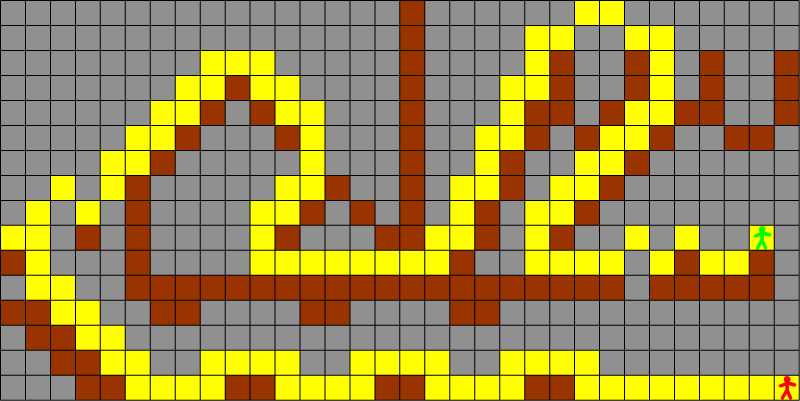
Bradley Pearce | Games Programming | February 29, 2016

Pathfinding

AI Game Programming



# **Overview**

The purpose of this project is to demonstrate a pathfinding algorithm. At a minimum we must show a map on screen, show a start position and allow the user to define an end position. We must then represent the path to the end position. Along with these requirements, we can extend these features by adding a unit to move along the path, different terrain which affects the movement cost to each node and after the character begins to move define a new end position and the path updates.

# **Research**

To get the most efficient and optimum path to a node I will be using A\* as ‘like Dijkstra’s algorithm it can be used to find the shortest path however A\* is as fast as the Greedy Best-First search’ as Patel (2016) put it. Where Dijkstra’s algorithm used only the cost from the current node to find the shortest path, it needs to evaluate a lot of nodes before it find the best path, which isn’t very efficient. The greedy best-first on the other hand uses a heuristic to find the quickest path, but not the best path. A\* uses a combination of both the gcost (Dijkstra’s algorithm) and the hcost (Greedy best-first algorithm) to be both fast and optimal.

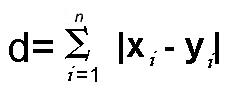
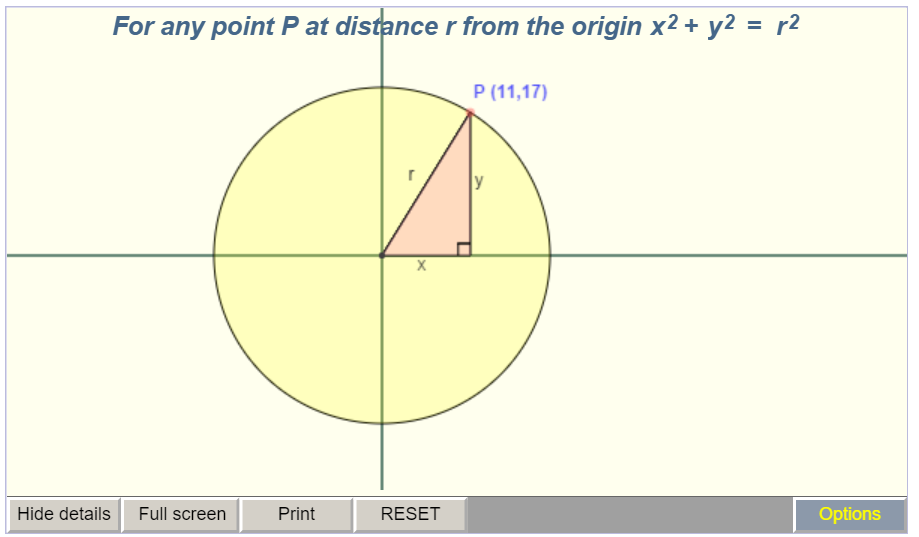
An important aspect of Pathfinding is heuristic calculations. Using A\* requires the use of an admissible heuristic to get the optimum path. ‘A heuristic is admissible if it is a lower band estimate that underestimates the shortest remaining distance to the goal’ as Edelkamp (2012) stated. A\* requires that the optimal path be found through an admissible heuristic as if the heuristic is not admissible, then we can have an estimation that is bigger than the actual path cost from the current node to the target node. If the higher path estimation is on the fittest cost path, the algorithm will not explore it and it may find a non-optimal path to the goal. An admissible heuristic is the Manhattan which is ‘the distance between the current node and the goal node in each axis added together multiplied by the cost of moving a single space’ according to (Patel, 2016).

