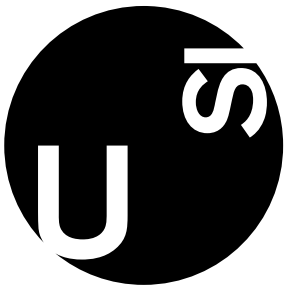


Università della Svizzera italiana

Faculty of Informatics

Bachelor Thesis

June 26, 2022

3D navigation with Octree

Albertini Joy

Abstract

The focus of the project is to add in the game engine Unity a 3D navigation system in a space with obstacles because Unity by default supports only navigation for agents on surface called Nav-mesh. Path planning is usually composed of two parts: [4]

- **Generate graph**: discretize in this case the 3D space into a graph.
- Path planning with computed graph.

Graph generation: To develop such system, we use as a data structure an octree, which discretizes the space aiming at implementing the obstacle avoidance of agents, and defining in which cube an agent can or cannot move. After discretizing the space, we build a dual graph based on the octree. All cubes define a node by its volume centre, there would be an edge of the dual graph by every cube which is in contact with another cube's face. A neighbour could be a larger, a smaller, or an equal sized cube. We can efficiently find the neighbours by generating an ID to all nodes which represent a node location in the octree and octree tree structure. This is the graph which is used by agents to path-plan. To path plan an agent must know in which cube of the octree it is located, since path planning is only performed in nodes of the dual graph. We need to consider that an agent can be in any position in the space and not exclusively on a dual graph node. To find it efficiently we can use the octree, simply by going down in the tree structure by checking in which octant the position of the agent belongs to.

Path planning: An agent being in a game needs to react promptly, so we need a trade-off between the quality of a path and time to compute it:

- **A***: is Dijkstra with an additional heuristic, that is the Euclidean distance to the goal, it is possible to prioritize which node to expand. Paths are composed by edges of the dual graph.
- **Post-smoothing**: It is able to straighten paths already computed by A*.
- **Basic Theta***: It performs post-smoothing during the search to the goal node.
- **Lazy Theta***: Improves computation time of Basic Theta * reducing the number of line-of-sight checks.

Multi-target single-source path planning The most common situation in a game is that multiple enemies will try to get to the player, so I optimized the computation of paths for multiple-targets single-source.

Advisor
Prof. Evanthia Papadopoulou

Advisor's approval (Prof. Evanthia Papadopoulou): Date:

1 Introduction

1.1 Terminology

Definition 1 (Agent) *is an object that moves in the 3D space. It can be in any position of the 3D space, and it is a general term to describe both **source** and **target** of path planning at the same time. Sometimes, in the context of video-games, I will refer to the **source** as the **player** and to the **target** as the **enemy**. The player is usually controlled by the person playing the game, while the enemy need path planning in order to reach the player in the 3D space.*

Definition 2 (max_{depth}) *Is a user defined parameter, which defines the maximum number of recursive subdivisions of the octree in 8 octants (definition 3), and it is the depth of the octree tree structure.*

Definition 3 (Octant) *is one cube of the eight subdivision of a "Euclidean three-dimensional coordinate system defined by the signs of the coordinates" [16], you can see them it in figure 8 with the cartesian system.*

1.2 Objectives

The focus of the project is to add 3D navigation in a space with obstacles to Unity game engine, because not implemented by default in the engine. Currently Unity supports only navigation on a surface called Nav-mesh: this navigation system only allows movement on a surface. This surface can have elevation following the level floor as shown in figure 1. The nav-mesh agent (definition 1) navigation is limited, for example a game located in space couldn't possibly have agents moving through planets. My navigation system would work in that space definition instead, like for example in figure 2. With this system agents such as enemies will be capable of path plan in the 3D space and reach the player.
To have usable navigation of agents we need algorithm with fast execution time, but also path length and path appearance is important since an agent with non optimal path can look dumb or not very reactive. Therefore, a major part of the project will focus on the evaluation of algorithms and their properties. A common situation in a game is that multiple enemies will try to reach the player, so I optimized the calculation of multiple-source single-target path planning, but the problem gets actually inverted in **single-source multi-target** which is easier, so i will refer to the enemies as targets and to the player as the source, refer to definition 1. To create such system, we have multiple objective:

- **Octree creation** : Discretization of the 3D space in blocked and unblocked cubes, more efficient compared to a grid.
- **Dual graph creation** : From the Octree cubes generate the dual graph (graph used to path plan).
- **Find agent nearest node of the dual graph** : Agents can be in any position of the 3D space, and to path plan they need to connect to the dual graph.
- **Path planning** : Multiple algorithm implemented and tested: Dijkstra, A*, Greedy Best-first-search, Post-smoothing, Basic Theta*, and Lazy Theta*.
- **Multi-target single-source optimization** : All path planning algorithms will work with this optimization.
- **Movement of agents**

1.3 Background

Unity [14], developed by Unity Technologies, is a very versatile game engine. It is used in different fields such as Architecture, Automotive and Film industry, it can handle the production in 2D and 3D environments, it supports a plethora of system IOS, Androis, Windows, Mac OS, Playstations, Virtual reality system such as Oculus and Microsoft Hololens and Smart TVs. I have used mainly Unity for collision detection in this work.
Paper [2] will give us a solution on how to compute neighbours of a cube in order to create the dual graph, giving each octant an ID. This will be used to compute the dual graph. Papers from the same author Nash Alex [5], [1], [6] and [4] (will contain all 3 previous papers and much more information), contains very detailed explanations and analysis on technique to represent the 2D and 3D environment for path planning in video games (but not the octree), and the path planning algorithms that I developed **A***, **Post smoothing** and **Any-angle path planning algorithms**.

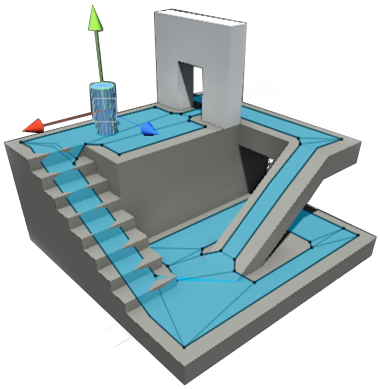


Figure 1. Nav-mesh in blue, agent is represented as a cylinder [15]: Unity Nav-mesh

1.4 Results

We have effectively created a working system for agents to navigate in 3D in Unity, with agents that moves realistically using **Any-angle path planning** or algorithm using **Post-smoothing**. The octree effectively optimizes the space complexity and reduces the time computation of path planning algorithms compared to cubic grids, while the multi-target single source optimization performs better in mostly all scenarios than calculating the path for each target singularly.

2 Octree Data structure

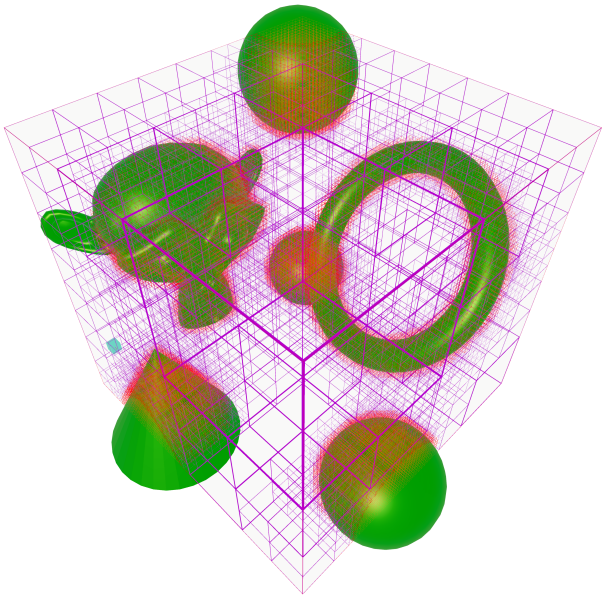


Figure 2. Octree in purple, max depth 6, obstacles in green, collision cubes in red

2.1 Octree definition

Octree is a tree data structure in which each node of the tree has eight children. Octree is used as a hierarchical technique to discretize 3D space [4]: it recursively subdivides itself in 8 **octants** (definition 3) on necessity. It discretizes with smaller cubes the space in the presence of obstacles, thus where more precise representation is needed to be able to travel in the space; and larger cubes in areas where obstacles are less present (open areas) [4]. An octree is the three dimensional equivalent of quadtrees and its mainly used in 3D graphics and game engines [20].

2.2 Octree recursion

In my implementation of the octree, if a node's cube collides with an obstacle in the space, it will split itself in eight **octants** each defining a node of the tree. Subsequently if those nodes' cubes collide with an obstacle they will divide themselves in eight octants and so on until reaching a user defined maximum number of subdivisions (you can see this process in figure 3). The user defined value is called max_{depth} (definition 2) since it will define the depth of the resulting tree. The start cube $size$ and $position$ is user defined, and each time we split we divide by two this size. This recursive implementation will lead to a structure with different cube's sizes, smaller when near an obstacle and larger when further from them, as is shown in figure 2 and figure 3.
If you compare figure 2 and figure 10 you can see the correlation with the discretization of the space by the octants and the tree data structure.

