Procedural Generation of Navigation Meshes
In Arbitrary 2D Environments

Timothy Leonard
School of Computing and Mathematics
University of Derby
Derbyshire, United Kingdom
tim@programmingbytim.com

Abstract—This paper describes the benefits of using navigation mesh's over traditional node based graphs for path finding in 2 dimensional game environments, with specific emphasis placed on improving the quality of game behavior. We demonstrate several different methods of smoothing the resulting paths using the benefits that navigation meshes provide, that are impractical or impossible to do with traditional node based graphs.

Keywords—navigation meshs; pathfinding; path smoothing; artificial intelligence; funnel algorithm; raycasting; triangulation; rectangle expansion; search space; A*; marching squares

I. INTRODUCTION

In games path finding refers to the calculation and following of the simplest and shorted route between 2 different points in a world. Examples of this in use can be easily seen in most games, e.g. cars moving around a city, enemies moving around obstacles towards the player, and so on.

Generating realistic paths and efficient paths is one of the most integral parts of AI design in modern videos games. How the developer goes about it can make or break the game, changing the NPC’s from appearing near-human to being so dumb they appear to continually walk into walls.

Writing path finding code can be a frustrating activity for developers, complex path finding algorithms can be confusing, difficult to understand and easily broken. Small mistakes can lead to incorrect paths being generated, or generation taking too long to be usable in real-time.

One of the earliest and simplest solutions to generating paths in video games was littering the world with waypoints that represent a node-graph and calculating paths using Dijkstra’s algorithm. While this algorithm worked fine for small and simple areas, it does not scale up well for larger graphs as the algorithm results in all nodes being evaluated before a final path is calculated.

As a solution to the problem of scalability, A* was introduced. It was essentially a rewritten version of Dijkstra algorithm that made use of a heuristic that limited the number of nodes that needed evaluation to generate the shorted-path. At best it was far more efficient that Dijkstra and at worse it had the same performance, there was no real downside to using it

A* has now been widely adopted by the games industry. It’s currently the go-to solution for path finding [2], as it’s efficient, tried and tested and because it’s easy to implement. Most research now being done is simply in optimizing A*, rather than looking at alternatives.

One of the most promising areas for optimizing A*, for use in games, is finding different, and more efficient, methods to organize the search-space that the A* paths are built on. At the moment the search-space for the world is usually represented path-waypoints littered throughout the world, which A* then uses as its graph.

While this representation works, it doesn’t work well for AI, as it provides no representation of what areas of the map are actually walkable, all the AI knows is that it must follow the path exactly or it may encounter collision. This produces AI that tends to be predictable, that walks in straight and unnatural lines and is usually very bad at avoiding dynamic obstacles. Avoiding this behavior either means drastically increasing the density of waypoint nodes in the map, which heavily impacts A*’s performance, or using a different method to represent the search-space.

An alternative method for representing the search space is through the use of navigation-mesh’s (navmesh’s). Navmesh’s provide a representation of the environment that describes exactly what parts of the map are walkable and which are not. They are usually represented in the form of several polygons encompassing the walkable area.

Navmesh’s also have the benefit that they tend to need considerably fewer polygons to cover the same area that would normally be covered by many waypoint nodes. This increases the speed at which paths can be generated.
Fig. 1 Screenshot of map represented using waypoints. [1]

Fig. 2 Screenshot of same map as Fig 1 represented using a navmesh. [1]

Note the simplicity of the graph compared to the one shown in Fig 1

Representing the world in this way allows AI characters to smooth paths and dynamically avoid obstacles by moving onto areas it knows are walkable, rather than just rigidly following waypoints as would have to be done in a waypoint-graph to assure avoidance of collision. This gives characters a far more natural and responsive behavior.

Fig. 3 Screenshot of a path generated on a waypoint-graph. [1]

Fig. 4 Screenshot of the same path generated on a navmesh. Notice how more natural the path appears. [1]

So if navmesh’s provide a better alternative to waypoint-graphs, why are they not used more often in video games? The simple answer is that it’s far simpler to generate and use a graph of waypoints than it is to generate and use a navmesh.

