3D Pathfinding in voxelized world with A*, Theta* and Lazy Theta*

Hang Yu, Lowell Novitch

Fa20 CS3/580 Research
DigiPen Institute of Technology

1 Introduction

In the game *Homeworld* (1999), it introduces battle in a 3D dimension free space with sparse and disjoint environmental objects only; spaceships in the game would only need steering behavior to avoid obstacles. For traversing in more complex scenes, 3D pathfinding is required.

In this project, we present a fast and reliable solution using commonly used graph-based pathfinding algorithm, A* algorithm, and two of its variants, Theta* and Lazy Theta* algorithm in a voxelized world in UE4.

2 Related Work

In Ruoqi He & Chia-Man Hung[1], the paper demonstrates a 3D path finding solution in Unity3D. They implemented a progressive Octree data structure and corner-graph as optimization technique.

In Venugopalan Sreedharan[2], the author published a production-ready UE4 plugin for 3D pathfinding that was used in the author's own game. The implementation takes account the volume of the path finding agent and is able to progressively recalculate paths on dynamic obstacles.

In Canis Wong[3], the author released a solution that was built in Unity3D that allows boids to navigate in 3D space with dynamic obstacles.

Daniel & Nathan[4] has released a benchmark data set that is generated from *Warframe* (2013), but there is no actual benchmark contest about that data set yet[1].

[1] From search engine Google and Bing, there is none so far at this moment Dec,2020.

3 Main Algorithm

From work done by Daniel & Nathan[4], we choose commonly-used graph-based algorithm, A*, to be the prime candidate for our solution for 3D pathfinding. We also choose two variants, Theta* and Lazy Theta*, which has their advantages and disadvantages in navigating in 3D space.

3.1 World Representation

The underlying framework that this project is based on is called DoNNavigation(**DoN** for the rest of the paper) plugin provided by Venugopalan Sreedharan[2]. It features a procedure of creating a 3D voxel grid by specifying the voxel size and the dimension of the grid. The procedure can be done in drag and drop fashion in the UE4 editor.

Thus, we are able to apply pathfinding algorithms in a 3D grid space directly with a maximum of 26 degrees of freedom(**DOF**) at each grid cell. However, 26 DOFs cannot be used directly as access to (x, y, z) does not guarantee access to (x + 1, y, z + 1), so we need to verify the accessibility of (x, y, z + 1), (x + 1, y, z), and (x + 1, y, z + 1) before adding (x + 1, y, z + 1) as a neighbor and so on.

An obvious problem with a plain 3D grid is the memory space it requires is $O(N^3)$, Ruoqi He et al.[2] proposed a solution that uses the subdivision of voxel Octree to represent spaces occupied by obstacles. But it introduces a new procedure in that they need to convert the Octree into its dual graph in order for the pathfinding algorithm to work. We believe their method is less approachable than a 3D grid but a production solution should use their approach.

*3.2 A**

Peter Hart et als[5] first described the A* algorithm in 1968. It extends the Dijkstra's Algorithm by introducing a heuristic function to the final cost such that the A* algorithm combines the strength of both.

The A* algorithm solves the shortest path problem by always expanding in the direction of smallest cost, it builds a tree of paths from the starting nodes, exploring paths one step at a time until one path reaches the destination node.

Each node is associated with three values, g, h and f where $f(n, \mathrm{N}) = g(n) + h(n, \mathrm{N})$ and n stands for the current node, N stands for the destination node. $g(n)$ is the cost of the path from starting node to the current node. And $h(n, \mathrm{N})$ represents a number that estimates the cost from the current node to destination. If $h(n, \mathrm{N})$ is chosen from a function that always produces a value that is smaller or equal to the actual cost to the destination, then A* is said to be admissible which guarantees an optimal path on return.

As the Dijkstra's Algorithm, A* puts nodes into 2 sets, either *open* or *closed* respectively. *closed* contains nodes that are removed from *open* and those nodes need not to be added to *open* again. *open* stores all nodes that are explored but not all of their neighbors

are yet to be explored, and *open* sorts those nodes based on their f values with the least value being the next node to be explored. Once the node that is removed from *open* is the destination, the algorithm is terminated.

*3.4 Theta**

Nash et al[6] first published the Theta* algorithm in 2007. It slightly modified the A* algorithm such that it is allows any node to become the parent node of any other node. It enables any-angle pathfinding which plain A* cannot.

The key difference is that in the procedure of *ComputeCost* (see the pseudo code), Theta* does a line of sight test from the parent of the current node to an unexplored neighbor. If the line of sight test fails, then we simply do the same procedure as A* in Path 1. But when the test succeeds, Theta* execute Path 2 where we use the parent of the current node in updating the unexplored neighbor. Thus, the unexplored neighbor is parented with the parent of the current node and its *g* value is updated from the parent as well. Due to triangle inequality, Path 2 is always smaller or equal to Path 1.