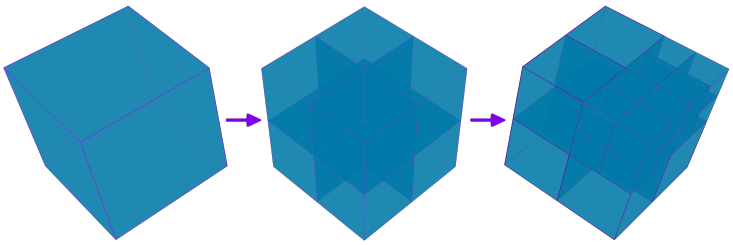


Figure 3. Octree subdivision recursion

Min-depth

There is another user defined parameter min_{depth}, which will give to the user more control on the octree creation, by defining a minimum number of subdivisions. Consequently, an octree's node will subdivide itself even if it doesn't collide since it has not reached the minimum depth. When the min_{depth} is equal to max depth the octree will become a grid.

2.3 Octree implementation

2.3.1 Octree recursion

Each octree node will store:

- Variable $position$ in the 3D space, which is the centre of the cube.
- Variable $size$ of the cube.
- Variable $depth$ in the tree.
- Variable $node_{ID}$, an ID for the node defines its relative position in the space from the root node, refer to section 3.3.1.
- Set of child nodes, variable called $child$, maintain the tree structure of the octree.
- Set of neighbour nodes, variable called $neighbours$, when the **dual graph** 3 is computed, it will contain the neighbour nodes.
- A booelan $leaf_{node}$: is true if the node doesn't collide with obstacles and has reached min_{depth} , making it a leaf of the octree tree structure.
- A boolean $valid_{descendant}$: is true, if at least one of the descendant nodes is a $leaf_{node}$.

Root node

It starts by a creation of the root node, in which the user choose the $position$ and the $size$ of the start cube (most external cube wireframe in red in figure 2). If it collides with an obstacle, it will subdivide itself in 8 octants (definition 3), defining 8 child nodes:

Child nodes

Each time there is a subdivision, those node's variables are computed:

- The $position$ is relative to the parent node position, it's computed by increasing or decreasing by $1/4$ of the parent's $size$, the $position$ will be the centre of the volume of the cube. The different combination of increasing or decreasing will define the 8 octants $position$, for example the position of cube 101 shown in figure 8 is calculated:
  - $child.position.x = parent.position.x + (parent.size/4)$
  - $child.position.y = parent.position.y + (parent.size/4)$
  - $child.position.z = parent.position.z + (parent.size/4)$
- $node_{ID}$ is computed during the recursion, explained in section 3.3.1, later used to compute the octree dual graph.
- $size = \frac{parent.size}{2}$.
- $depth = parent.depth + 1$.
- $child$ if this cube collides it will subdivide itself in 8 octants, and setting them as child nodes.
- $leaf_{node}$: true if it doesn't collide with obstacles and has reached min_{depth} subdivision.
- $valid_{descendant}$: is true, if one of the descendant nodes is a $leaf_{node}$.

2.3.2 Collision detection

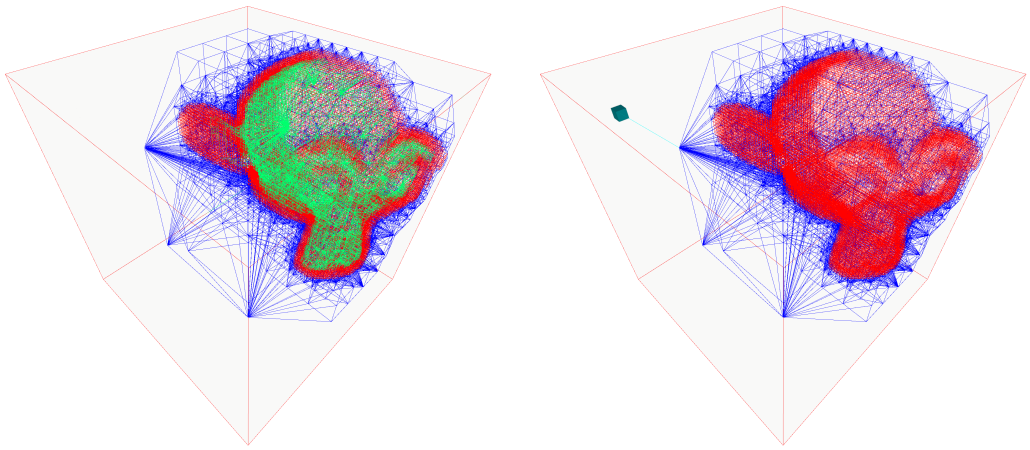
During the octree recursion we need to check if a cube collides with an obstacle. For detecting collisions I'm using the **built-in** Unity function $Physics.OverlapBox$ [12], which checks for collisions by placing a "checking cube" of the same size and position of the current node.2.3.1.
Convex shape meshes: is a mesh where any two vertices of the mesh, a line between them will be contained by the shape of the mesh, like cubes, spheres, rectangles ecc ... [8] **Concave shape meshes** are the opposite of **Convex shape meshes**, you can see a concave shape mesh in figure 4 a).

- **Concave shape meshes**: If an obstacle is not a convex shape mesh, the built-in function will consider a collision with the obstacle only on the surface of the mesh, but not inside of it. This means that if the "checking cube" is situated completely inside the mesh it will not be considered as a collision even though it should. Therefore, if there are nodes inside the mesh, those nodes will need to be removed (later explained).
- **Convex shape meshes**: Cubes which are completely inside the obstacles are correctly considered collisions.

Cleaning of nodes

After discretization the octree will contain branches and nodes which are useless since they collide with an obstacle and all of their descendants are colliding leaves of the octree. Those nodes are both non $leaf_{node}$ and non $valid_{descendant}$ (refer to section 2.3.1), and they will be removed. Nodes of the octree that are not $leaf_{node}$ but are $valid_{descendant}$ are kept instead, because they are necessary to reach a $leaf_{node}$ going down to the tree structure. After the cleaning of those nodes the octree could look like the figure 5, where red vertices are $leaf_{node}$ and blue vertices are not $leaf_{node}$ but are $valid_{descendant}$ vertices. This process will reduce the space complexity. This final octree structure has the property that every branch will always lead to a $leaf_{node}$.
We have two situations:

- **All obstacles are convex shape meshes**: In this situation we simply traverse the octree removing the nodes that are both non $leaf_{node}$ and non $valid_{descendant}$.
- **Not all obstacles are convex shape meshes:** As we said before, in case of non convex shape meshes only cubes that collides with surface of the mesh are correctly considered as collisions. When a cube defined by the Unity function $Physics.OverlapBox$ is completely inside an obstacle it will be not detected as a collision, so there will be nodes inside the mesh. To remove those nodes, we need a more complex solution.



(a) Non convex shape mesh, resulting in dual graph with two connected components

(b) Non convex shape mesh, dual graph after the cleaning there will be only one connected component

Figure 4. Non convex shape mesh problem

Cleaning nodes inside meshes

Let's consider figure 4 a), we have a single non convex shape mesh, you can see after the calculation dual graph (section 3) we have **two connected components** because there are nodes inside the mesh, due to the Unity function $Physics.OverlapBox$. Let's consider figure 4 a) we would like to remove the nodes composing the green graph connected component. A user needs to define which connected component he wants to keep. The tan box in figure 4 b) is used for this purpose; a user will place this box in a cube belonging to connected components that he wants to keep (the thin cyan line from the box in figure 4 b) represent the connection to that dual graph). We will still compute the dual graph for both the connected components, after that we perform the cleaning of unwanted nodes by finding the nearest node in the dual graph of the tan box (refer to section 4) and execute BFS (Breadth-first search) from this node to find all nodes of the blue connected components and remove the others.

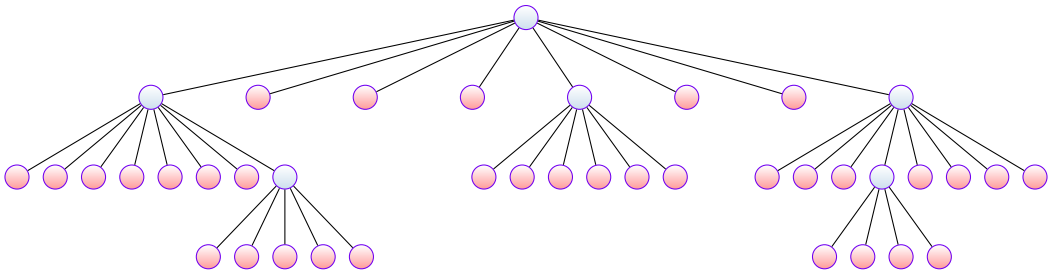


Figure 5. Final Octree structure

2.3.3 Mesh generation

I couldn't use **Unity gizmo** to represent the octree, gizmos are the Unity' solution to represent debug information. Gizmos have a simple implementation that let you draw a wire cube but is drawn frame by frame. The octree represented in figure 6 has more than 8000 cubes, and drawing it frame by frame would make the editor view in Unity lag and crash. I need another way to represent the octree in engine, therefore I decided to generate a mesh of all the cubes. For each node in the octree which is a $leaf_{node}$ (refer to 2.3.1), a primitive cube is placed in the node's $position$ and with node' $size$. Once is created it takes the mesh information and saves it into an array. After that the primitive cube is destroyed. Then all meshes information in the array are combined together generating the octree mesh by using Unity Built in function $Mesh.CombineMeshes$ [11].

