A* Algorithms

Tobias Andreas Madlberger
Zementhaufen
HTBLuVA
St. Plöten, Austria
tobias.madlberger@htlstp.at

Fabian Popov
Zementhaufen
HTBLuVA
St. Pölten, Austria
fabian.popov@htlstp.at

Mehdi Jaqubi
Zementhaufen
HTBLuVA
St. Pölten, Austria
mehdi.jaqubi@htlstp.at

***Abstract*—This text is about different path finding algorithms and their performance.**

Keywords—potential, heuristic, threads, node

I. Introduction

Pathfinding algorithms are used to solve problems where you have to find the path between point a and point b, where the two points might be separated from another by obstacles, like walls. Something these algorithms have to do extremely fast, while still finding the most efficient way to travel from point a to point b.

In this paper, we'll be covering the most flexible one of them. It's called A*

II. State of the art

Currently, A* is able to find paths between to points so quickly, they no longer have to be pre backed into games. Multiple pathfinding operations can run on different threads on modern computers and complete their search within milliseconds.

III. Concept

At first, the algorithm needs a space to operate on. The space we'll be using in this paper is two dimensional, having a x and y coordinate, but you can always extent this space into the n dimensional space. You also have need to determine the starting point and the destination. These points might be called point a and point b. Then the algorithm needs a heuristic. Heuristics are being used to determine how close the algorithm got to the destination point.

IV. Implementation

First, the algorithm needs a open set. The set will contain all nodes the algorithm discovered and wants to check. It also needs a closed set, to keep track of the nodes it already visited. To calculate which node is better than the other one, the algorithm needs the g score and f score. The g score is a score representing the cheapest path from start to the current node. The f score is the sum of the g score and the heuristic for the current node. While the open set isn't empty, the algorithm will keep on searching. While the algorithm is searching, it will get the node from the open list with the lowest f score. If the node that got returned by this process is the goal, the goal was found and the algorithm can proceed to reconstruct the path it took. When the node wasn't the node it was searching for, it'll remove it form it's open set and get all neighbours of this node. For each neighbour, the algorithm checks it g score. If the neighbours g score is better than the one that got previously recorded, the neighbour gets added to the open set. In this step you would also implement a check for any other conditions the node has to have to be a valid open node. Example of this would be to check if the node is a wall.

When the open set is empty, but the goal node wasn't reached, the algorithm knowns that there is no path to the destination.

V. Result

Running this algorithm like it was described in the section above, you will already see an big improvement compared to path finding algorithms like DFS or Dijkstra. But to achieve even better performance, you'll have to look at the parts that are the most time consuming. Doing this, will show that the algorithm is most busy checking for the node with the least f score in the open list. This can be improved upon by using a min-heap or priority queue instead of an hash set.

Acknowledgment

This paper wouldn't be possible to be without our teacher, Prof. Mikko Döller, programming teacher in the HTBLuVA St. Pölten, whom explained the a* algorithm to us in great detail.