PATHFINDING IN 3D SPACE - A*, THETA*, LAZY THETA* IN OCTREE STRUCTURE

Third-Year Student Project

March 8, 2016

Ruoqi He & Chia-Man Hung

1 INTRODUCTION

Pathfinding in 3D space with obstacles is an essential problem in real 3D strategy games or in drone navigation. In this project we present a viable solution to 3D pathfinding based on the commonly used graph-based algorithm in games, A* algorithm, and its two variants, Theta* algorithm and Lazy Theta* algorithm. It is also based on an octree space division.

2 RELATED WORK

Among the first 3D real-time strategy games, *Homeworld*, released in 1999 developed by Relic Entertainment and published by Sierra Entertainment, is the most successful one since it featured a 3D environment in space, therefore allowing movement in every direction. However, in this game, obstacles in space are convexe and disjoined. Their pathfinding algorithm is simple - when coming across an obstable, the moving object only have to get around it by choosing a random direction and it will surely reach its destination. It is not satisfying in more complex scenes. In 2002, another big real-time strategy game, *Warcraft III*, is released. Although it is claimed to be in 3D, it is intrinsicly a 2D game since moving objects can only fly at a fixed height.

In 2005, Surazhsky et als.[8] provided an algorithm to compute exact geodesics on 3D surfaces represented by triangle meshes. In 2013, Harabor et als.[4] proposed an algorithm named Anya to compute exact shortest paths based on A* algorithm on a 2D grid. However, there is not any known algorithm yet to compute exact shortest path in 3D space. In 2009, Crane et als.[2] introduced an algorithm to compute approximate geodesics by simulating a heat propagation and simply solving linear equations. Although it is simple enough on 3D surfaces, it is hard to be extended in 3D space. It works correctly only on a homogeneous tetrahedral triangulation of the space, which requires considerable amount of nodes. Another difficulty consists of combining the two border conditions, which has a more significant affect in 3D space.

The heat method mentioned above, as well as classical pathfinding algorithms such as Dijkstra, solve the problem globally, ie. find an approximate shortest path to any node from a given source. Another algorithm named Floyd-Warshall can even find the shortest paths between every pair of nodes. However, they cost a great amount of time and require considerable memory. Thus, they are not adaptable to complex scenes.

In 2012, Cui et als.[3] reviewed a pathfinding method on 2D or 3D surfaces. The main idea is to first find a tunnel in a navigation mesh by A* algorithm and then find the shortest path in the tunnel. This method is not optimal since the found tunnel is not always the best choice.

It is also hard to be extended to 3D space due to the complexity of finding the shortest path in a 3D tunnel.

3
MAIN ALGORITHMS

We choose to explore the application of the popular graph-based algorithm in 3D space. Since no exact method is currently available, we aim at finding a good approximate solution. The algorithms that we use are variants of A* algorithm.

3.1 World representation

A*-based algorithms are often used on a grid representation of the terrain or a navigation mesh with its dual graph. In our case, a plain 3D grid map is obviously too costly in most circumstances – it will require $O((\text{worldSize} / \text{gridSize})^3)$ memory space to store everything. A navigation mesh in 3D is a collection of connected convex polygons which represents the passable space. However it is difficult to find an algorithm that decomposes a polygonal space into natural convex blocks. There exists an algorithm of convex polygon decomposition in Cgal library but it divides the space into a collection of vertical columns, which is not suitable for pathfinding. If we consider a tetrahedral triangulation of the space, we also face some problems. A full homogeneous tetrahedral triangulation with added points will create a large number of small tetrahedrons – Its memory consumption is comparable to space grid map, and the tunnel found with the dual graph will be likely too narrow which creates zigzag in open areas. Without added points in space, a tetrahedral triangulation will mostly become long stripes connecting two obstacle, causing the dual graph to be very inaccurate.

We need a simple and hierarchical structure to represent the space. We choose therefore to use the octree representation of the world, in which all the obstacles are registered in the lowest level (leaves) of the octree. With this structure the memory cost is approximately $O(a log a)$ where a is the surface area of the scene, which is linear to the input mesh data if the meshes use similar sized triangles. We consider two methods to transform the octree into a graph: one is the dual graph and the other is the edge-corner graph constructed with voxel corners and edges. The graph connects the passable areas and is used by the following pathfinding algorithms.

