Final Project Report
CS 633

Triangulating and Pathfinding in Unity3D

Abstract:

Triangulation is a well understood technique for discretizing a space into manageable chunks. Likewise, A* is a well-known heuristic search technique for shortest path solutions in a discrete space. By building a visibility graph between triangle centroids, a start point, and an end point, in a 2D projected triangulation of a 3D space we can use A* to navigate on it in order to quickly find a shortest path in a convenient discretization.

Motivation and Introduction:

The primary motivation for this project is a selfish one: to build on my personal library of useful tools and scripts for various game engines. More importantly: to better understand these methods and the applications of them to things I have worked with before. Consequently, I took a two-part approach to generating a nice triangulation and followed it up with a method of pathfinding similar to things I have done in the past, yet distinct. We triangulate, convert to Delaunay, then build a visibility graph from the triangulation.

In a 3D game there is a lot going on. Fortunately for us, we can ignore most if it. In fact, when it comes to determining autonomous agent pathing, we can ignore the entire third dimension!

To determine an agent's path, we need minimally a current location and a goal location, or destination. Since most agents in a video game will stay firmly on the ground, we can determine our path along the ground. This also means we do not need to chunk the entire 3D space the agent will be traversing. Instead, we can find only those vertices which would impact the agent, project those vertices into 2D space on the ground, and continue from there.

Once we have simplified the space we are working in we can worry about how we are going to find our path. A traditional approach is to apply A* shortest path search to some discretized space. Naïve solutions involve creating a grid, which could be done in our 3D world. However, grids can be large and to make a sufficiently usable grid we would have to find a subdivision degree no less than the space our agent can occupy that gives us space around obstacles to traverse. Individually subdividing grid squares around obstacles is a noteworthy compromise, but complicates the A* method (we would have to decide what costs we wished to arbitrate to the subdivided squares). Instead, we can look at triangulating the space and using the generated triangles as our A* "grid."

Of course, not all triangulations are created equally. We could generate many triangulations that would be bulky and not well suited to navigation. Triangulations with a lot of very long and thin triangles will cover a strange range of space and make for a nonsensical A* result if we try to consider only their centroids. We would have to find some other way to convert the triangles into usable pathfinding information. Consequently, we seek to find a Delaunay triangulation, to give us a nice triangulation. We will do this by first finding any triangulation, and then converting it into a Delaunay legal form.

Methods Used and Studied:

Computational Geometry – Algorithms and Applications: Delaunay Triangulation

From this we used the concept of Legalizing edges to convert existing triangles into Delaunay legal triangles by removing illegal edges and replacing them with their counterparts. By the textbook's definition, an illegal edge is any edge we could flip to locally increase the smallest angle (of a pair of adjacent triangles sharing that edge). In doing so we can turn any triangulation into a Delaunay triangulation.

To begin, we have to decide which vertices are actually relevant to our agent. Not all vertices can actually restrict our movement. We start by gathering only those vertices from which we can project a line straight downwards (in world space) and contact a floor object

within the height (possibly padding it if our agent has any variance in height through their actions) of our agent. Any vertices that are too high above their nearest floor, or which contact a non-floor object first can be ignored. Those which are too high obviously will not impact our agent significantly, and those which contact some non-floor object will be included within the convex hull of the 2D projection of the object and therefore can be excluded from our triangulation.

We then project our remaining vertices downwards into their respective floor objects, and will use these vertices henceforth. We can further prune our triangulation vertices by removing duplicates, or near duplicates. We take the list of vertices we have and one by one add them to a new list of "final" vertices, only if there are no vertices already in this list that are within some distance epsilon of the vertex in question. This lets us remove any vertices that were effectively overlapping in 2D because they sat above and below each other in 3D. We also include some additional vertices from the edge/corners of our floor. We now have our set of vertices to triangulate.

Unlike previous investigations into triangulation, we have an additional constraint to consider here. We cannot connect any vertices that cannot see each other. So, before creating any edges or triangles, we will have to check that no obstacles lie between the vertices being connected.

We begin by finding some initial triangle (any trio of points that can see each other). We then add to our set of triangles incrementally by finding points we can connect to existing triangles. For each point left in our list, search edges we have made for line of sight to their vertices and conflicts with other edges. If we find none, add the triangle between these points and add the two new edges to the list of edges. Remove the point from the list. Rinse and repeat until we are out of vertices to add to the triangulation.