One important thing to notice is that Theta* could be confused as a rubber banding technique, it is not. Because the searching performed by Theta* generates the *open* and *closed* set almost completely different from A*, thus their pathfinding results looks different even when A* results are rubber banded with line of sight tests.

One obvious drawback for Theta* is that its speed performance must be impacted by the line of sight algorithm. In an actual 3D game environment, there are additional factors that need to be involved which we will discuss in the implementation section later. To counter this drawback, we will introduce a variant of Theta*.

*3.4 Lazy Theta**

This algorithm was also made by Nash et al[7] in 2010. The Lazy Theta* optimistically assume the line of sight is a success and execute the Path 2 in the procedure *ComputeCost*. And only when removing a node from the *open* set, it does a late line of sight test to verify that assumption.

If a node does not have line of sight with its parent, then we reparent and update the node with a neighbor that is in the *closed* set with the lowest cost. The idea behind is that there must be at least one of its neighbors pushed the current node into the *open* set, thus that neighbor must be in the *closed* set already.

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 "success";
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');
16:    return "fail";
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:    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 "success";
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');
16:    return "fail";
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:    if lineOfSight(parent(s), s') then
28:        /* Path2 */
29:        if g(parent(s)) + c(parent(s), s') < g(s') then
30:            parent(s') := parent(s);
31:            g(s') := g(parent(s)) + c(parent(s), s');
32:    else
33:        /* Path1 */
34:        if g(s) + c(s, s') < g(s') then
35:            parent(s') := s;
36:            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 "success";
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');
17:    return "fail";
18:
19: procedure UPDATEVERTEX(S, S')
20:    g_old := g(s');
21:    ComputeCost(s, s');
22:    if g(s') < g_old then
23:        if s' ∈ open then
24:            open.Remove(s');
25:        open.Insert(s', g(s') + h(s');
26:
27: procedure COMPUTECOST(S, S')
28:    /* Path2 */
29:    if g(parent(s)) + c(parent(s), s') < g(s') then
30:        parent(s') := parent(s);
31:        g(s') := g(parent(s)) + c(parent(s), s');
32:
33: procedure SETVERTEX(S)
34:    if !lineOfSight(parent(s), s) then
35:        /* Path1 */
36:        parent(s) := argmin_{s'∈nghbr(s)∩closed}(g(s') + c(s', s));
37:        g(s) := min_{s'∈nghbr(s)∩closed}(g(s') + c(s', s));
```

4 Implementation

4.1 Environment

This project is built on UE4.25.4 on win10 x64. Most of our code is in c++14 that does the actual pathfinding. We build all algorithms on top of the free DoN plugin available in UE4 Marketplace. We expose a behavior tree task to the editor with several black board keys for debug drawing. The entire testing environment is written in Blueprint.

4.2 Voxelization