3.2 A*

Peter Hart et als. of Stanford Research Institute (now SRI International) first described the algorithm in 1968 [5]. It is an extension of Dijkstra's 1959 algorithm.

A* solves the problem by searching the solution that incurs the smallest cost (least distance travelled in our case) among all possible paths. Starting from a specific node of a graph, it constructs a tree of paths starting from that node, expanding paths one step at a time, until one of its paths ends at the predetermined goal node.

Each note has corresponding f, g and h values and $f(n) = g(n) + h(n)$ where n is the last node on the path, $g(n)$ is the cost of the path from the start node to n, and $h(n)$ is a heuristic that estimates the cost of the cheapest path from n to the goal. There are two sets of nodes, named *open* and *closed* respectively. *open* stores nodes that are explored but not all its neighboring nodes are explored while *closed* stores nodes where it and all its neighboring nodes are explored. At the beginning, we add the source node into *open*. At each iteration of its main loop, we pick a node with the smallest f-value from *open* and explore its neighboring nodes. Details are illustrated in Algorithm 1. We stop the iteration when reaching the destination.

3.3 Theta*

In 2007, Nash et als.[6] proposed Theta* algorithm, a variant of A* algorithm.

The key difference between Theta* and A* is that Theta* allows the parent of a vertex to be any vertex, unlike A* where the parent must be a visible neighbor. Theta* is identical to A* except that Theta* updates the g-value and parent of an unexpanded visible neighbor s' of vertex s by considering the following two paths (procedure ComputeCost):

Path 1: As done by A*, Theta* considers the path from the start vertex to $s[= g(s)]$ and from s to s' in a straight line $[= c(s, s')]$, resulting in a length of $g(s) + c(s, s')$ (Line 33).

Path 2: To allow for any-angle paths, Theta* also considers the path from the start vertex to $parent(s)[= g(parent(s))]$ and from $parent(s)$ to s' in a straight line $[= c(parent(s), s')]$, resulting in a length of $g(parent(s)) + c(parent(s), s')$ if s' has line-of-sight to $parent(s)$ (Line 28). The idea behind considering Path 2 is that Path 2 is no longer than Path 1 due to the triangle inequality if s' has line-of-sight to $parent(s)$.

3.4 Lazy Theta*

Later in 2010, Nash et als.[7] proposed Lazy Theta* algorithm, a variant of Theta* algorithm.

Theta* considers Path 2 if s' and $parent(s)$ have line-of-sight. Otherwise, it considers Path 1. Lazy Theta* optimistically assumes that s and $parent(s)$ have line-of-sight without performing a line-of-sight check (Line 28). Thus, it delays the line-of-sight check and considers only Path 2. This assumption may of course be incorrect. Therefore, Lazy Theta* performs the line-of-sight check in procedure SetVertex immediately before expanding vertex s'. If s' and $parent(s')$ indeed have line-of-sight (Line 33), then the assumption was correct and Lazy Theta* does not change the g-value and parent of s'. If s' and $parent(s')$ do not have line-of-sight, then Lazy Theta* updates the g-value and parent of s' according to Path 1 by considering

Algorithm 1 A*

```
1:  procedure Main()
2:      open := closed := φ;
3:      g(s_start) := 0;
4:      parent(s_start) := s_start;
5:      open.Insert(s_start, g(s_start) + h(s_start));
6:      while open ≠ φ do
7:          s := open.Pop();
8:          if s = s_goal then return "path found";
9:          closed := closed ∪ {s};
10:         for each s' ∈ nghbr(s) do
11:             if s' ∉ closed then
12:                 if s' ∉ open then
13:                     g(s') := ∞;
14:                     parent(s') := NULL;
15:                 UpdateVertex(s, s');
        return "no path found";
16:
17: procedure UpdateVertex(s,s')
18:     g_old := g(s');
19:     ComputeCost(s, s');
20:     if g(s') < g_old then
21:         if s' ∈ open then
22:             open.Remove(s');
23:         open.Insert(s', g(s') + h(s'));
24:
25: procedure ComputeCost(s,s')
26:     /* Path 1 */
27:     if g(s) + c(s, s') < g(s') then
28:         parent(s') := s;
29:         g(s') := g(s) + c(s, s');
```