Navmesh’s, when they are used, are typically manually placed by artists when creating maps. Few solutions exist to automatically generate them, even fewer exist for purely 2D environments. The ones that do exist tend to be hard to implement, overly complex or they produce suboptimal navmesh’s.

This is in contrast to waypoint-graphs which can be dynamically generated with little more than a handful of collision checks across the map.

This paper discusses the different methods that can be used to generate paths in arbitrary 2D environments and documents the creation of a simple implementation of some of the methods discussed.

II. RELATED WORKS

Because of the importance of the topic, many different people have been involved in research regarding path finding within the scope of video games. Few people however have focused specifically on the area of navmesh’s, and even fewer have looked specifically at navmesh’s in two dimensional environments. The following 3 papers are the most detailed ones that were available at the time of writing.

The authors of [6] have covered a wide scope of information, discussing the many different ways that search-spaces can be described for the purposes of path finding. They briefly touch on the ways in which the search-space can be optimized for use in the A* algorithm, during which they discuss the benefits of navmesh’s and how they are becoming more used in games because of the benefits they bring for intuitive AI. They show how navmesh’s can be used to make considerably smoother paths for AI to follow. However they don’t go into detail on how navmesh’s are actually generated in practice.

The authors of [7] go into more detail on the actual implementation and use of navmesh’s. They talk about the different ways in which paths can be plotted between polygons in a navmesh, specifically mentioning the funnel algorithm, and briefly describing how the concept of building a navmesh comes down to simple triangulation and polygonization. However they don’t specifically go into mentioning how this techniques can be applied to arbitrary environments. Their figures all show manually created examples.

The authors of [8] provide even more in-depth coverage of the subject area. They specifically look into the many ways that navigation meshes benefit games, as well as mentioning the potential problems they could cause. They describe how navigation meshes can often be automatically generated using collision geometry present in the level, and go on to describe an algorithm similar to our own, that will automatically produce a navmesh across the walkable area of the map. Their algorithm consists of finding the contours of the collision
geometry and using Delaunay tessellation to produce the navigation mesh polygons.

III. GENERATING NAVMESHS

To show the process of generating navmesh’s we have developed a simple implementation that takes the form of a small simulator. The simulator takes two images, a bi-color collision image and a map image (the map image is purely for show). It then uses the collision images to generate a navmesh using a choice of generation algorithms, and begins populating the map will NPC’s that path find to a randomly placed target (which the user can move by clicking anywhere on the navmesh). Paths to the target can also be smoothed with a number of different algorithms, which are also described below.

**Fig. 5** Example map image used to generate navmesh.

**Fig. 6** Example collision image used to generate navmesh. Anything white is considered walkable, anything else is solid.

There are numerous different ways to generate navmesh’s in 2D environments. A lot of them are specific to the implementation they are being used in. We will primarily be looking at two of the most common, triangulation and rectangle expansion, as these both simple to understand and implement, as well as being among the most efficient methods.

IV. TRIANGULISATION

Triangulation involves working out the contour of the walkable area of the world, turning it into a complex polygon, then splitting it into multiple polygons (triangulation).

In our implementation the first step to successfully producing a navmesh with this algorithm is converting the collision-map into a simple grid-based representation. This simply involves sampling each pixel in the collision image and adding it to a 2D array as a simple no/collision flag. For addition performance with this step we also down-sample the collision image to be several times smaller than the original. This results in considerably higher performance with some the marginal downside that the navmesh does not exactly follow the contour of the walkable area, though this is typically not an issue in 2D environments which are typically tile-based.

Once we have a collision grid, we then find the first point on the contour of the walkable area, this is any point that has collision but neighbor’s an area of no-collision in the grid. From this position we run the marching squares algorithm, which gives us the contour of the entire walkable area. We continue doing this until we have visited every possible contour point in the grid. Every iteration of marching squares is an individual unconnected “walkable area”.