Octree shader

To have a visible and usable octree representation I need a wireframe shader. Such shader would show only the edges of the mesh. I slightly modified a wireframe shader that I found online [10].

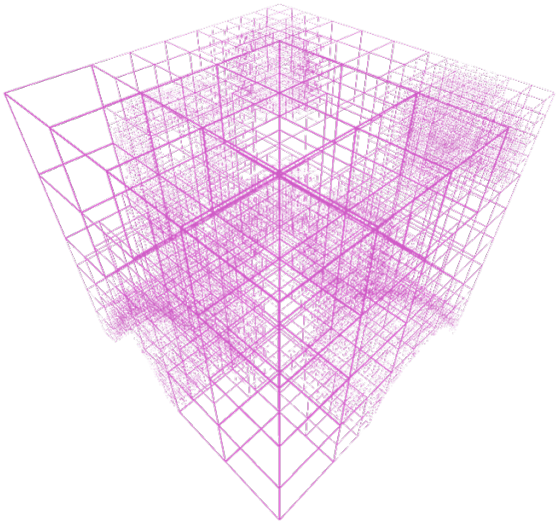


Figure 6. Mesh of the octree

2.3.4 Octree space and time complexities

Space complexity

The octree will subdivide further the octants only on necessity defined by the obstacles, this optimizes the space complexity since will reduce the number of nodes needed to represent the agent (definition 1) navigation space and subsequently also the pathfinding computation due to a smaller number of nodes to consider. This is critical in 3D navigation, because data to store increment drastically compared to the same problem in 2D dimension. The space complexity (number of nodes) of the octree depends entirely on the world definition, for example in a world in which obstacles occupy most of the space it will result the octree to become a grid.
Worst case is when the octree becomes a grid, and we have the minimum number of nodes removed by the clean function (section 2.3.2). It will happen when obstacles have the size of the minimum octant (definition 3) and are placed in an exact position such that the octree will be forced to subdivide always, becoming a grid. In this particular case the number of nodes removed by the cleaning section collision-detection will be exactly the number of obstacles, thus we have:
Worst case space complexity $= O(8^{max_{depth}} - obstacles)$, where max_{depth} is defined here 2, and $obstacles$ is the number of obstacles defined as above.

Time complexity

Compared to the space complexity we need also to consider the collision detection complexity, which is unfortunately not stated by the Unity documentation, the only knowledge of how it operates is that it uses a bitmask.

Unfortunately we have no knowledge if we are completely inside an obstacle, so we keep on splitting in octants until reaching max_{depth}, thus we have a worst case complexity when an obstacle occupy the entirety of the space that should be discretized:
Worst case time complexity $= O(8^{max_{depth}})$, max_{depth} is defined here 2.

3 Undirect octree dual graph

3.1 Octree Dual graph definition

After discretizing the space, we define a dual graph based on the octree as is shown in figure 7: **Nodes** of the dual graph are $leaf_{node}$ 2.3.1 of the octree, a node $position$ as stated previously is the volume centre of its cube. There would be an **edge** of the dual graph by every cube which is in contact with another cube's face. The neighbours to a node are each of the cube's nodes that touches one its 6 faces (example red cube neighbours in figure 7 are the coloured cubes). Since the dual graph is constructed with an octree compared to a 3D grid a neighbour could be a larger, a smaller, or an equal sized cube. This is the graph used by agents later to path-plan.

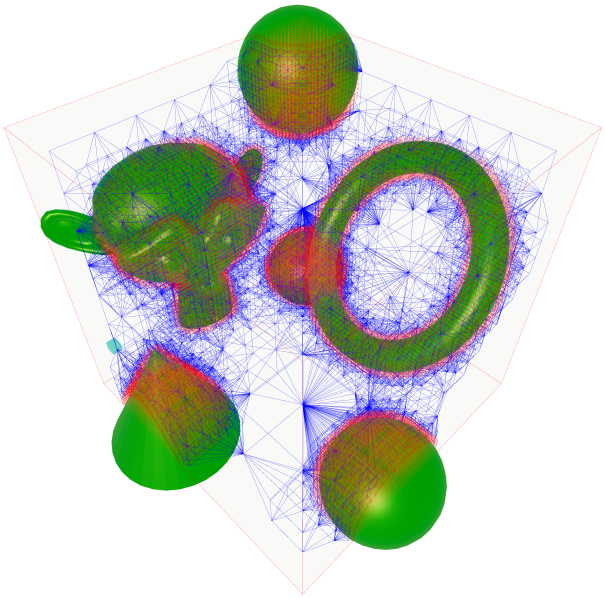


Figure 7. Octree dual graph in blue

The resulting dual graph is an **Euclidean graph**, which is a graph in which each node has a coordinate (the $position$); edges' length is the Euclidean distance between the two nodes, definition from [19] [4].

3.2 Octree Dual graph general implementation

We can efficiently find the neighbours by generating an ID of nodes while creating the octree and store them into a dictionary. A node ID embodies the node location in the space and the octree tree structure. The dictionary data structure allows us to efficiently query and retrieve nodes using simply the ID. The ID allows us to suppose the ID of a neighbour simply by shifting bits and later check if it exists using the dictionary.

3.3 Octree dual graph implementation

3.3.1 nodes'ID

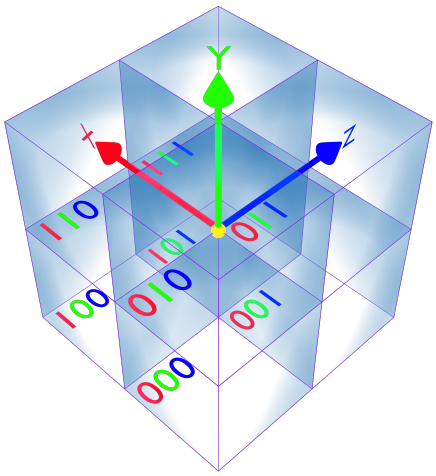


Figure 8. Last triplet of bit for the octants

To efficiently compute neighbours of a given node we need to assign an ID to each node, this ID is called $node_{ID}$, is computed during the octree recursion, a $node_{ID}$ of a new node is computed by stacking an additional triplet of

bits to the copied parent's $node_{ID}$. Therefore, a node has knowledge of each of its ancestors' IDs through its $node_{ID}$. The additional triplet of bits represents the position of the child node 3 relative to the parent (at centre of figure in yellow 8), the first bit represents the x axis, the second the y axis and the third the z axis, each of the bits in the triplet get assigned a value (0 false, 1 true) depending on the position of the child: For the bit representing x axis:

- $x = 1, if\ child.position.x > parent.position.x$
- $x = 0, if\ child.position.x < parent.position.x$

For the bit representing y axis:

- $y = 1, if\ child.position.y > parent.position.y$
- $y = 0, if\ child.position.y < parent.position.y$

For the bit representing z axis:

- $z = 1, if\ child.position.z > parent.position.z$
- $z = 0, if\ child.position.z < parent.position.z$

You can see this computation of those triplets for each octant in figure 8, remember that the position of the child node is the volume centre. So due to the octree recursion the $node_{ID}$ will be a list of triplets of bits, example 000.011.110 (is the $node_{ID}$ of the violet cube in figure 10). The root node will always have $node_{ID}$ 000. The $node_ID$ of a node contains the IDs of all ancestors for this node, every 3 bits sequence excluding the last one, representing a parent or an ancestor, you can see tree structure in figure 10. Let's consider $node_{ID}$ of the violet cube will contain:

- 000: root node (ancestor).
- 000.011: (parent)

3.3.2 Dictionary data structure

While computing the octree we will store each $leaf_{node}$ 2.3.1 by key $node_{ID}$ and value the entire node in a dictionary. Given a $node_{ID}$, a dictionary will allow us to:

- check the existence of a node in $amortized\ O(1)$
- retrieve the node in $amortized\ O(1)$