Algorithm 2 Theta*

```
1:  procedure Main()
2:      open := closed := φ;
3:      g(s_start) := 0;
4:      parent(s_start) := s_start;
5:      open.Insert(s_start, g(s_start) + h(s_start));
6:      while open ≠ φ do
7:          s := open.Pop();
8:          if s = s_goal then return "path found";
9:          closed := closed ∪ {s};
10:         for each s' ∈ nghbr(s) do
11:             if s' ∉ closed then
12:                 if s' ∉ open then
13:                     g(s') := ∞;
14:                     parent(s') := NULL;
15:                 UpdateVertex(s, s');
        return "no path found";
16:
17: procedure UpdateVertex(s,s')
18:     g_old := g(s');
19:     ComputeCost(s, s');
20:     if g(s') < g_old then
21:         if s' ∈ open then
22:             open.Remove(s');
23:         open.Insert(s', g(s') + h(s'));
24:
25: procedure ComputeCost(s,s')
26:     if lineOfSight(parent(s), s') then
27:         /* Path 2 */
28:         if g(parent(s)) + c(parent(s), s') < g(s') then
29:             parent(s') := parent(s);
30:             g(s') := g(parent(s)) + c(parent(s), s');
31:     else
32:         /* Path 1 */
33:         if g(s) + c(s, s') < g(s') then
34:             parent(s') := s;
35:             g(s') := g(s) + c(s, s');
```

Algorithm 3 Lazy Theta*

```
1:  procedure Main()
2:      open := closed := φ;
3:      g(s_start) := 0;
4:      parent(s_start) := s_start;
5:      open.Insert(s_start, g(s_start) + h(s_start));
6:      while open ≠ φ do
7:          s := open.Pop();
8:          SetVertex(s);
9:          if s = s_goal then return "path found";
10:         closed := closed ∪ {s};
11:         for each s' ∈ nghbr(s) do
12:             if s' ∉ closed then
13:                 if s' ∉ open then
14:                     g(s') := ∞;
15:                     parent(s') := NULL;
16:                 UpdateVertex(s, s');
        return "no path found";
17:
18: procedure UpdateVertex(s,s')
19:     g_old := g(s');
20:     ComputeCost(s, s');
21:     if g(s') < g_old then
22:         if s' ∈ open then
23:             open.Remove(s');
24:         open.Insert(s', g(s') + h(s'));
25:
26: procedure ComputeCost(s,s')
27:     /* Path 2 */
28:     if g(parent(s)) + c(parent(s), s') < g(s') then
29:         parent(s') := parent(s);
30:         g(s') := g(parent(s)) + c(parent(s), s');
31:
32: procedure SetVertex(s)
33:     if !lineOfSight(parent(s), s) then
34:         /* Path 1 */
35:         parent(s) := argmin_{s'∈nghbr(s)∩closed}(g(s') + c(s', s));
36:         g(s) := min_{s'∈nghbr(s)∩closed}(g(s') + c(s', s));
```

the path from s_{start} to each expanded visible neighbor s'' of s' and from s'' to s' in a straight line and choosing the shortest such path (Lines 35 & 36). We know that s' has at least one expanded visible neighbor because s' was added to the open list when Lazy Theta* expanded such a neighbor.

4 IMPLEMENTATION

4.1 Environment

We chose Unity to implement the 3D path finding algorithms in order to better visualize the result. All codes are written in C#, and the scene is built with Unity Editor. We use the C5 Generic Collection Library for the priority queue implementation, which is used by all main algorithms.

4.2 Graph construction

We first construct the octree representation of the scene. We subdivide the octree wherever the space it represents intersects an obstacle. The subdivision runs recursively until the lowest level (which is a fixed number), and the leaves of the octree containing the obstacle are marked not passable. Since the obstacles are normally objects represented by triangle meshes, we implemented a fast method to detect triangle-cube intersection, proposed by Akenine et als.[1]. This method uses separating axis theorem, which states that:

> *Two convex polyhedra, A and B, are disjoint if they can be separated along either an axis parallel to a normal of a face of either A or B, or along an axis formed from the cross product of an edge from A with and edge from B.*

After the octree is built, we transform it into a graph. The dual graph (Figure 1 left) is easy to build – we simply transform every leaf of the octree into a graph node and add the arcs between all the neighboring nodes. The edge-corner graph (Figure 1 right) is slightly more difficult since there are shared graph nodes for each octree nodes. We used a hash table to store the graph nodes created using their grid coordinates as the key, and another hash table to store the graph arcs using the coordinates of the two nodes it connects as the key. In this way repeated node and arc creation can be avoided.