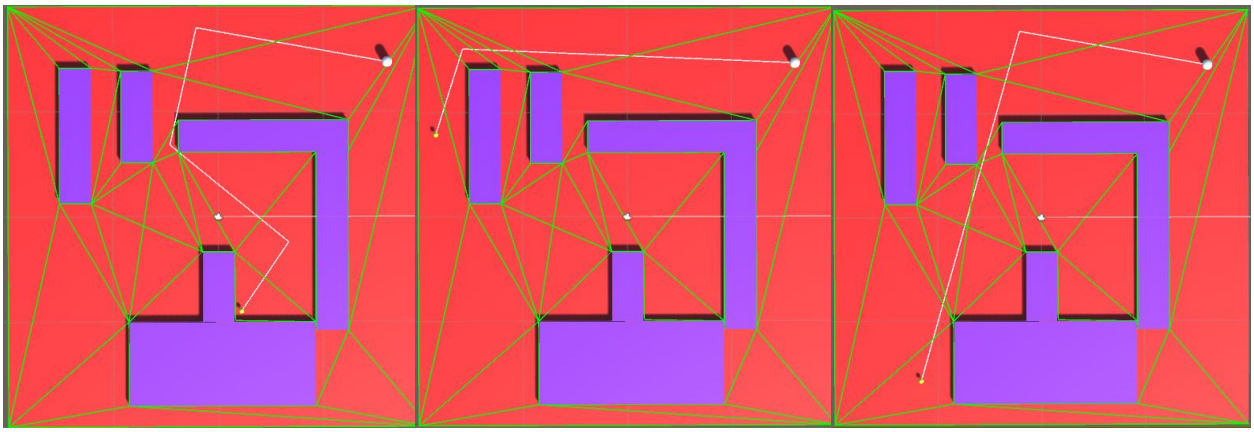
Once we have our initial triangulation, we can go through the list of edges and check for illegal results, correcting them as per the Delaunay Triangulation method from *Computational Geometry*.

Once we have our Delaunay triangulation, we can easily build a visibility graph to pass to any agents acting on this space. To do so, we simply iterate over our final list of triangles and determine their centroids by finding the intersection of lines from any two vertices in the triangle to their opposite edge. We collect these centroids, and use their positions to create visibility graph nodes. We then have these nodes seek out all other nodes they can see and add them to a list of neighbors each keeps internally. These internal lists will be used later for pathfinding, which we are nearly ready for at this point. We now have the core visibility graph for our space.

As far as Unity is concerned, we do all of this preprocessing (vertex pruning and triangulation) and initial path-building (our initial visibility graph) all on one "master" game object. This game object (called the MasterControlProgram, or MCP, in my scenes) also keeps track of a list of agents in the scene for our convenience in testing this method. The MCP takes the generated visibility graph (which, is really just a list of nodes which all know who their visible neighbors are) and hands it off to each agent. At the time of this writing we have only tested it with single-agent scenarios. The agent then adds its current and goal positions (if it has any goals active) to the list of nodes and connects them by visibility to the existing ones. It also normalizes all points from the ground to a height about at its center. This last step is important for the native position checking in Unity3D; points on the floor will detect as farther away from the agent, requiring us to use a wider distance epsilon when checking to see if the agent has arrived at its destination (intermediary or final).

Once equipped with this visibility graph, the agent can as needed remove/add its "start" and "goal" nodes and use the graph to run A*. As long as there are no changes in the environment, this will always guarantee the agent can find a path. For this implementation of A* we use a rather simple heuristic of distance- real distance up to this node, and straight-line distance to the goal. Recall that we are already connected to our neighbors by line of sight, so we are never at risk of picking a node that we cannot reach directly.