The DoN plugin provides a manager class that can be dragged and dropped into the scene, it features options to change the voxel size and grid sizes. We choose the following settings for this project:

Table 1 Voxel grid set up.

Voxel Size	100 cm
XGrid Size	50 (# of grids)
YGrid Size	80 (# of grids)
ZGrid Size	30 (# of grids)
Total grids	12000 (# of grids)
Total volume	12000 * 10^6 cm^3

4.3 Implementation of Algorithm

We follow the pseudo code in section 3 to implement all three algorithms. There are a few important details that are worth pointing out.

Our pathfinding task is done in a backthread in a query fashion, the behavior tree task simply checks if the task is finished every tick. Thus, we have constructed data structures that hold the node information inside each query. We use the Unreal Engine's built in TMap as the node data structure with the pointer to a voxel grid as the key, and its associated data as value. Each pathfinding query has its own data structure so that different query does not interfere. This kind of decentralized design suits the multithreaded query that we set up.

There are two optimizations that we did.

A major optimization is what we call a segment optimization. It uses an equalized the segment distance $c(s, s')$ (see the pseudo code section) in updating the g value, the equalized distance is equal to voxel size - 100. In principle, we should have the $c(s, s')$ depends on the distance from the current node to its neighbor, but by using an equalized distance in practice, all three algorithms explore less nodes which make them faster. The tradeoff is that that path might not be optimal.