We also experimented an additional constraint on the octree: we require that the difference of levels between neighboring octree leaves be no more than 1. We name such octree *progressive octree*. In the result section we will further explain the advantages of this constraint.

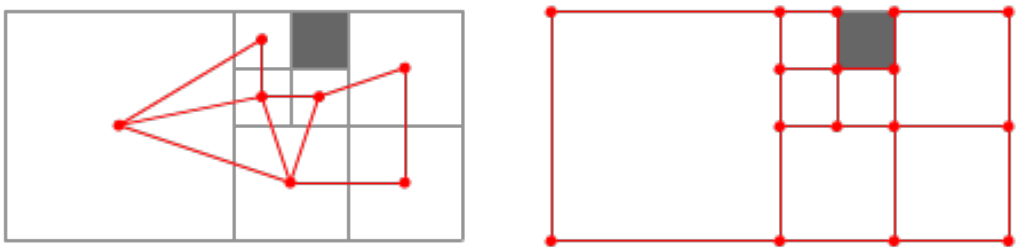

Figure 1: Left: dual graph/ Right: edge-corner graph

4.3 Implementation of algorithms

We follow the pseudo-code and successfully implemented the A*, Theta* and Lazy Theta* algorithms. We used a super class of the graph node to store additional information on the node such as value of g, f, and parent pointer. An instance of this class is only created when a node is explored in the path finding process, and it is stored in a hash table taking the corresponding graph node as the key. We use a priority queue for the open set to easily retrieve the best evaluated node, and the closed set is only a Boolean tag on the nodes. The heuristic function we use is simply the direct Euclidian distance. At the end, we apply some additional line of sight checks to improve the final result.

4.4 Line of sight

Since the line of sight check is the most time-consuming part of the Theta* algorithm, it is important to have a fast and robust method to perform line of sight check. Inspired by the 2D grid version of the line of sight check which runs entirely with integer values, we extend it to a 3D grid space version which is significantly more complex. The advantage of a line of sight check running on integers is that it can correctly answer the visibility of a path touching the edges or corners of a blocked voxel (which can be quite common in a grid).

4.5 Avoidance of exhaustive search

If the destination is unreachable, it may cause the path finding algorithm to run through all the available nodes of the graph before stopping and returning a negative answer. To avoid that, we can precompute the connectedness of the graph nodes, and mark each node with the index of the connected set it belongs to. In this way, we can immediately stop the algorithm if the source and the destination do not belong to the same connected set. In case of dynamic modification of the graph, we can use the union-find data structure to easily update the connected sets of the graph and to check if two nodes are connected ($O(logn)$ time in worst case).

4.6 Position of source and destination

Since the three algorithms implemented are all based on graphs, their resulting paths are sequences of segment lines connecting existing nodes. But the source and the destination point should be any given position in the space, which is not necessarily an existing node. To cope with this, we temporarily insert a source node and a destination node into the graph when initiating the path finding. These nodes are connected with the 8 corners of the corresponding octree leaves they are in, and their connected set information is also updated. Once the path is found, the temporary nodes are removed from the graph.