**Fig. 7** Shows the result of running marching squares on all contour points. Each purple dot is an individual point in the contour of the walkable area. Notice how this algorithm also finds all the “holes” in the middle of the grid.

Once we have run this algorithm we will have our contour defined as a list of points. This list of points is what we will used as the vertices for the triangles we will eventually make our navmesh out of.

To increase performance in the following steps we can simplify this contour a lot. To do this, we simply go through each point of the contour and if it’s on the same vertical or horizontal plane as the point before and after it, then we can remove it as it’s performs no function.

To prevent all the triangles having to stretch long distances to different vertices, which can produce undesirable results, we don’t remove ALL unnecessary points in the contour, we define a maximum distance between points, and will not remove points if the distance between the next and previous point are larger than this distance.
Fig. 8 Shows the result of simplifying the contour points, with a maximum point distance of 128.

We can now use these contour points as the vertices of a simple polygon. This polygon can then be split up into the triangles that make up our navmesh using a triangulation algorithm.

We make use of the ear-clipping algorithm, which is fast, simple to implement and produces reasonable results. The primarily downside to it, is that it doesn’t take into account “holes” in the polygon. We leave this exercise it up to the reader to further study algorithms that take complex polygons into consideration, as they can be considerably more complex.

The ear-clipping algorithm works by taking into account the fact that any given simple polygon with more than 4 points will always contain at least two no-overlapping ears [3]. A triangle is considered an ear if two sides are on the polygons edge and the third side is inside the polygon.

To deconstruct our contour polygon into individual triangles the algorithm recursively finds each ear and “clips it”, each individual ear clipped is considered a triangle within our navmesh.

Fig. 9 Shows a 4 vertex polygon. Four ears can be made out of this polygon; v1-v2-v3, v1-v2-v4, v2-v3-v4, v1-v3-v4.

The algorithm is relatively trivial to implement, it simply takes the form of a loop testing all 3-vertex-pairs for being ears and clipping them until only 3 vertices are left. Determining if the 3 vertices are an ear or not simply involves checking if the ear vertex is concave, and if the triangle contains no concave vertices [4].

The triangles generated by deconstructing our polygon are the triangles that make up our navmesh. To perform path finding across this navmesh all we need to do is associated each triangle with its neighbor and generate an A* path using the center of each polygon as a graph node. To calculate the neighbors of nodes we simply determine if they share two vertices, if they do, then they share a side and are considered neighbors.

Fig. 10 Navmesh generated using ear-clipping triangulation. Red lines show the neighbor association between individual triangles.

The paths generating will at this point still look no more natural than a normal waypoint graph, though they will be faster to calculate. The key step to making the paths more natural is smoothing the paths, we discuss this later on. As it is though, the AI already has a considerably expanded search space. It now has a far better representation of the walkable areas of the world, opening up the possibility of allowing more reactive movement.

Triangulation is a simple algorithm to implement, but difficult to get perfect. You may notice in our implementation we have a couple of major issues. Namely it’s difficult to support complex polygons without considerably more complicated algorithms. The algorithm can also produce sub-optimal navmesh’s, than those created by a user manually, notice the many very thin triangles that occur in wide areas.

V. RECTANGLE EXPANSION

Rectangular expansion is a producing navmesh’s that keeps the benefits of generating the navmesh out of triangles, but bypasses the problems associated with triangulation. It supports complex polygons and produces near-optimal output.

The algorithm works fairly simply, it’s also has a slight
performance edge of triangulation as we don’t have to calculate the contour of the map to run it.

The algorithm works by fitting as many squares as possible into the map to form a navmesh. To do this we run a loop, where each iteration we find a point in that’s not in collision, and not already in our navmesh. We then create a square that encompasses the point we selected, and expand it both horizontally and vertically until it encroaches upon collision or other rectangles in the navmesh. We finish iterating when we can no longer find points that are not already in our navmesh or in collision.