A minor optimization is that since the Unreal Engine uses centimeters as the unit for all of its distance calculation, we can replace float/double with unsigned integer as the value

type in the priority queue that represents the *open* set.

4.4 Line of Sight

The line of sight test consisted of two steps.

We first execute a simple line trace first with UWorld::LineTraceSingleByObjectType. If it succeeds, then we proceed to a more complicated test by passing the agent pawn's collision component to UWorld::SweepMultiByObjectType such that it does a sweep test from start to end against static and dynamic object types.

If both tests succeed, then we are safe to say there is a line of sight.

4.5 Rubber Banding

A rubber banding method is applied to the raw result of all three algorithms. It uses the same line of sight algorithm described in the above section. It iterates from the destination to the starting position and eliminates redundant visible nodes.

5 Results

In this section, we demonstrate statistics we collect from results of all three algorithms in various circumstances.

The raw results from A* are colored in yellow line, and its rubber banned result is colored in black line. Purple line is Theta* and orange line is Lazy Theta* for both raw results and rubber banded results.

5.1 Impact of Segment Optimization

As mentioned in section 4.3, we implemented an optimization such that the *g* value is updated with a fix value for all neighbors. It not only results in visual difference of path chosen, but also impacts open and closed set, and speed.

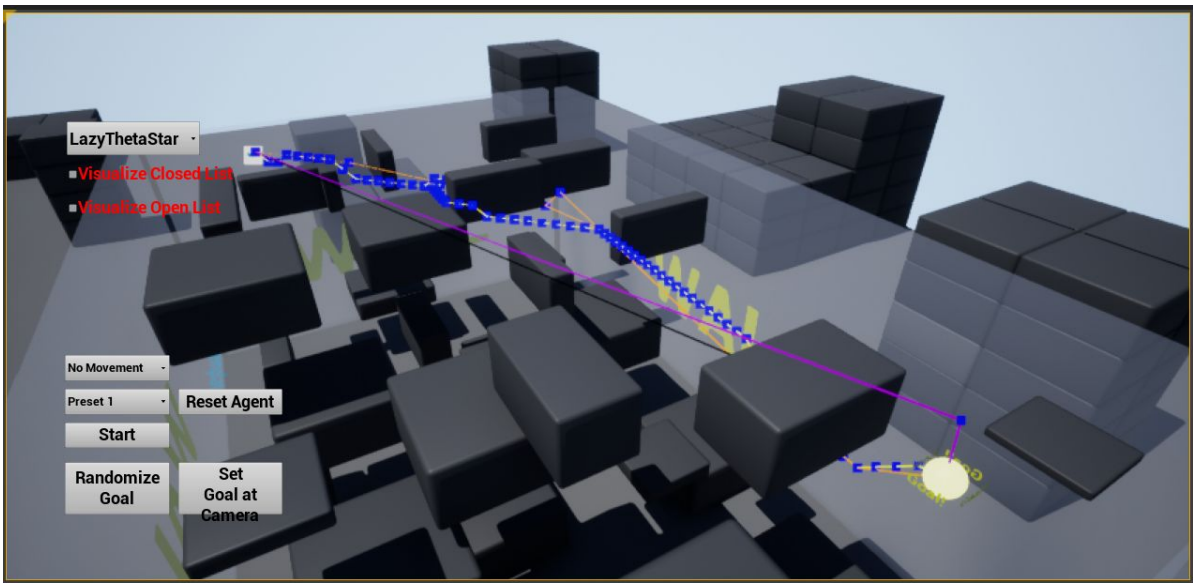

Figure 1 Preset 1 path location and agent location with all three algorithms with segment optimization.

