open the downloaded project in Unity and explore the scene *Pathfinding* . This scene will allow us to explore and test the implementation of the A* search algorithm applied to a navmesh graph. When you click a first time with the left mouse button, a green sphere will appear in the screen. This represents the start position for the pathfinding algorithm. When you press the mouse button a second time, a red sphere will appear on the position you clicked. The red sphere represents the goal position. Once the two spheres are defined the corresponding positions will be sent for the pathfinding algorithm.

b. The project provided includes a free Unity package called RAIN[1]. This package uses a Recast based algorithm to automatically create a NavMesh from the level geometry. In the provided scene, if you click in the object "Navigation Mesh" you will be able to see the created NavMesh for the scene.

c. Analise the code for the PathfindingManager, and the code inside the AIJ.Unity.Pathfinding directory. Take a look at the heuristic function, and at the simple data structure implementation for the open/closed set. Examine carefully the AStarPathFinding class and try to understand what it does.

2. Implement the A* search algorithm

a. Implement the A* search algorithm by completing the Search method inside the AStarPathfinding class. Use the algorithm specified in the theoretical classes. Read the comments in the method to better understand what you need to return. his method should return true if the Search process finished (i.e either because it found a solution or

[1] If the package provided does not run directly, install RAIN from the asset store https://www.assetstore.unity3d.com/en/#!/content/23569 or from Rival Theory website.

because there was no solution). It should return false if the search process didn't finish yet. But for now you don’t need to worry about it. When returning true, the solution parameter should be set to null if there is no solution. Otherwise it must contain the found solution. You also do not need to worry about debug information for now (nodes processed, maximum size for open, etc).

b. Once implemented, try to run the scenario and see the fill and the final path obtained. Can you determine the heuristic function used just by looking at the fill?
c. Now that you manage to implement an initial version, we want to be able to return from the search algorithm after a predefined number of nodes has been visited/processed. Change the algorithm in order to accommodate this feature. Use a value of 10-15 for the number of nodes visited per cycle. Also if the caller wants to receive a partial path, you should calculate the partial path when the search terminates before finding the solution. This is done by calculating the path from the current best node to the start node.
d. Run the scenario again and detect the difference from the previous version.

3. Debug Information and Small Optimizations

a. In order to properly assess and understand the efficiency of the algorithm it is important to provide information about the search process. When performing a search, update the values for TotalProcessedNodes, MaxOpenNodes (the maximum size that the Open list had during the search process), and TotalProcessingTime (using Unity’s time).
b. Implement the Euclidean distance heuristic and initialize the search algorithm with this heuristic function. Look at the fill created by the new heuristic function. See the difference when two points are selected in an open space versus a more closed space.
c. Choose a strategy to solve ties between nodes with the same F-Value, and implement it in your algorithm (you can also change the code provided if you would like).
d. Implement a data structure for the closed set based on hashmap/hashtables. Use this new data structure for the closed set, and see the differences in terms of performance.