Fig a. Equation for Manhattan heuristic.

Another mathematical solution to an issue was finding if a point had entered the radius of another point. This would be so we can appropriately move units across nodes and check if the node has entered or left the radius of another node. As Math Open (2009) describe ‘A circle can be defined as the locus of all points that satisfy the equation x^2 + y^2 = r^2 where x and y are the coordinates of each point and r is the radius of the circle’. This means that any value where the unit’s position^2 is lower than node radius’ ^2 must be within the area of the circle. This mathematical knowledge allows me to move units smoothly from point to point without difficulty.

Fig b. An example diagram of how the equation works to produce the radius and area of a circle.



# **Program Description**

The program firstly loads data from a text file defining where each node will be placed and what properties of the node will be set based on that data such as traversal cost, colour and where the node is walkable or not. Loading all nodes from a text file allowed for rapid map creation and design changes based on the data stored in the text file. Then the program creates a grid structure for each node so defining neighboring nodes would be very simple to do, for this I used a two dimensional array as opposed to a vector as it was simpler to define and the program wasn’t going to be prone to errors. The grid structure also allowed for finding nodes using a position on screen was very easy as well. Then we define a pathfinder object which takes in our grid which we can use to find paths to nodes on the grid.

Entering the main loop and draw our tile map and call our grid’s update function where the control of our path finder and input is handled. Using the mouse, the user can pick nodes that are walkable to set as start and end nodes to find a path with. A series of flags controls how each node is differentiated as the start and end node and once both are set the find path function is called once to not take up too much resources each frame iteration. The find path returns a pointer to a container of pointers to nodes so we know which nodes are in the path and the path returned is stored in our grid class.

Handling path finding by returning a container of references to nodes meant that the path could be stored and iterated over rather than relying on the parent of each node. The parent of each node is used to trace back the path once the end node has been found by the find path function but then the node is stored in the path container from start to end node so moving along the path is simple. All of the class designs were done so that interfacing with them was simple and efficient, simply calling the find path function once returns the optimal path to the destination. The simple colour scheme was used to highlight the important elements of the program such as the path and interacting with the program was very simple and effective.

# **Analysis**

Overall, I think my program was very successful in implementing an effective A\* algorithm and the interface provided means that a character can move based on a defined path chosen by the algorithm. My main goal, as it was a real time application, was to make the algorithm as quick as possible to execute. After implementing the algorithm, even if the path is being found every frame the amount of time to retrieve the path is low. This could be due to the small amount of nodes to evaluate given the tile map is quite small compared to a map that may be used in a game level but that’s yet to be tested and proven.

The effect of adding different traversal costs to each walkable node also proved interesting and affected the path taken by the pathfinder quite significantly. In all cases where the path could be cheaper by using a cheaper node, the path chooses the cheaper option, proving the ability of the algorithm quite well.

Unfortunately, when handling input the mouse’s position isn’t always at the exact node it’s pointing to, albeit very close however this can result in finding it difficult to define start and end nodes. The same problem also occurs when moving the unit along the path. The unit doesn’t appear centered on the path and sometimes slightly goes over unwalkable tiles. A small error but the appearance of the movement along the path to be successful should at least be convincing. This may be due to how sprites are drawn in the SFML library where their position is centered at the left corner.

Overall, I think the demonstration of the pathfinder was very successful in demonstrating the A\* algorithm and its use in real time applications.

# **References**

Math Open, . (2009). *Basic Equation of a circle.* Available: <http://www.mathopenref.com/coordbasiccircle.html> Last accessed 29/02/2016.

Patel, A. (2016). *Amit's Thoughts on Pathfinding .* Available: <http://theory.stanford.edu/~amitp/GameProgramming/index.html> . Last accessed 29/02/2016.

Edelkamp, Stefan . (2012). *Heuristic Functions.* Available: <http://www.tzi.de/~edelkamp/lectures/hs/slides/heuristics-slide.pdf> . Last accessed 29/02/2016.

Bevilacqua, F . (2013). *Understanding Steering Behaviours.* Available: <http://gamedevelopment.tutsplus.com/tutorials/understanding-steering-behaviors-collision-avoidance--gamedev-7777> . Last accessed 29/02/2016.

Fig a. <http://www.improvedoutcomes.com/docs/WebSiteDocs/image/diagram_manhattan_distance_metric.gif>

Fig b. <http://www.mathopenref.com/coordbasiccircle.html>