Results/Findings:

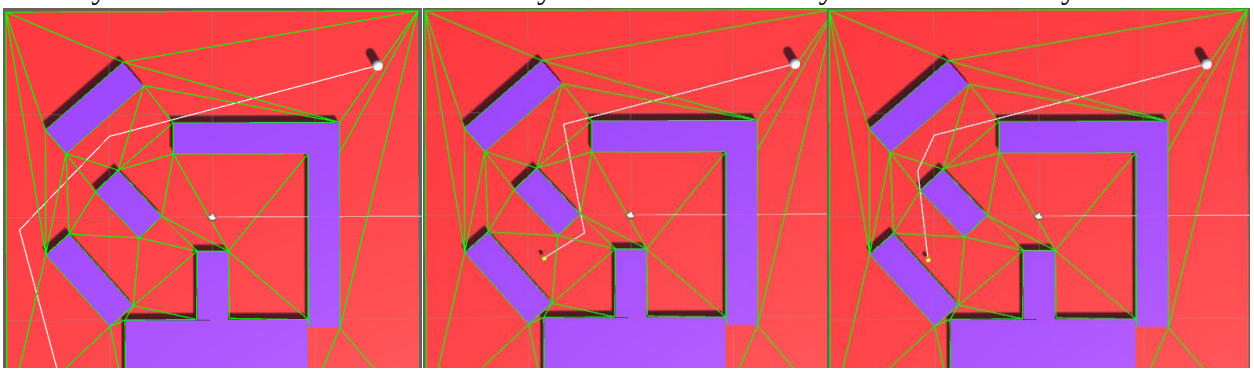


Demonstration of triangulation on the basic set of obstacles. The yellow orb is the goal position, and the white capsule is the agent. Blue blocks are obstacles, green lines are the triangles, white line is the path, and red is the floor. Note how the path ignores the centroids of triangles it does not need, beelining straight for the goal when it can see it.

I encountered some difficulty initially with condensing the 3D environment into a 2D working space. A lot of this came down to the intricacies of Unity3D collision/LoS detection. At first, I kept getting bad triangulations- triangulations which overlapped with obstacles- because of this. After adding a quick O(n) preprocessing step to push all vertices slightly away from their obstacles to ensure none of them count as being inside them (which causes them not to "see" the obstacle when checking LoS with points on the other side of it) and making an adjustment to ignore non-static (any non-environment) object when determining LoS this problem quickly disappeared. This was an expected challenge.

As per A*, we are of course guaranteed an optimal path, although with our heuristics we are by no means guaranteed an optimal search. Fortunately, the algorithm is robust against cornering itself.

The complexity of this implementation is a little disappointing. Unfortunately, it's also the only real metric I have to measure it by at the moment. Unity's time metric only measures in



Demonstration of a second configuration of obstacles. Note the slight adjustment in the path between images center and right.

seconds, and this does not seem capable of getting millisecond or better precision outside of a frame-by-frame delta-time, which sadly is not helpful in this case since we do most of our calculations in other threads to avoid pausing the simulation (which means we do not have access to a meaningful delta-time). Consequently, all attempts to measure the time taken by the running code comes back as 0s.

Which brings us back to the complexity. Breaking this down into pieces, we have O(n) time to gather all relevant vertices, some function bounded by O(n^2) to remove duplicates, O(n) to build the initial triangulation, O(k) to convert to Delaunay (where k is the number of edges), and O(p^2) to build the visibility graph (where p is the number of resultant triangles). The complexity of A* is exponential based on the branching factor and the depth of the solution(O(b^d)). Fortunately, due to our heuristics and the convenience of visibility as a neighboring factor (allowing us to skip several triangles to reach the best ones faster) the complexity of our A* should not be too bad relative to the space it's in.

An additional fortune is that unless the environment changes, we only need to do most of these calculations once. Alternatively, it can be said that we only need to do some of these calculations once per significant environmental event. In the case of this project, that is indeed once overall for preprocessing vertices and triangulation (though see Discussion and Future Work for more on this). We only have to connect two nodes per agent for O(a*p^2) each time we want to change our destination and re-path (where a is the number of agents changing their path at the same time; for an individual agent or as a general rule, a = 1 is sufficient, since these are all independent operations), and we only have to run A* as often. These last two are substantially faster than the preprocessing operations in terms of complexity (as long as A* is playing nice and our lines of sight are long; tight and winding corridors should increase the complexity, but it's still sufficiently fast as a one-shot operation).

Conclusion:

Ultimately, I did achieve my original goal of implementing triangulation of obstacles into a convenient free-space area for A* pathfinding. Despite a few persisting hiccups, I am actually rather pleased with how it turned out.

Vertex collection and preprocessing is among the slowest of operations in terms of complexity, along with building the visibility graph. Triangulation and Delaunay conversion is hectic but fairly fast. A* does not disappoint, and it was good to see it play nice with the visibility graph. Since classic implementations tend to connect nodes to their nearest neighbors, not their farthest, I was unsure if it would cause any problems.