3.3.3 Neighbours node

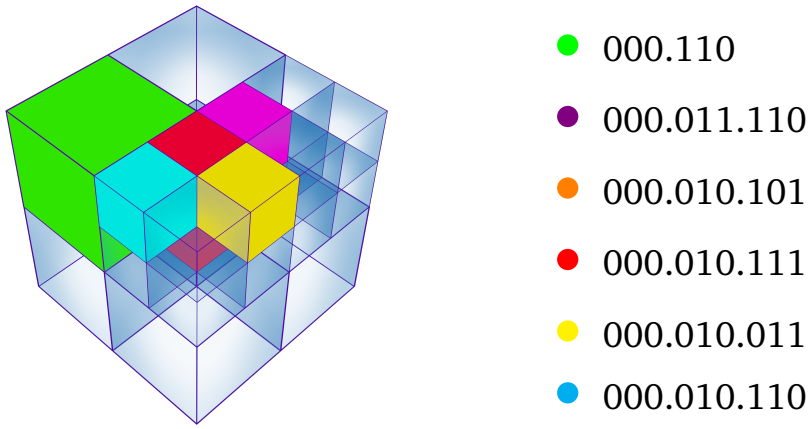


Figure 9. Red cube neighbours

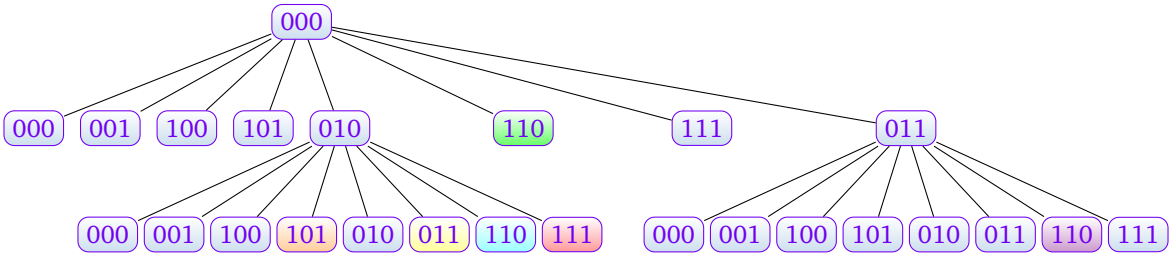


Figure 10. Octree, tree structure

Neighbours of a node are all nodes which cubes touch one of the 6 faces of the given cube of the node, you can see the neighbours of the red cube in figure 9, a node can have 1 to $n-7$ neighbours:

- A cube with all neighbours'cubes of the same size will have 6 neighbours, one for each face.
- A larger cube with smaller neighbours'cubes can have up to $n-7$ neighbours.
- Some nodes will have less than 6 neighbours because they are located in a corner of the octree or are placed near to an obstacle.

To find all neighbours of one octant we exploit its $node_{ID}$. Based on its value alone we can compute the supposed $node_{ID}$ of the neighbours. "Supposed" since this computed neighbour $node_{ID}$ could be not existing. Every time we suppose a $node_{ID}$ we check the existence in the dictionary, if the $node_{ID}$ exists its node will be added to the list of $neighbours$ 2.3.1 of that node, and also the inverse to make an undirect graph. Thus to compute each node list of $neighbours$ we iterate in each $leaf_{node}$ 2.3.1, computing:

Same size cube same parent

We can simply find the neighbour with same size cube same parent, by inverting one at a time for each axis bit of the considered $node_{ID}$ last triplet bits. Example, let's consider figure 9 and figure 10, consider $node_{ID}$ of the red cube and red node in the tree: 000.010.111

- Neighbour in x axis: 000.010.011 ($node_{ID}$ of yellow cube and yellow node)
- Neighbour in y axis: 000.010.101 ($node_{ID}$ of orange cube and orange node)
- Neighbour in z axis: 000.010.110 ($node_{ID}$ of cyan cube and cyan node)

Complexity: $3 \cdot ammortized\ O(1) = ammortized O(1)$

Same size cube different parent

Bit flip level axis [2] The bit flip level is the number of times the considered axis bit (x or y or z) starting from the last triplets of 3 bits up to the root triplet (not considering it) doesn't change, also counting the first that changes:

- the bit flip level in the z direction of 000.111.010.111.101 is 3
- the bit flip level in the y direction of 000.111.101.001.100 is 4

To find a neighbour with same size cube but different parent we need to find the bit flip level, then flip all bits considered axis in the bit flip level. This process is repeated for each of the 3 axes: Example in figure 9 and figure 10, red cube $node_{ID}$: 000.010.111:

- In x 000.010.011 → invert bits 000.110.011 (neighbour doesn't exist)
- In y 000.010.111 → invert bits 000.000.101 (neighbour doesn't exist)
- In z 000.010.111 → invert bits 000.011.110 ($node_{ID}$ of violet cube and node)

Complexity worst-case: $3 \cdot O(max_{depth} - 1)$ for bit flipping, plus $3 \cdot amortized\ O(1)$ for retrieving from dictionary, thus the final complexity is amortized $O(max_{depth} - 1)$, max_{depth} is defined here 2.

Larger size cube parent or ancestor

In the case we don't find a neighbour node using the $node_{ID}$ computed for one of the axes in the above paragraph **same size cube different parent**, it means that the neighbour's cube could be larger, smaller, or could not exist, thus we need to try to find those neighbours. To find a neighbour's node with a larger cube box, we simply need to check whether a **parent** or **ancestor** exists in the dictionary iteratively. Thanks to $node_{ID}$ that embodies the tree structure of the octree, then we can simply remove:

- the last 3 bits to find the **parent**
- the last two sequence of 3 bits to find the first **ancestor**
- the last three sequence of 3 bits to find the **second ancestor**
- and so, on up to the **root**, but not considering the root.

If we iterate up to the root, it means that a neighbour doesn't exist. Example considers red cube with $node_{ID}$ 000.010.111 in figure 9 and figure 10:

- Consider 000.110.011 $node_{ID}$ of same sized neighbour that doesn't exist as stated in the paragraph above**same size different parent**, the parent with $node_{ID}$ 000.110 exists is the green cube in figure 9 and is the correct neighbour.
- Consider also 000.000.101, parent doesn't exist, and root shouldn't be considered. So, there is no neighbour for the top face which is correct since the red cube has no cube on the top of it.

Complexities in worst-case: $3 \cdot O(max_{depth} - 1)$ for removing 3 bits sequences for every axis, plus $3 \cdot (max_{depth} - 1) \cdot amortized\ O(1)$ for retrieving from dictionary, so the final complexity is amortized $O(max_{depth} - 1)$, max_{depth} is the defined here 2.

Smaller size cubes

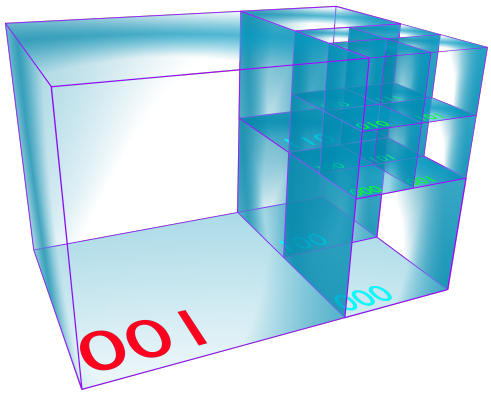


Figure 11. Small sized octant last triple of bits

In my implementation I don't search for smaller sized cubes when searching for neighbours' nodes, I only search for larger sized cubes while iterate in all $leaf_{node}$ instead: that's because a smaller sized cube will eventually find as a neighbour the larger sized cubes, and as described above once we find a neighbour we will set in both directions.
To find a smaller sized cube neighbour, we need to add to our $node_{ID}$ the considered octant triplet bit code that we want to retrieve, refer to figure 8. Example: Let's consider for example 000.000.001 the $node_{ID}$ of the largest cube in figure 11. First, we need to suppose the $node_{ID}$ of a neighbour with same size cube with different parent as in section **same size cube different parent**, which will be non existent in the dictionary because it subdivided itself in smaller 8 octants, so 000.000.000. After that based on our last triplet code 001, we can determine the smaller neighbours cube $node_{ID}$ triplet bit code in cyan in figure 11.

- 000.000.000.000 → exist in dictionary
- 000.000.000.100 → exist in dictionary

- 000.000.000.010 → **dosen't exist in dictionary**
- 000.000.000.110 → exist in dictionary

For the supposed $node_{ID}$ 000.000.000.010 is not present in the dictionary, because it divided itself further in 8 octants, so we need to try also IDs at two level more in depth. Consequently, we need to try to retrieve 4 new neighbours $node_{ID}$ (in green in figure 11):

- 000.000.000.010.000
- 000.000.000.010.010
- 000.000.000.010.100
- 000.000.000.010.110

In the worst case for a neighbour, we need to generate IDs and query the dictionary: $3 \cdot 4 \cdot max_{depth} - 1$ times (where max_{depth} is 2), which is 4 times more expensive than finding only larger cubes, that's the reason for searching only parent neighbours in my implementation.

3.4 References

The idea and concept of assigning IDs to the octants, bit flipping and to find efficently the neighbours is taken by source [2].

4 Find agent nearest node

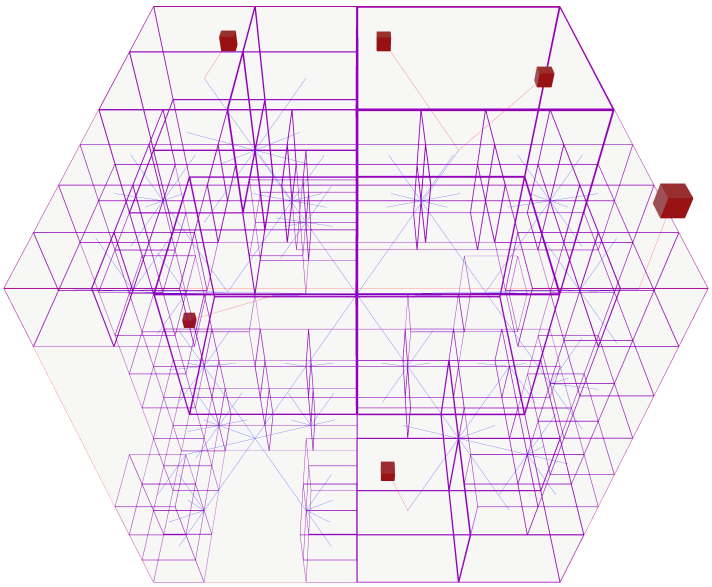


Figure 12. Octree cubes in violet in relation with the octree tree structure shown in blue, agents in red cubes, connection to node's cubes are shown in red.