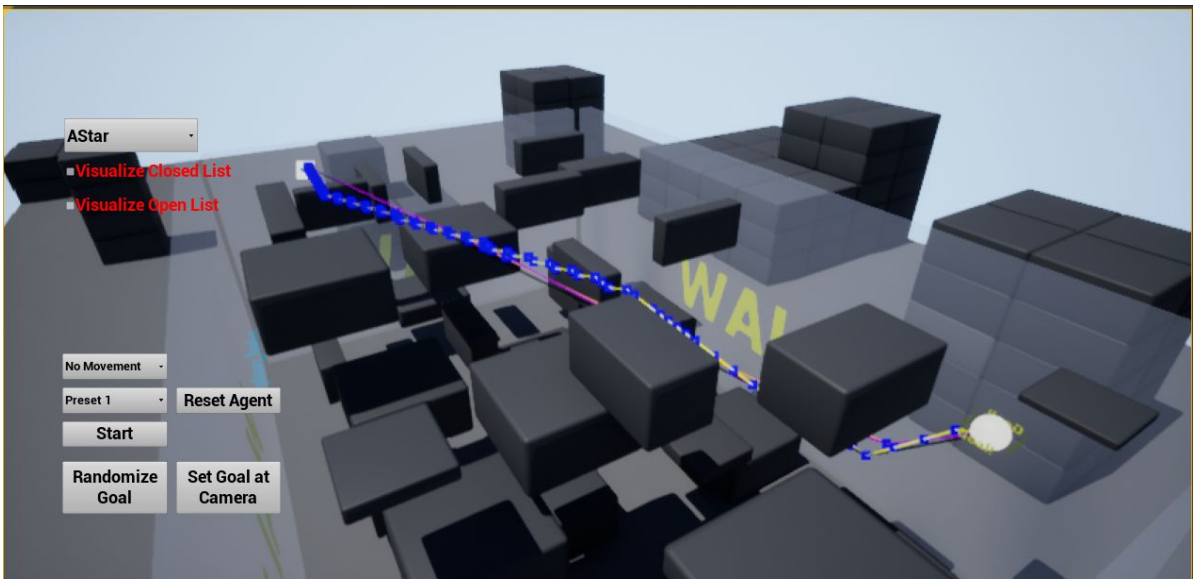

Figure 2 Preset 1 path location and agent location with all three algorithms without segment optimization.

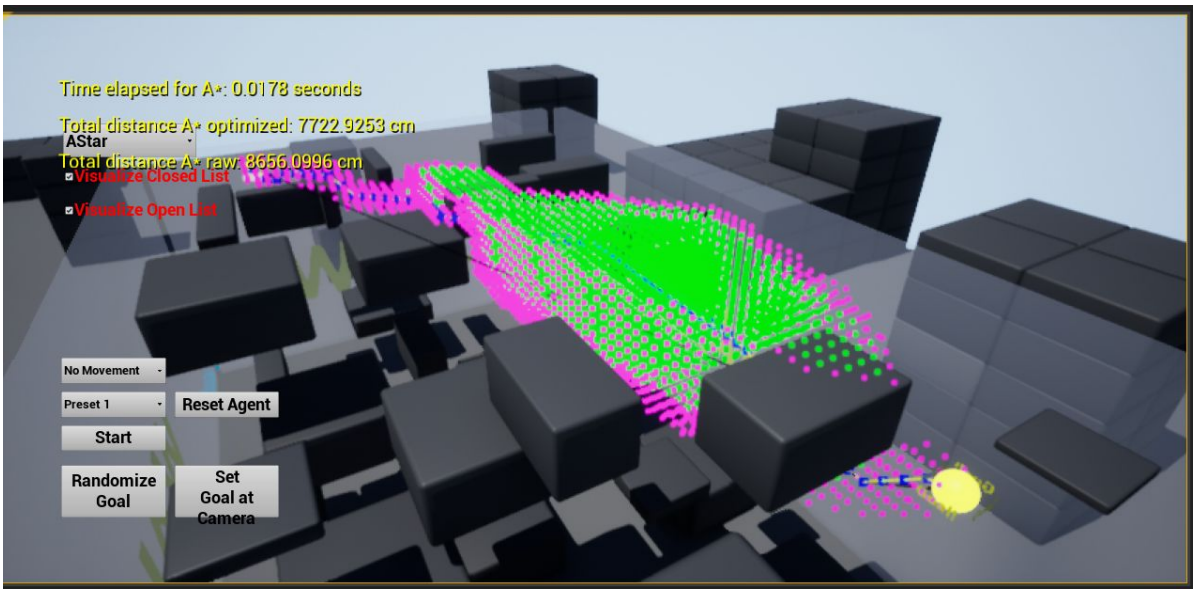

Figure 3 Preset 1 path location and agent location with A* algorithms with segment optimization.

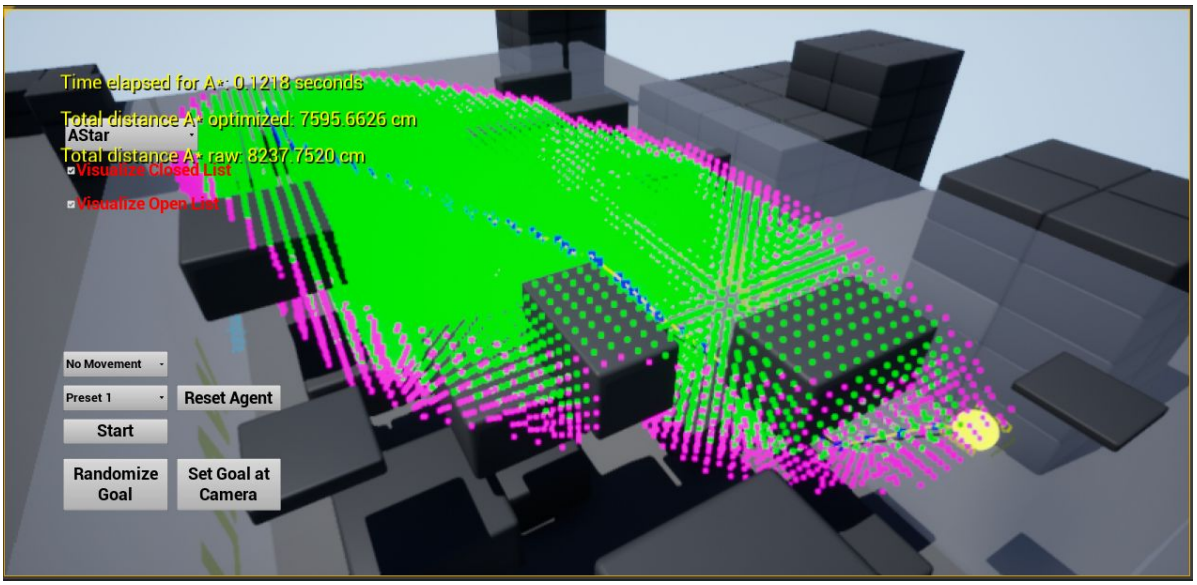

Figure 4 Preset 1 path location and agent location with A* algorithms without segment optimization.

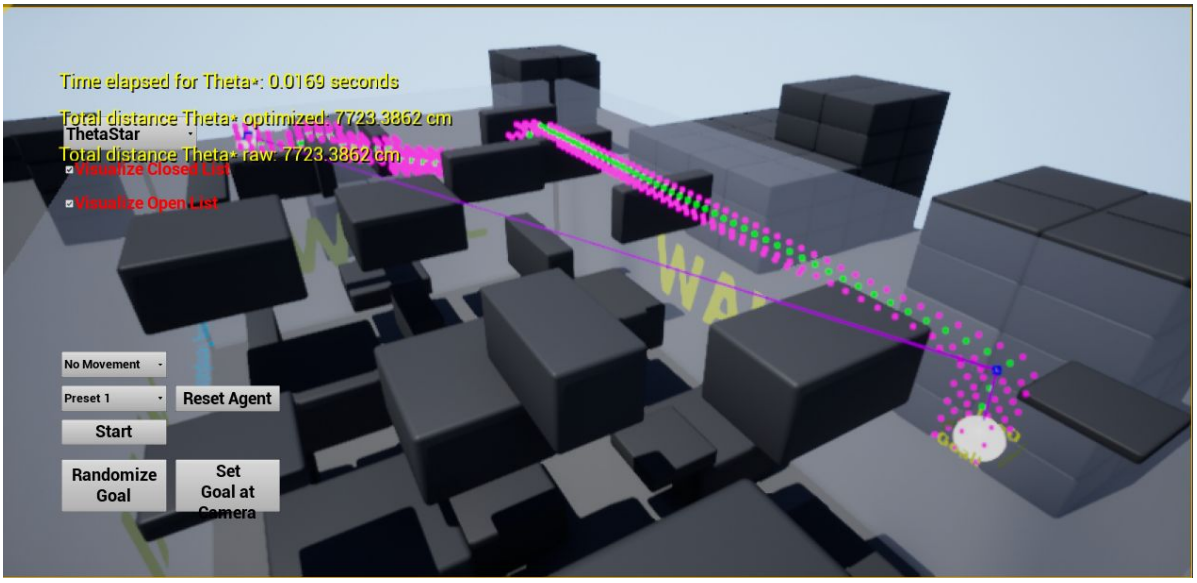

Figure 5 Preset 1 path location and agent location with Theta* algorithms with segment optimization.

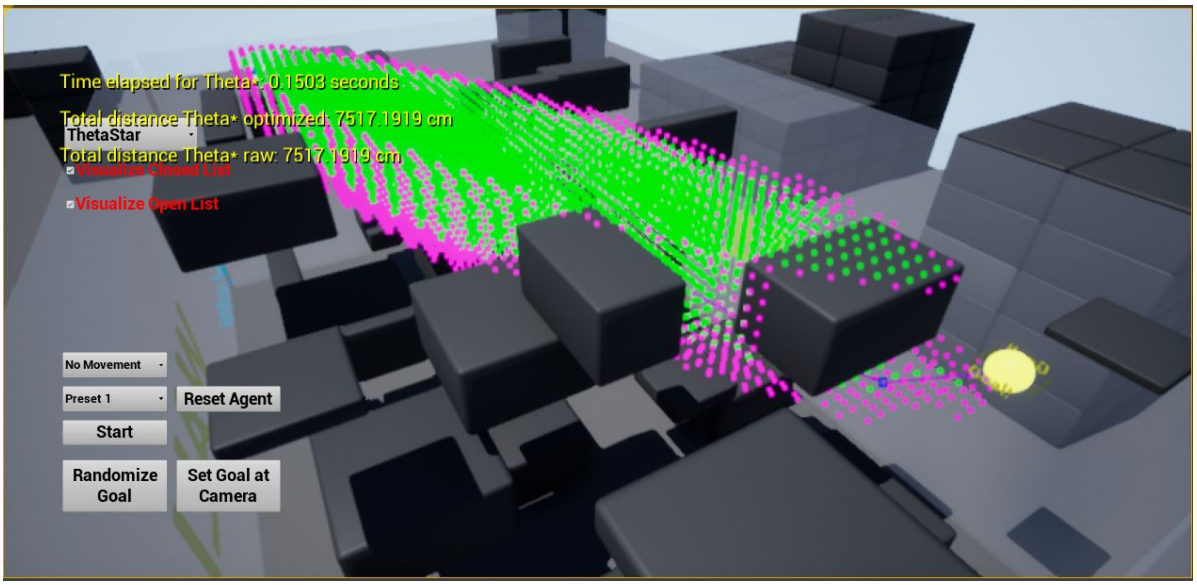

Figure 6 Preset 1 path location and agent location with Theta* algorithms without segment optimization.

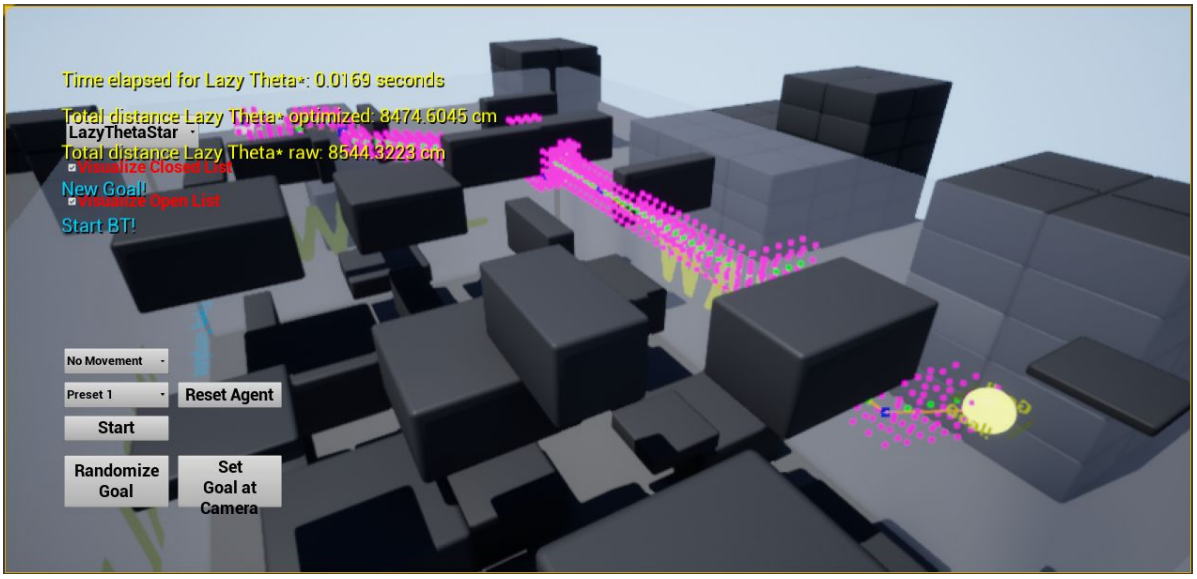

Figure 7 Preset 1 path location and agent location with Lazy Theta* algorithms with segment optimization.

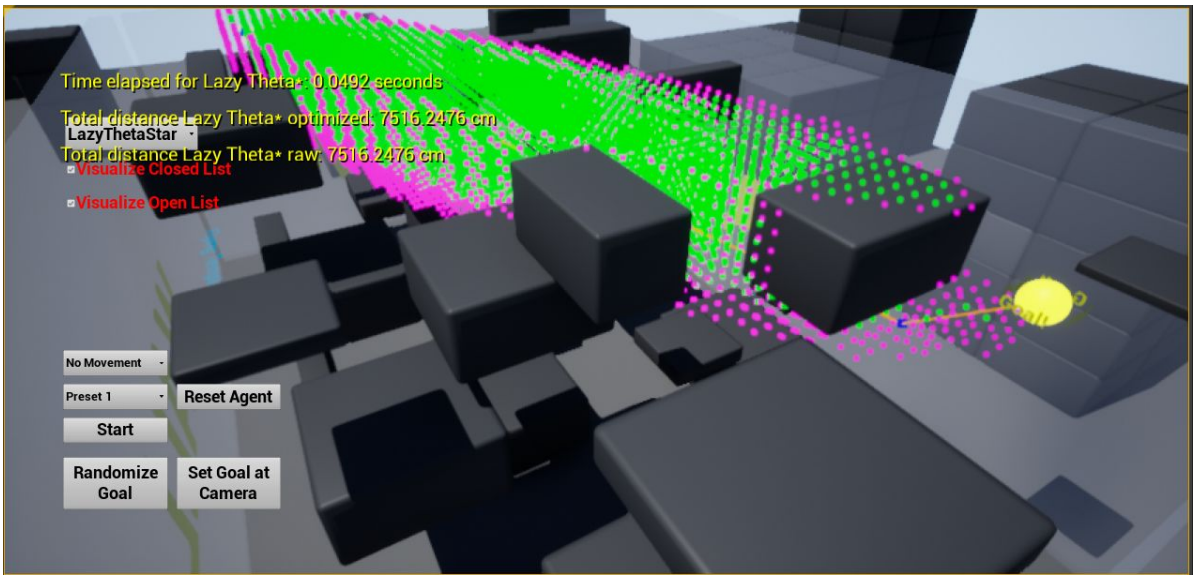

Figure 8 Preset 1 path location and agent location with Lazy Theta* algorithms without segment optimization.

The segment optimization has the ability to reduce the number of nodes that A*, Theta* and Lazy Theta*, which is the main reason that it archives great speeds up. As the comparison of Figure 3, 4, 5, 6 shows, it achieves at least 10 times faster speed for A* and Theta*, while maintaining a relative the same total distance (about 7500 cm) in the rubber banded result. However, this optimization makes Lazy Theta* basically loses the shortest path in terms of preset 1 as the distance increases from 7516 cm to 8575 cm.