Discussion and Future Work:

There are still a few kinks I would like to work out in time, mostly regarding the finesse of preprocessing vertices and ensuring that they do not end up inside objects (two objects close together can effectively eat each other's vertices and cause those verts to connect with triangles through their neighboring obstacle. My current padding solution does not prevent this). One plausible solution might be to gather obstacles nearby in some radius and account for all of them in the shifting, but that can lead to some wonkiness as well depending on the positions of each relative to the vertex. A better test, but one that would undoubtedly be much slower would be to gather the obstacles as above, but instead of doing one shift based on all of them iteratively shifting with some accounting for each, giving preference to the original obstacle the vertex belongs to and checking the vertex's position in relation to the bounding boxes of each obstacle to ensure it is not inside one (since the LoS checks I do to detect obstacles cannot tell me if a vertex is inside an object).

Additionally, I do not think it will be too difficult to expand this setup to account for dynamic obstacles, and I would very much like to at least expand it to account for changes in the environment (e.g. the clearance or addition of obstacles).

Let us start by looking at the latter, more general case: additional and removal of obstacles. When such an event occurs three things will need to happen to make the changes we would need: first we would have to either remove the vertices that are now gone or add the new ones that have appeared. Each of these introduce some complications of their own if we wish to avoid completely regathering the vertices. In the case of removal, we would have to at least identify any duplicates nearby that would have been used in this area still so we can include them; in the case of addition, we would have to parse out the new duplicates. The latter is far

easier than the former. A lazy approach would be, as I mentioned, to regather all the vertices. The impact of this action could be lessened by keeping track of sections of floors and only doing this operation for affected sections, and it is still only O(n)- it is just inconvenient to discard memory like that.

As for dynamic obstacles, this change is actually relatively simple and would not require too much unnecessary computation. We leave the structure of our pathfinding as is and simply add a check from our position to our most immediate destination (which can be some triangle centroid or our goal node) for line of sight. This is the costliest change, actually, since the act of determining LoS is a fairly costly operation. Fortunately, Unity's ability to do this is sufficiently fast as to be many time per frame without much detriment (so long as care has been taken throughout development not to overdo anything). We would also have to decide when and how often we wanted to do this check; often enough to detect the change as something blocks our path, but not too often as to have many unnecessary LoS checks.

If we detect an obstacle between us and our immediate destination- perhaps only if we detect such an obstruction over the course of some amount of time- we re-path. There is another small complication to be addressed here, which is ensuring that the newly (and in this case temporarily) blocked path does not get considered, since the visibility graph still shows that it is viable. Simply removing the node is not a valid solution, since there could be others beyond the obstacle that detect as clear but that are really blocked, and moreover we do not want to get rid of it so that we can have it later, when the obstacle moves. One thing we can do that would be simple would be to add a LoS check between nodes when doing A*. This has the potential to muck up A* and prevent a solution from being found if nodes are getting blocked and unblocked. It also adds a lot of checks that in the current system without dynamic obstacles would be totally redundant (which makes it a very specialized addition). Overall not a great fix, but maybe a reasonably sufficient start to one if it ran fast.

For the sake of this project I only worked with a single, flat floor. In theory as I have designed it the addition of stacked floors or slopes should be no problem. There will almost certainly have to be a few changes in order to incorporate them seamlessly, but it shouldn't be a

total overhaul. The big thing for each would be doing the operation on each section of floor independently. Vertices contacting a second story floor of a building do not need to be considered for the first floor. While I am working on that it might also be nice to chunk large singular floor objects into smaller sections to let the script be more applicable to a wider range of environments. Sectionalizing floors and operations like this would enable me to run these all in parallel and cover more ground without taking an exponential hit to speed.

The other problem specifically regarding slopes would be that my current implementation reduces everything to X-Z calculations and ignores the Y-axis, assuming that everything has been projected to 2D. A slope could still be treated as 2D, but would need to use all 3 axes to operate. This will be the more significant and damaging overhaul to the existing project.

Bibliography

Berg, Mark de. *Computational Geometry: Algorithms and Applications*. World Publishing Corporation, 2013.

Introduction to A*, 26 May 2014, www.redblobgames.com/pathfinding/a-star/introduction.html.

Unity3D API, https://docs.unity3d.com/ScriptReference/