To path plan an agent (definition 1) must refer to a dual graph node, this dual graph node is the nearest to the current position of the agent. This necessity is due to the fact that path planning is only performed in nodes of the dual graph. An agent can be in any position in the 3D space not exclusively on a dual graph node position as is shown in figure 12, remind that a dual graph node is $leaf_{node}$ of the octree refer to section 2.3.1. To find it efficiently we can use the octree tree structure, simply by going down in the tree structure by checking in which of the nodes an agent is closest to.

4.1 Find agent nearest node implementation

To find an agent closest dual graph node which is equivalent as $leaf_{node}$, we can go down the tree structure of the octree. Is important to note that the tree structure considered is the one after the removal of unnecessary nodes refer to section 2.3.2. This updated octree structure has the property that every branch will lead to a $leaf_{node}$. The algorithm Agent nearest node 1 traverses the tree structure branching to the tree node with minimum distance to the agent until reaching a $leaf_{node}$ which is a node of the dual graph. This algorithm is the reason why we kept the full octree tree structure and not only the nodes which are($leaf_{node}$).

Algorithm 1 Agent nearest node(root, agent)

```
nearest_node ← root
while octant ≠ leaf_node do
    min_node ← null
    min_distance ← ∞
    for each child ∈ nearest_node do
        current_distance ← EuclideanDistance(child.position, agent.position)
        if current_distance < min_distance then
            min_node ← child
            min_distance ← current_distance
        end if
    end for
    nearest_node ← min_node
end while
return octant
```

4.2 Find agent nearest node complexity

In the worst case we simply going down the tree structure from the root, the depth of the tree is defined by max_{depth} (definition here 2), for each of the level of the tree to decide the branch we need up to 8 interactions.
Worst case complexity $= O(8 * max_{depth}) = O(max_{depth})$

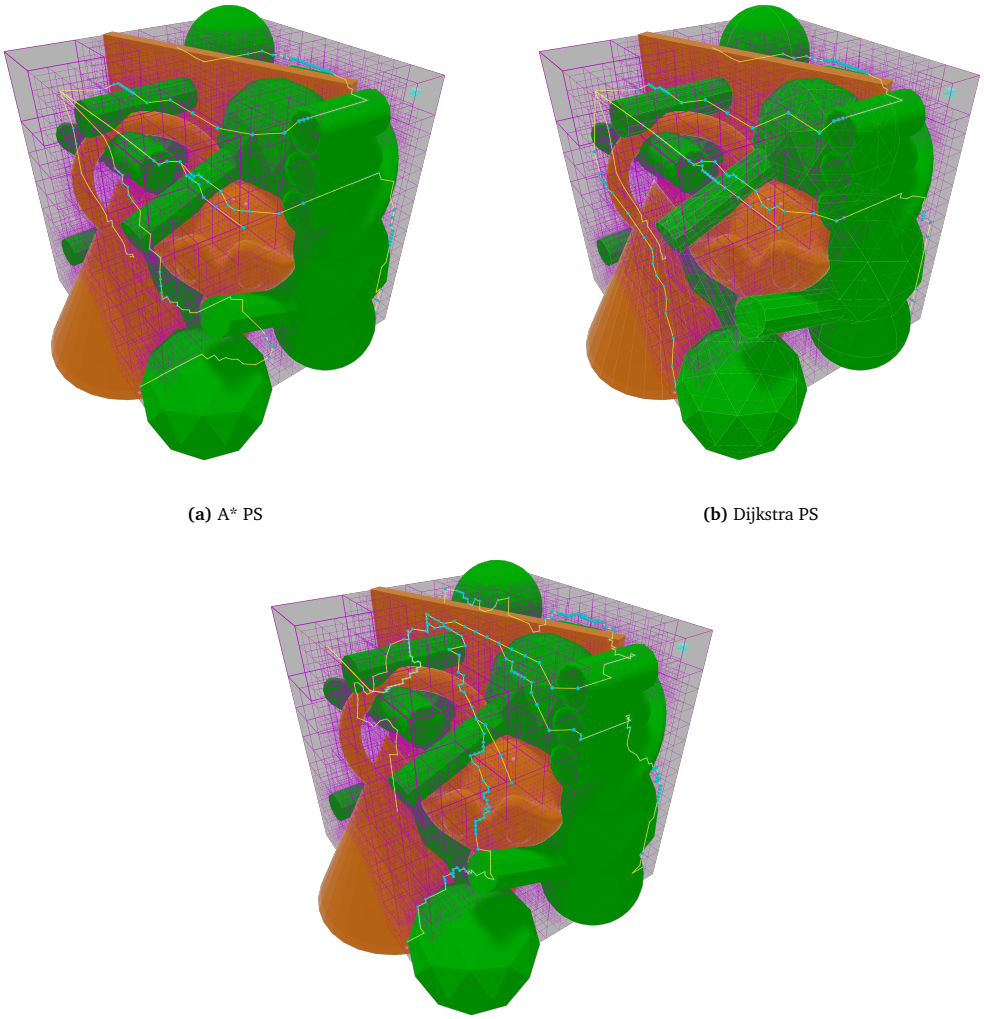
5 Path planning

Now that we have a usable graph to path plan and that we know in which of the nodes of the dual graph the agents (definition 1) connect (refer to section 3), we can finally path plan between **source** and **targets** . Path planning is computed at runtime this means that it needs to be as efficient as possible in terms of time, because an agent must react promptly to a new situation for example in the context of a videogame an enemy that has spotted the player needs to start following it promptly to be able to reach it. Thus, we need trade-off between the length of the path and the time to compute it, usually shorter paths need more time to be computed. For example, in a graph with many nodes, Dijkstra algorithm would be too demanding in terms of computation. this is where **A* algorithm** comes into play having an additional heuristic which is the distance to the target. It makes the research more informed in which region of the graph to expand to reach the target. **Greedy Best-first-search** on the other hand will only expand nodes based only A* additional heuristic, making it very fast since it will expand very few nodes compared A*, but the length of the path will increase substantially and also path look gets way less realistic.
The algorithm above mentioned can compute path that are composed only by edges of the dual graph called **edge constrained paths** [4], and the algorithm that computes those are called **edge constrained path planning algorithms** [4]. The path computed by those algorithms will have a lot of turns since the dual graph is grid like being constructed from an octree, the multitude of turnings will give an agent following the path a robotic look.

The edges of the dual graph are guaranteed to make an agent not collide with obstacles. **Post-smoothing** is algorithm applied as after the calculation of an **edge constrained path**, this algorithm is capable of straightening the paths by checking for new smaller edges between the nodes composing the path, effectively removing nodes from the path that are in between the new edge; we should remember that in the 3D space there are obstacles so we need to check if the agent will not collide to an obstacle moving through the new edge, **Line-of-sight** will do exactly this, check if the new line between two nodes will not collide with obstacles, you keep in mind that Line-of-sight checks are costly. Post-smoothing will enhance the path look making a path with as few turns as possible and decreases path length, already with post smoothing we have a usable system with realistic paths.

To further improve the path length we could embody **Post-smoothing** during the search of the target by effectively considering each node expansion two paths the dual graph path and the "Post-smoothing path", those algorithm are called **Any-angle path-planning algorithms**, the compute path are **Any-angle paths**: are "paths composed by line segments whose end points are nodes on the dual graph" (definition from [4]) Those are Any-angle-path-planning algorihms are **Basic Theta*** and **Lazy Theta***, Lazy Theta * is an improvement to **Basic Theta*** that reduces the number of Line-of-sight checks, thus reducing time to compute. All those algorithm are guaranteed to work in our dual graph since they are guaranteed to work in any Euclidean graph [6], [4].

5.1 Edge constrained path planning



(a) A* PS

(b) Dijkstra PS

(c) Greedy Best-first-search PS

Figure 13. Edge constrained path planning, 1 souce, 5 targets

We will define three algorithm that will find edge constrained paths: **Dijkstra**, **Greedy Best-first-search**,**A***.

5.1.1 Dijkstra

I didn't develop **Dijkstra** as standalone algorithm, I can simply derive the Dijkstra algorithm by setting h parameter of A* value to 0, and g value to 1 (refer to 5.1.4), this way we will use only the **G(a,b)** function of A* that calculates the distance from the source. The distance from the source for each node is stored in the parameter value $gScore$. The dual graph is "weighted indirectly" meaning the weight of each edge is the **Euclidean distance** between the two vertexes, those values are not stored instead are calculated at the moment of node expansion. Path found by Dijkstra are true shortest path (in the case of **edge constrained paths**), but runtime will be the highest compared to the other

two, path look is the best possible for path composed of edge of the dual graph, since path length are the shortest possible. You can see the paths for 5 targets in figure 13 b).