Notice that the total time for algorithms with segment optimization on is greater than 16.67 ms because the backthread result is checked by the main thread in the next tick, it implies that the pathfinding finishes within one frame. We will show records of precise timing in the next section.

5.2 Speed and Distance

Times and distances are averaged over 3 tests. All distances come from rubber banded results.

Table 2 Preset 1

Algorithm	Segment Optimization		No Segment Optimization	
A*	18.9 ms	7722 cm	160.7 ms	7595 cm
Theta*	5.3 ms	7723 cm	188.5 ms	7517 cm
Lazy Theta*	2.7 ms	8474 cm	45.4 ms	7516 cm

Table 3 Preset 2

Algorithm	Segment Optimization		No Segment Optimization	
A*	6.2 ms	7932 cm	41.2 ms	7675 cm
Theta*	8.6 ms	7968 cm	89.2 ms	7652 cm
Lazy Theta*	3.2 ms	7933 cm	12.4 ms	7655 cm

Preset 3 is not available because it is a test to demonstrate the algorithm would not waste time on line of sight destination.

Table 4 Preset 4

Algorithm	Segment Optimization		No Segment Optimization	
A*	19.0 ms	6435 cm	31.4 ms	6491 cm
Theta*	8.5 ms	6545 cm	69.1 ms	6431 cm
Lazy Theta*	3.2 ms	6525 cm	11.8 ms	6429 cm

Among all testable presets, Lazy Theta* always perform the best in terms of speed, and it is best for 2 of the 3, that are not line of sight, while segment optimization is off. Even though A* with segment optimization always has the shortest distance, its speed in our test cases are in general 4~5 times slower than Lazy Theta*. Therefore, we believe Lazy Theta* has the best speed to distance tradeoff.

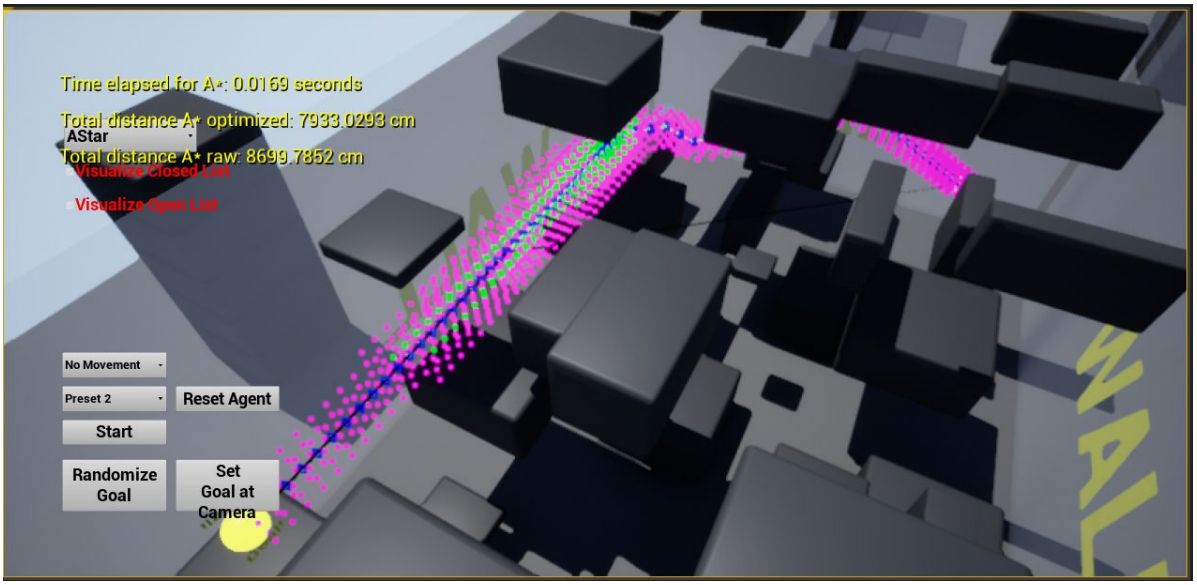

Figure 9 Preset 2 path location and agent location with A* algorithms with segment optimization.

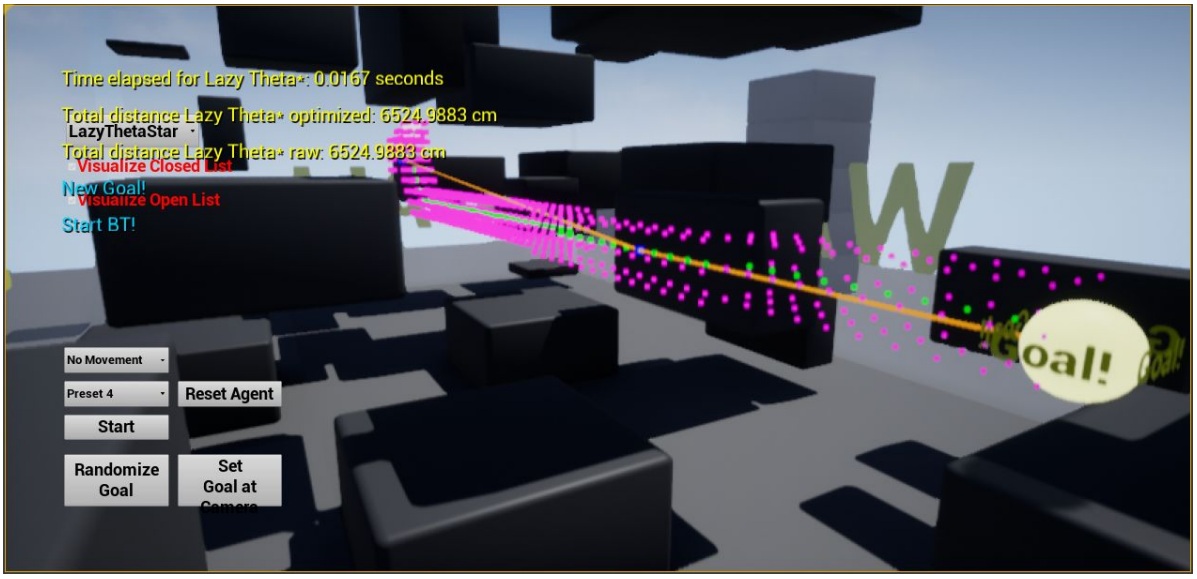

Figure 10 Preset 4 path location and agent location with Lazy Theta* algorithms with segment optimization.