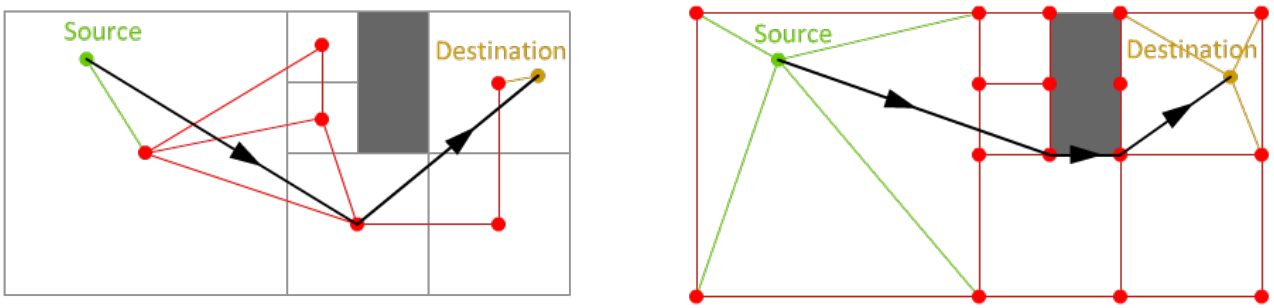

Figure 2: Left: dual graph/ Right: edge-corner graph

4.7 Multisource

The algorithms can be optimized for multiple sources single target situation, which is commonly used in group path finding. To do so, we first invert it into a single source multiple targets problem which is easier to consider. After the algorithm find the path between the source and the first target, all intermediate data (closed set, open set, g values for explored nodes) are kept for the following path searches. If the next target is in the closed set, then the path has already been found – simply backtracking from this node will yield the result. And if not, since the h value has changed, we recalculate the f values for each node in the open set and update the priority queue. We than continue the calculation with the existing open set until reaching the next target. The inverse paths of the paths we find will be the final result. This optimization is particularly efficient if the sources are close to each other, which is very common in strategy games.

4.8 Application in games

In a video game, we can apply the algorithm to multiple moving units and assign to them the paths to follow. However a unit often has a colliding radius and should not move too close to the obstacle. We can therefore introduce a fixed clearance graph by extending every vertex of the obstacle mesh along its normal by a certain distance before creating the octree. This distance should be set to the radius of the unit. In this way, we can guarantee that the unit will not move into the given range of the obstacles.

Another important thing in games is to keep the units from colliding into each other while still following the right path. When a group of units is moving together we can introduce an invisible force to alter their planned trajectory in order to keep a safe distance. However in certain cases this can lead to incorrect paths, as illustrated in Figure 3. We solve this by doing a short path find for each unit from the current position to the next node in its following path, and add the result to the main path. This can trigger if the unit is blocked, too off path, or just doing a regular check per second since this intermediate path check is often very short and quick.

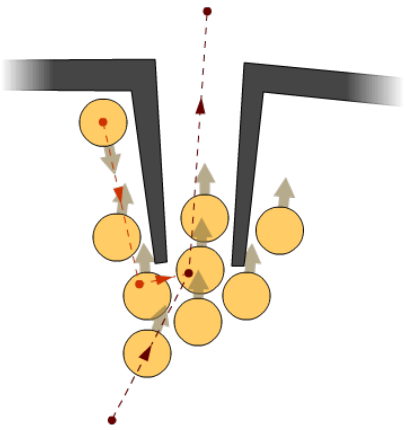

Figure 3: Quick check to correct incorrect position

5 RESULTS

While the results of the A* algorithm are ladder-like and unrealistic as expected, the results of the all-angle algorithms are quite good and natural. We tested the three algorithms with different graph types and octree level on different scenes. The first two scenes are simple situations (a single wall and a sphere) where exact distance can be calculated, and the last scene is a more complicated one. The paths are chosen randomly, and the average calculation times and path lengths for 100 executions are listed as below: (The following time cost is mesured in the Unity Editor and we can expect better performance on release build.)

1. A single wall. The exact shortest path can be calculated by considering the minimum of the two possible paths. The position of the wall is such that in the first test (left) source is generally inside sparser areas (larger octree leaves) and in the second test (right) destination is inside sparser areas. (See Figure 4)

Data Structure	Algorithm	distance	time cost	distance	time cost
Octree	A*	125.18%	1.6ms	124.56%	4.5ms
	Theta*	101.58%	11.8ms	109.24%	53.0ms
	Lazy Theta*	101.73%	6.6ms	109.48%	22.1ms
Progressive Octree	A*	125.19%	4.6ms	126.27%	5.8ms
	Theta*	101.58%	21.5ms	101.63%	42.7ms
	Lazy Theta*	101.39%	10.6ms	102.27%	18.1ms

Table 1: Comparison - A single wall - Left: source sparse/ Right: destination sparse

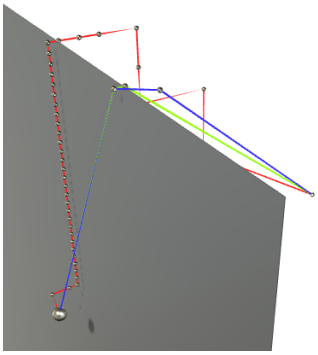

Figure 4: Wall - Red: A*/ Green: Theta*/ Blue: Lazy Theta*

2. A single sphere. The exact shortest path is made of two straight lines and a circle arc. The first test is done using a smaller octree (smaller maximum level), and the second using an octree with 2 more levels. (See Figure 5)