5.1.2 Greedy Best-first-search algorithm

"**Best-first search** is a class of search algorithms that will expand always the a node that fit at best a given rule [18]", **Greedy Best-first-search** is an algorithm that searches only using an heuristics based on the supposed distance to the target to expand, the closest node to the target is judge by the heuristic function, and will be expanded next [18]. As for Dijkstra I didn't develop it by a standalone algorithm, I derived **Greedy Best-first-search** from **A*** algorithm, by setting the parameter of A* g value to 0 and h to 1 (refer to 5.1.4). The heuristics function is **H(b)** it approximates the distance to the goal by using **Euclidean distance** between current node b and the goal node. The function is an approximation since it has no knowledge of the obstacles. Greedy Best-first-search is the opposite of Dijkstra, the runtime will be the fastest, but path look and length will suffer, as you also can see in figure 13 c)

5.1.3 A*

A* is a path-planning algorithm used mainly in robotics and video games [1], A* belongs to the class Best-first-search [17], it aims at finding path in terms of trade-off between length of the path and time to compute, A* can be thought as variant of Dijkstra that uses additional heuristics knowledge the **H(b)** function, making it more informed compared to Dijkstra [4], such that will bias the search through areas closer to the target, instead of searching without direction as in Dijkstra, so reducing the number of nodes expanded by the algorithm. The **H(b)** is an approximation of the distance to goal node from node b, different H function can be implemented for A*, in my case the H function is the Euclidean distance from the current expanded node b to the goal node. A* will use The **F(a,b)** function to choose which node to expand next, which is calculated as **F(a,b) = G(a,b) + H(b)**, where $G(a,b)$ as described in section 5.1.1 is the distance from the source, embodying the **H(s)** function will make F(a,b) be an approximation of length of shortest path from the start node through node b to the goal node [1]. Making A* an in-between algorithm respect to **Dijkstra** and **Greedy Best-first-search algorithm**, also in terms of time and length of the path, although path will still look realistic as Dijkstra. Consider figure 13 a)

5.1.4 A* implementation

A* works by assigning a scores to nodes, the score is the $fscore$ calculated by the $F(b)$ function, this is done in the $UpdateNodePriorityA$ algorithm 3. A* will expand next always the node with lowest calculated $fscore$ in the main loop algorithmAstar, using a priority queue to do it efficiently.

A* calculation of node score

A* computes a score for each node expanded node by using $F(a,b)$ function: Let's consider current expanding node a, a neighbour node b, and the node that we want to reach c the goal node. A* will calculate the priority of the node to by the function:

$$F(a,b) = G(a,b) + H(b)$$

$$G(a,b) = a.gScore + (g * EuclideanDistance(a.position, b.position)$$

$$H(b) = h * EuclideanDistance(b.position, c.position)$$

The value of G will be store in the node in $gScore$, the value of F will be store in $fScore$ of the node. Where: **G** computes the distance from the source for node b adding **a.gScore** will be the shortest path from the $startnode$ to the node a that we have found. **H** is the distance from the current visited node b to the $goalnode$, it is the added heuristic that differentiate it from Dijkstra, The H function is not exact since as no knowledge of obstacles in the space, for example in figure 18 will make expand nodes in the top middle corner of the image but those are blocked by the obstacle. The computation of node scores is shown in the algorithms 3, 5, 6 7.

Weighting the score

The user has two float parameters g and h that can influence the calculation, both values can assume as minimum value 0 and maximum value 1, by setting the values by the extreme we can transform **A*** in different algorithms.

- g = 1 and h = 1: standard A* configuration
- g = 1 and h = 0: doesn't consider **H(b)** score, becomes **Dijkstra**

- g = 0 and h = 1: consider only the heuristics fucntion **H(b)** becomes **greedy best-first-search**

A developer can tweak g and h value to fit better a particular graph, thus A* is easily adaptable, for this reason is popular as a path planning solution in videogames [9]:

- Setting g to a value closer to 1 and h closer to 0, will make the algorithm expand more nodes improving the length of the path.
- Setting to g to values close to 0 and h closer to 1, will have the opposite effect, expanding less nodes and going straight to the target using only the heuristic to move.

Concept of g and h taken from [9].

A* Data structures

A* utilises two data structure to be efficient a Priority queue for $open$ and a set for $closed$ 2 $Open$ is a **Priority queue** containing all nodes which need to be expanded. In the version of C# that uses unity by default this data structure is absent, so I added the package **C5** [7] through **Nuget** [3] a package manager for Unity. To implement the Priority queue from C5 I used an **Interval-heap** which will provide the best result in terms of complexity for a priority queue, page 239 of [7] complexities. $Closed$ is a set, in the closed set are stored the nodes which all neighbours have been explored, it also ensures that A* expands every node only once [1].

A * algorithm

Algorithm 2 A*(source, target, g, h)

```
1:  global variables
2:      open, closed
3:      g, h
4:      source, target
5:  end global variables
6:  if target ∈ closed or target ∈ open then
7:      return true
8:  end if
9:  source.parent ← source
10: source.gScore ← 0
11: source.fscore ← F(source)
12: open.add(source.fscore,source)
13: while open ≠ ∅ do
14:     current ← open.Pop()
15:     SetVertex(current)
16:     if current = target then
17:         return true
18:     end if
19:     closed.add(current)
20:     for each neighbour ∈ current.neighbours do
21:         if neighbour ∉ closed then
22:             if neighbour ∉ open then
23:                 neighbour.gScore ← ∞
24:                 neighbour.fScore ← ∞
25:                 neighbour.parent ← null
26:             end if
27:             UpdateNodePriorityA(current, neighbour)
28:         end if
29:     end for
30: end while
31: return false
```

Algorithm 3 UpdateNodePriorityA(current, neighbour)

1: $neighbour_{\text{gscore}} \leftarrow current.gScore + G(current, neighbour)$
2: **if** $neighbour_{\text{gscore}} < neighbour.gScore$ **then**
3: $neighbour.gScore \leftarrow neighbour_{\text{gscore}}$
4: $neighbour.fScore \leftarrow F(neighbour)$
5: $neighbour.parent \leftarrow current$
6: **if** $neighbour \in open$ **then**
7: $open.replace(neighbour.fscore, neighbour)$
8: **else**
9: $open.add(neighbour.fscore, neighbour)$
10: **end if**
11: **end if**

Algorithm 4 Clear

$open \leftarrow \emptyset$
$closed \leftarrow \emptyset$

Algorithm 5 F(node)

return $node.gScore + H(n)$

Algorithm 6 H(node)

return $h * EuclideanDistance(node.position, target.position)$

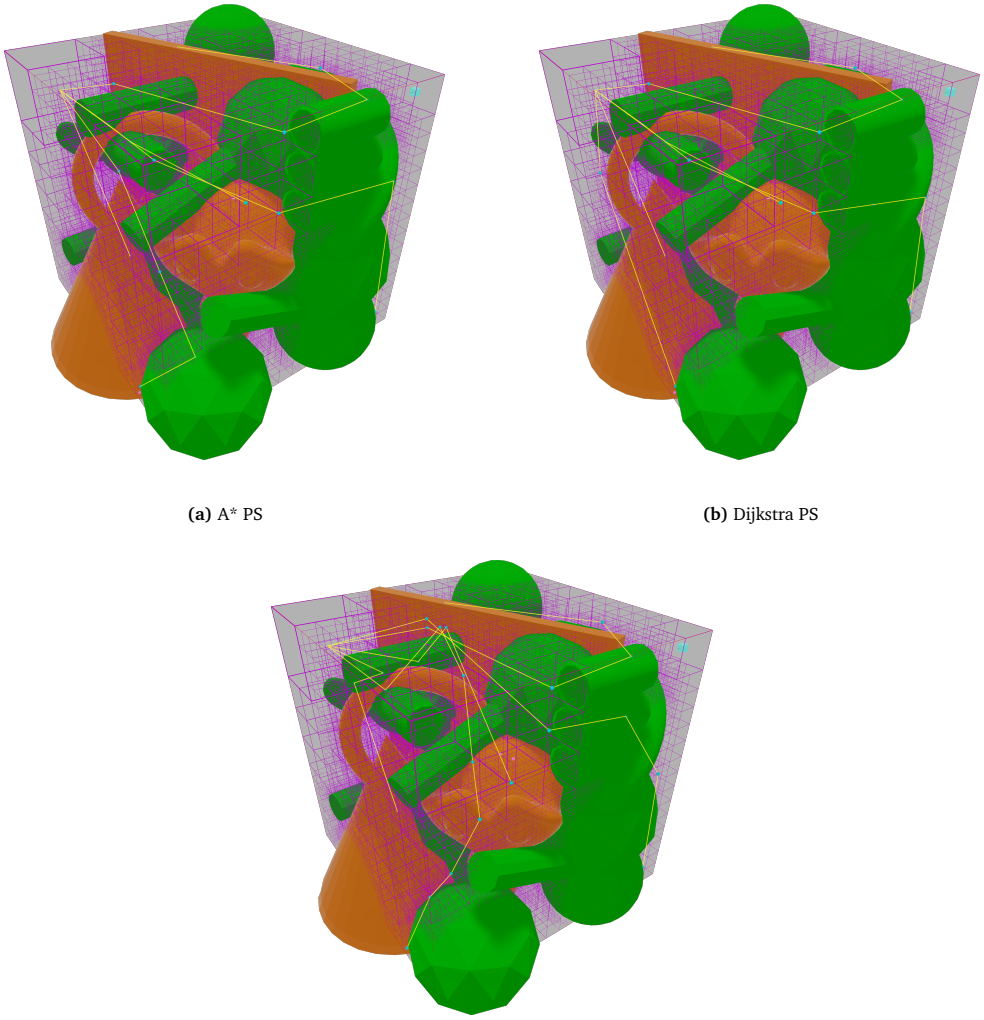
Algorithm 7 G($node_1$,$node_2$)