6 Conclusion

Pathfinding is an essential component in navigating in 3D space for video games. We have implemented three graph-based search algorithms, A*, Theta* and Lazy Theta* that operates on a voxelized 3D world. They are efficient in finding an approximated shortest path while executing the pathfinding task in a backthread. We have found an optimization

that uses equalized segment distance that trades accuracy with speed. The tradeoff and differences with all three algorithms are discussed. Lazy Theta* in general has the best speed to distance trade off that should be recommended for 3D pathfinding.

7 Reference

7.1 Online Document:

1. [Ruoqi He & Chia-Man Hung] Pathfinding in 3D Space - A*, Theta*, Lazy Theta* in Octree Structure](https://ascane.github.io/assets/portfolio/pathfinding3d-report.pdf)
2. [Venugopalan Sreedharan] DoN's 3D Pathfinding for Flying AI](https://www.unrealengine.com/marketplace/en-US/product/don-s-3d-pathfinding-flying-ai)
3. [Canis Wong] 3D space path finding { Octree, Boid, A* }](https://forum.unity.com/threads/3d-space-path-finding-octree-boid-a.500142/)
4. [Daniel & Nathan] Benchmarks for Pathfinding in 3D Voxel Space](https://www.cs.du.edu/~sturtevant/papers/voxels.pdf, https://movingai.com/benchmarks/voxels.html)

7.2 Journals:

5. [Peter Hart 68] 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. [Nash 07] Alex Nash, Kenny Daniel, Sven Koenig, and Ariel Felner. Thetaˆ*: Any-angle path planning on grids. Proceedings of the National Conference on Artificial Intelligence, 22(2):1177, 2007.
7. [Nash 10] 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.