Data Structure	Algorithm	distance	time cost	distance	time cost
Octree	A*	121.38%	1.5ms	121.03%	28.0ms
	Theta*	104.59%	6.7ms	102.71%	236.1ms
	Lazy Theta*	104.65%	4.2ms	102.34%	114.8ms
Progressive Octree	A*	127.43%	3.3ms	126.88%	55.6ms
	Theta*	103.49%	6.3ms	101.23%	229.4ms
	Lazy Theta*	103.68%	3.2ms	101.16%	108.8ms

Table 2: Comparison - A single sphere - Left: level = 7/ Right: level = 9

3. A complex scene filled with large obstacles. The exact distance is not available so the distances displayed here are absolute distances. The first test is based on edge-corner graphs and the second one is based on dual graphs.

These tests show some very interesting results. None of the algorithms finds exact shortest

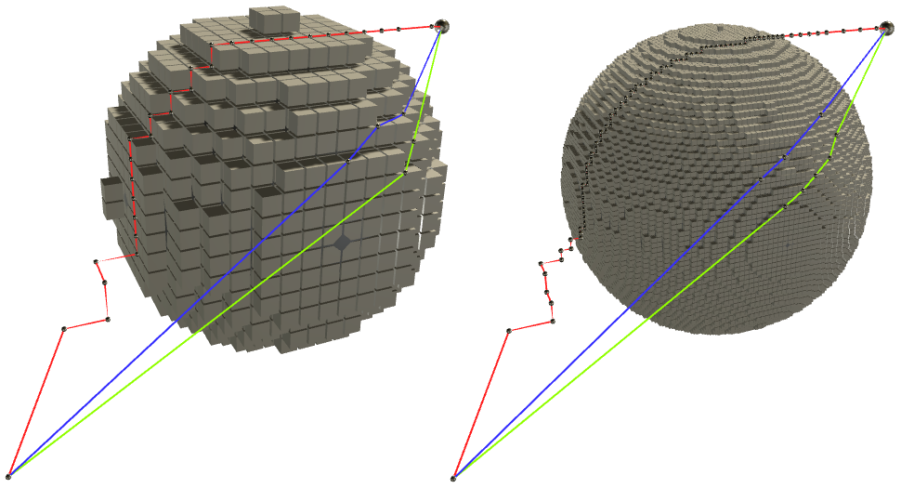

Figure 5: Sphere - Right is two level higher than Left

Data Structure	Algorithm	distance	time cost	distance	time cost
Octree	A*	3.2472	9.5ms	3.3302	5.3ms
	Theta*	2.4108	23.9ms	2.4600	45.5ms
	Lazy Theta*	2.4135	9.5ms	2.4592	16.3ms
Progressive Octree	A*	3.3949	14.1ms	3.3222	7.2ms
	Theta*	2.4009	17.6ms	2.4158	43.1ms
	Lazy Theta*	2.4057	7.8ms	2.4205	14.6ms

Table 3: Comparison - A complex scene - Left: edge-corner graph/ Right: dual graph

paths, but the two any-angle algorithms are able to find short paths with acceptable error (<5%). We find out that the Lazy Theta* algorithm has in average a 100% speed up compared to Theta*, while having an average path length almost the same as that of Theta*. This makes Lazy Theta* a promising choice for 3D path finding. We also find out that if the destination is in sparse areas the paths found by our algorithms will likely be longer. However, the use of progressive octree can well resolve this problem, as shown in the second test of the simple wall experiment. The progressive octree is also proven to be a good choice for graph construction, as it reduces the average time cost of the any-angle algorithms even if more nodes are inserted into the graph. The dual graph which have fewer nodes also makes it more difficult for any-angle algorithms to find the path. So the best choice so far is using the combination of progressive

Data Structure	Algorithm	distance	time cost
Edge-corner graph	Theta*	4.3549	53.5ms
	Lazy Theta*	4.3718	28.9ms
Crossed graph	Theta*	4.3388	185.3ms
	Lazy Theta*	4.3411	31.0ms