return $g * EuclideanDistance(node_1.position, node_2.position)$

5.1.5 A*, Best-first-search and Dijkstra paths limitations

A*, Greedy Best-first-search, and Dijkstra are edge constrained algorithms, thus paths are constructed by graph edges. The octree dual graph is a grid like because of how is generated, so edge constrained path resembles the graph. The resulting path will be very robotic looking with a lot of rigid turns as is shown in figures 13. To improve on the look and the path lenght we need to compute paths that are not composed exclusively on edges of the dual graph thus **Any-angle-paths**, refer to section 5.

5.2 Post-smoothing



(a) A* PS

(b) Dijkstra PS

(c) Greedy Best-first-search PS

Figure 14. Edge constrained path planning algorithms with **post smoothing**, 1 source, 5 targets

Post-smoothing is an algorithm that post process an already compute path enhancing it, it can be applied on the edge constrained paths (refer to section 5) It will straighten the path computed reducing at maximum the number of turns making the path look more realistic (as you can see in figures 14), at the point that is usable in an application, and also decreases considerably the path length. Paths are composed of nodes. Post-smoothing exploits **triangle inequality** to straighten those paths. Post Smoothing goes through the nodes in the path removing each of the node in between two nodes on the path which have line-of-sight, the new path is ensured to be smaller or equal to the previous thanks to **triangle inequality** [4], [1]. **Triangle inequality law**: states that the "sum the lengths of any two edges of a triangle is larger or equal than the length of the remaining edge" [1], [4]. Path smoothed by post smoothing becomes **any-angle paths**, refer to 5.

5.2.1 Post-smoothing implementation

Line-of-sight

There is a line-of-sight between two nodes in the space, if a line in which the ends points are the nodes position doesn't intersect any obstacles. To check for the line-of-sight I used Unity built-in function $Physic.SphereCast$ [13], which starts a ray from a first node position to and second node position, in the case the ray doesn't collide with an obstacle we have line-of-sight between the two nodes. Using a single ray is how is described in source [1], but in real scenario we need to also consider the volume of the agent, because the source is not a single point with no volume

in the space, but it is rather a 3D mesh, so we must enclose the mesh with a simplified collider a sphere to check for collision, thus the ray become a sphere.

Post-smoothing algorithm

Algorithm 8 PostSmoothing($p_0, p_1, p_2 \dots p_n$, agent)

```
1: smoothPath ← ∅
2: smoothPath.add(p_0))
3: for i = 1, i < n - 1, i++ do
4:     if NOT agent.LineOfSight(p_i+1, smoothPath.last()) then
5:         smoothPath.add(p_i))
6:     end if
7: end for
8: smoothPath.add(p_n))
```

Line 4: smoothPath.last() returns the last element inserted in $smoothPath$ list. Path $p_0, p_1, p_2 \dots p_n$ is the path compute by A* or variants, p_0 start node and p_n the goal node will be always in the path. Lines 3-6: we iterate in each of the node in the path by increasing the index i until we have **line-of-sight** between node p_{i+1} and the last added node in $smoothPath$, In case we don't have line of sight we can add to the smoothed path the last node p_i which we know has line of sight since compute in the previous iteration of the for loop; basically we will effectively remove all nodes between the last added node to $smoothPath$ and p_i. Then we repeat this process with p_i now being the last element in $smoothPath$ as the start to the line-of-sight check.

5.3 Any Angle Path planning

In any angle path, planning is no longer constrained to only graph edges, but they still propagate information only to graph nodes [1] Any angle path planning algorithms will improve the path length compared to edge constrained path planning algorithms and post smoothing.

5.3.1 Basic Theta*

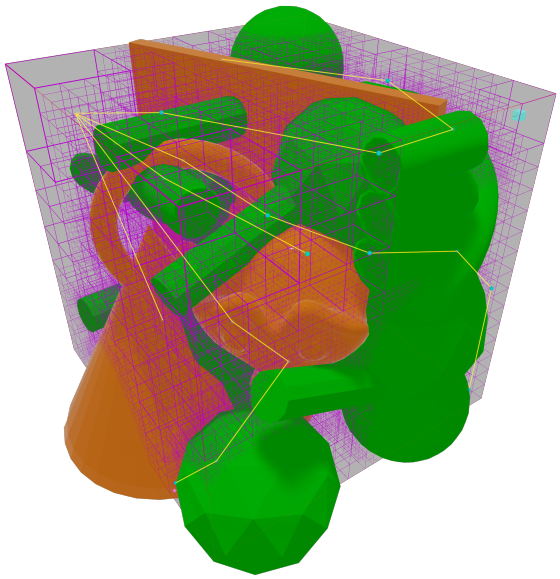


Figure 15. Basic Theta *, 1 source, 5 targets

Basic Theta* basically "calculates post-smoothing" 5.2 while computing the path, every node expansion it checks for two paths, the **smoothed path** and the regular **edge constrained path**(thus following the edge of the dual graph). It calculates immediately the smoothed path compared to A*, so it tends to expand less nodes, since straighten paths

tends to be smaller. Consequently, the $fScore$ 5.1.4, will be breaking ties with similar $fScore$ values common in grid graphs [1]. We should keep in mind the cost of line-of-sight checks that will be computed for a multitude of nodes, not only the nodes in the path as in A* PS.

Basic Theta* implementation

The implementation remains the same as in algorithm A* 2, the only line of code that changes in line 27 in $UpdateNodePriorityTheta$ 9. Let's consider node y, x the parent of node y, z one neighbour node of x, Basic theta* considers two possible paths every node expansion:

- **Path 1 (smoothed path)**: allows for any-angle paths, by checking whether there is **line-of-sight** between the parent x and the current node's neighbour z. If there is line-of-sight **path 1** is guaranteed to smaller or equal in length than**path 2** thanks to the triangle inequality [1], but we need to check anyway if the length of the between x and z is lower than $x \rightarrow y \rightarrow z$ since the nodes could all lie in a line and updating the priority in the queue is expensive. If these conditions are met it will update the path effectively removing node y setting z's parent x. Therefore, Basic Theta* allows any nodes in line-of-sight to be parent of a node [6], and not only the **direct neighbours** (neighbours defined by the dual graph).
- **Path 2 (edge constrained path)**: In the case that there is not line-of-sight, it will simply update the nodes as in A* using the edge of dual graph.

Basic Theta* algorithm

Algorithm 9 UpdateNodePriorityTheta(current, neighbour)

```
if agent.LineOfSight(current.parent, neighbour) then
    neighbour_gscore ← current.parent.gscore + G(current.parent, neighbour)
    if neighbour_gscore < neighbour.gScore then
        neighbour.gScore ← neighbour_gscore
        neighbour.fScore ← F(neighbour)
        neighbour.parent ← current.parent
        if neighbour ∈ open then
            open.replace(neighbour.fScore, neighbour)
        else
            open.add(neighbour.fScore, neighbour)
        end if
    end if
else
    UpdateNodePriorityA(current, neighbour)
end if
```

5.3.2 Lazy Theta*

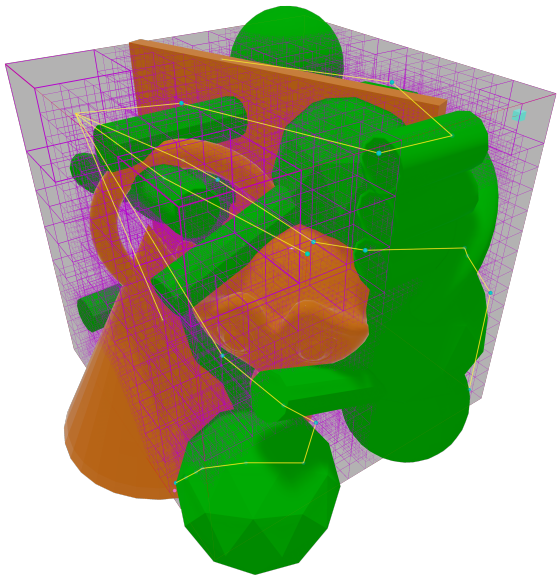


Figure 16. Lazy Theta*, 1 source, 5 targets

Lines of sight checks are expensive, so reducing them as much as possible is important especially in a 3D environment [6], in fact Basic Theta* was develop for 2D environment, instead Lazy theta* is optimized for the 3D, reducing greatly the number of line-of-sight checks.