When we have finished iterating we take each square and split it diagonally into two triangles. Each triangle is then associated with all the triangles made of neighboring squares. This then gives us a graph we can use to generate paths using A*.

VI. SMOOTHING PATHS

Now that we have a usable navmesh, we can generate paths using A* or any other path finding algorithm of choice. However at the moment paths are generated based on the mid-point of each triangle in the navmesh. This produces the same unnatural zig-zag style paths that a waypoint system would produce.

To make the paths look more natural we need to implement methods of path smoothing. In our implementation we used two of the most common, fastest and simplest methods to implement, ray-casting and the funnel algorithm.

VII. RAYCAST ALGORITHM

The ray-cast algorithm goes through each 3 sequential nodes in the path list and casts a ray between the first and last nodes. If the ray stays inside the navmesh then the middle node is superfluous and can be removed to produce a path straighter and more natural looking than previously.

To determine if a ray stays inside the navmesh by checking multiple evenly-spaced points along the ray, and seeing if the points lie inside any of the navmesh’s polygons. If any are outside, the ray is considered to go outside the navmesh.

The algorithm is slower than others, but simple to implement and fast enough to work in real-time. However its performance degrades the larger the map becomes, and the more polygons that need to be checked.

VIII. FUNNEL ALGORITHM

The funnel algorithm is a more advanced method of smoothing paths that has considerably reduced time over the raycast algorithm.

It works by taking the assumption that each triangle in the navmesh shares a full side with its neighbor’s – a fact that makes it inaccurate when using rectangular-expansion, but perfect when using triangulation – and that to follow the path the path follower must pass (‘funnel’) through the shared sides (‘portals’) of all triangles it passes through [5].
At the start of the algorithm, what is known as the “apex” (the point of the line we want to shorten) is set to the starting point, and the left and right side vertices are stored. The algorithm then enters a loop where the left and right vertices are constantly moved forward. Each time we move the vertices forward we determine if a ray can be cast between the “apex” and the new point, if not then we have found a corner. We then set the last point as a new node in our path list and set that vertex as the new “apex”. The easiest way of visualizing this is to think of stretching a rubber band through the portals from the start point and the end point.

This algorithm works considerably faster than ray-casting between each individual node and produces a similar result. The speed comes at a price however, this method does not work correctly with the rectangular expansion method and can cause

IX. CONCLUSIONS

During this paper several different methods have been discussed that can be used to generate and path find on navmesh’s in 2 dimensional space. They are all viable and effective methods to be used in games, and are considerably more effective and natural looking than previous waypoint based graphs that are currently the norm for 2D games. Our simulator can easily support many hundreds of paths being continually generated without any noticeable slowdown, something that is unlikely to be seen when using traditional waypoint graph based path finding.

However the largest and most interesting aspect of using navmesh’s is dependent on how the user exploits the new expanded search-space that this algorithm provides. It’s particularly useful with the development of AI.

I’ve provided a simple example of what can be done using this in the simulator. With this we can have many hundreds of NPC’s all path finding to the same point, whilst using the knowledge of what is walkable that the navmesh provides to dynamically avoid each other.

I believe overall the simulator effectively implements the different algorithms and provides a general base for those interested in learning about how to implement the use of navmesh’s in 2D environments. The implementation and discussion should hopefully be simple enough that most
developers will be able to understand and easily implement its suggestions.

X. FUTURE WORK

There are primarily two different things that I believe deserve future work on with this.

Triangulation should be capable of being run of complex polygons that could include holes. Ear-clipping, though a good method for simple levels, will simply not be useful for larger or more complicated levels, which are almost certain to contain holes within the navmesh. A possible solution to this would be to use a more advanced algorithm, perhaps the Delaunay Triangulation algorithm rather than ear-clipping.

Some form of spline based interpolation should also be implemented. At the moment paths are still very linear, though efficient it does not appear natural. Interpolating between points in the path, and making sure the interpolated curves do not go off the navmesh, would greatly add the naturalness of behavior when NPC’s follow these paths.

REFERENCES