Table 4: Comparison - Maze - Progressive octree

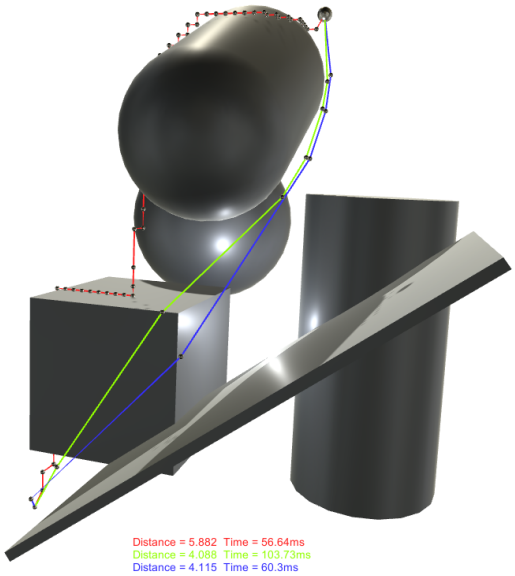

Figure 6: Edge-corner graph - Red: A*/ Green: Theta*/ Blue: Lazy Theta*

octree and edge-corner graph, along with the Lazy Theta* algorithm.

By using an octree with higher level the error of the path can effectively be reduced, but at a considerable cost, because one increased level means 8 times the calculation.

6 CONCLUSION

Pathfinding is an important topic and has applications in several domains, such as video games and robotics. The three algorithms A*, Theta*, Lazy Theta* are efficient ways to compute approximate shortest paths without exploring the whole graph in most cases. In this project, we proposed a progressive octree and edge-corner graph data structure. We succeeded in implementing them and have made some comparaisons. Our implementation is optimized by computing connexe components to avoid exhaustive search. The case of multisource is taken into account by keeping the closed set and updating the f values of the open set. Several details of application in games such as group movement are also discussed.

REFERENCES

[1] Tomas Akenine-Möller. Fast 3d triangle-box overlap testing. In *ACM SIGGRAPH 2005 Courses*, page 8. ACM, 2005.

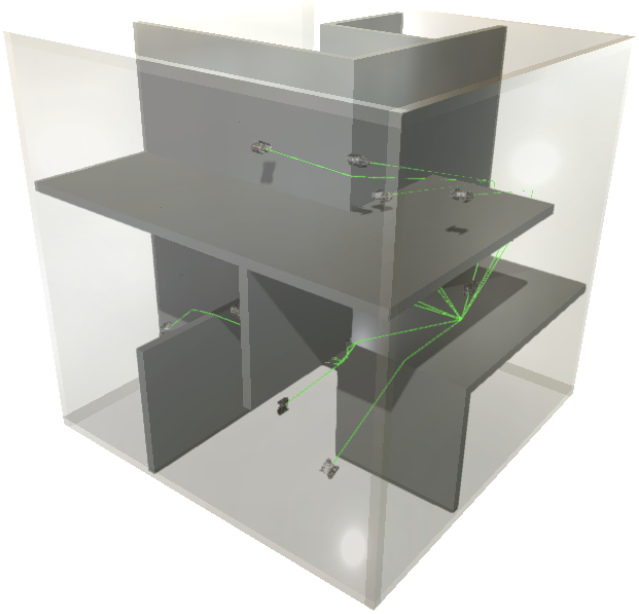

Figure 7: 3d maze with space unit navigation

[2] Keenan Crane, Clarisse Weischedel, and Max Wardetzky. Geodesics in heat: A new approach to computing distance based on heat flow. *ACM Transactions on Graphics (TOG)*, 32(5):152, 2013.

[3] Xiao Cui and Hao Shi. An overview of pathfinding in navigation mesh. *IJCSNS*, 12(12):48, 2012.

[4] Daniel Damir Harabor, Alban Grastien, et al. An optimal any-angle pathfinding algorithm. In *ICAPS*, 2013.

[5] Nilsson Hart and Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems, Science, and Cybernetics*, SSC-4(2):100–107, 1968.

[6] Alex Nash, Kenny Daniel, Sven Koenig, and Ariel Felner. Theta^*: Any-angle path planning on grids. *Proceedings of the National Conference on Artificial Intelligence*, 22(2):1177, 2007.

[7] Alex Nash, Sven Koenig, and Craig Tovey. Lazy theta*: Any-angle path planning and path length analysis in 3d. In *Proceedings of the National Conference on Artificial Intelligence*, 2010.

[8] Vitaly Surazhsky, Tatiana Surazhsky, Danil Kirsanov, Steven J Gortler, and Hugues Hoppe. Fast exact and approximate geodesics on meshes. In *ACM transactions on graphics (TOG)*, volume 24, pages 553–560. ACM, 2005.