Lazy Theta* implementation

Refer to algorithm A* 2 the only difference are that function $SetVertex$ 10 is implemented, and line 27 changes in $UpdateNodePriorityLazyTheta$ 11. Lazy theta * uses lazy evaluation [6] in **path 2**: while expanding the node in algorithm 11. Instead of computing the line-of-sight (refer to section 5.2.1) between each of the unexpanded neighbours z nodes and parent node x as in basic theta*, lazy theta assumes optimistically that there is line-of-sight between x and z and postpone the check only when node z is retrieved by the priority queue. Algorithm $SetVertex$ 10 checks if the assumption was correct $SetVertex$, if it is, it will not change **path 2**, on the contrary will revert to **path 1** [6]. **Path 1**: $SetVertex$ revert to path 1: let's consider we retrieve back z, the parent is x and there is no line-of-sight between them. We take the best direct neighbour (using edge of the dual graph) of x which is already in the closed set so y, and we set y's parent z and reset also the $gScore$. With this simple assumption we reduce considerably the line-of-sight checks, since we don't do line-of-sight checks for each of node that will be in $open$ but only for the one that will be in $closed$ [6]

Lazy Theta* algorithm

Algorithm 10 SetVertex(current)

```
1: if NOT agent.LineOfSight(current.parent, current) then
2:     min_node ← null
3:     minimum ← ∞
4:     for each n ∈ current.neighbours do
5:         if n ∈ closed then
6:             score ← n.gScore + G(n, current)
7:             if score < minimum then
8:                 minimum ← score
9:                 min_node ← n
10:            end if
11:        end if
12:    end for
13:    current.parent ← min_node
14:    current.gScore ← minimum
15: end if
```

Algorithm 11 UpdateNodePriorityLazyTheta(current, neighbour)

```
1: neighbour_gscore ← current.parent.gscore + G(current.parent, neighbour)
2: if neighbour_gscore < neighbour.gScore then
3:     neighbour.gScore ← neighbour_gscore
4:     neighbour.fScore ← F(neighbour)
5:     neighbour.parent ← current.parent
6:     if neighbour ∈ open then
7:         open.replace(neighbour.fScore, neighbour)
8:     else
9:         open.add(neighbour.fScore, neighbour)
10:    end if
11: end if
```

5.4 References

Those concepts and algorithms A*, Post-smoothing, Basic Theta* were taken from source [1], in the this source is described for the 2D environments. The only difference is how we calculate the distances, in a 3D environment we use the Euclidean distance. In the paper from the same author [6] describes how to modify A* Post-smoothing, Basic Theta* for 3D environments, is the source also of concepts and algorithm for Lazy Theta*. Paper [4] will contain both paper and many other concepts.

6 Multi-target single-source path planning

In a game one common situation is that multiple sources or “enemies” will try to follow a target, commonly the "player". I made an optimization for this situation making path planning algorithms above mentioned work for **multi-source single-target**. The problem gets inverted to **multi-target single-source**, thus "enemies" becomes the targets and the source the "player". It works simply calling for example A* once for each target to find, without removing the nodes calculated in previous searches. The system should also support that multiple targets will be referring to the same dual graph node, remember path planning is only compute between nodes of the dual graph, an agent (definition 1) need to find its nearest node of the octree dual graph (refer to section 4).

6.1 Multi-target single-source implementation

A target needs to register to the source, since the path is calculated from the source to the target. The **source** will contain multiple registered **targets**, and it will sort them basing on the distance from itself, therefore the nearest target will be the first in the list. Afterwards it will execute one path-planning algorithm for each of the element of the list,

but it will not clear the previously inserted nodes in both $open$ priority queue and $closed$ set, refer to 5.1.4. Thus, for the second target when the searches will start some nodes in the dual graph will have been already expanded, so $open$ and $closed$ are not empty. Is important that each target is a different node of the dual graph, in the case of multiple targets on the same node, we need to make sure that the algorithm is executed only once for each different target.

There are two small modification that we need to do to A* to make it compatible (subsequently all algorithm deriving from A* will be compatible, meaning all algorithm defined above.)

- Lines 6 – 8 of 2 it checks right away if the node was already found by previous computation.
- Algorithm 2 does not right away **clear** the open or closed data structure every time a search start, but has Clear algorithm 4 called by the multisource implementation.

7 Comparison

7.1 How testing is performed

Testing is performed in different worlds; every world will have different characteristics defined by the shape of the obstacles and how they are placed. For example, 17 shows an open world, while figure 18 shows a maze so there is only one way to reach the destination. We will have only five **targets** during testing, and each target will be assigned to a random dual graph node until visiting every node once. The source will not change node and will be placed in a meaningful position. We can recompute many times this configuration, applying different random **seeds** such that the position combination of the five targets will change. This approximated testing configuration is necessary since it would take too much time to compute all possible combination for five targets. All values in the table are the averaged of all those re-computation:

- **Time**: is the average time of all execution in milliseconds, to find all 5 paths.
- **Length**: is the average length of the 5 paths to reach targets
- **Line-of-sight**: is the average number of **line-of-sight** 5.2.1 checks in order to compute the solution.
- **Closed**:is are the average number of nodes that the solver went through to find all targets (the $closed$ set size)
- **Tradeoff** is computed $\frac{Time}{\frac{1}{travelled-distance}} * 100$, so smaller values are better, it will represent the trade-off between time and path length.

The algorithms A*, Basic Theta*, Lazy Theta* in the test will have values $g = 1$ and $h = 1$, the standard configuration.

multi-target single-source testing

To properly test the optimization of multi-target single-source I computed two tables:

- Single-target Single-source: table that calculates the path singularly for each of the agents in the space and then average their results.
- Multi-targets Single source: which implements the optimization.

7.2 Tests multi-target vs single-target

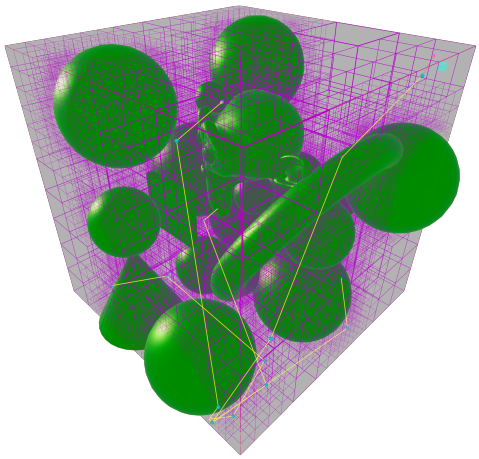


Figure 17. Word1 sparse objects, Nodes 21091, max-depth: 6 2

Table 1. World1 single-target single-source

Algorithm	Time	Length	Line-of-sight	Closed	Trade-off
Dijkstra	36.82	609.86	0.00	10 544.97	224.52
BestFirstSearch	0.35	924.53	0.00	133.92	3.24
A*	10.15	614.45	0.00	2495.91	62.38
Dijkstra PS	33.37	524.51	2.44	10 544.97	175.02
BestFirstSearch PS	0.09	565.32	3.49	133.92	0.52
A* PS	7.10	522.06	2.49	2495.91	37.07
Basic Theta*	5.68	354.27	2949.23	906.78	20.11
Lazy Theta*	3.76	360.93	124.24	947.79	13.56

Table 2. World1 multi-target single-source

Algorithm	Time	Length	Line-of-sight	Closed	Trade-off
Dijkstra	11.07	609.86	0.00	3506.62	67.52
BestFirstSearch	1.00	1293.53	0.00	223.65	12.92
A*	5.96	649.74	0.00	1493.33	38.75
Dijkstra PS	10.58	524.52	2.44	3506.62	55.49
BestFirstSearch PS	0.84	730.53	4.71	223.65	6.15
A* PS	5.10	538.42	2.64	1493.33	27.47
Basic Theta*	5.39	394.87	2404.23	738.36	21.30
Lazy Theta*	3.92	399.43	92.44	766.01	15.66

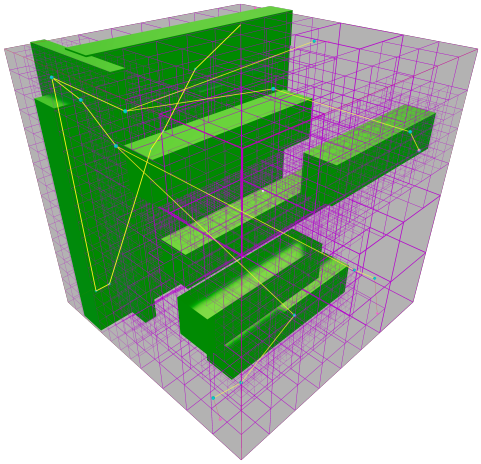


Figure 18. Word2 large blocking wall, Nodes 8701, max-depth: 6 2

Table 3. World2 single-target single source

Algorithm	Time	Length	Line-of-sight	Closed	Trade-off
Dijkstra	9.37	609.91	0.00	3503.89	57.13
BestFirstSearch	0.66	1283.53	0.00	222.13	8.51
A*	3.95	649.56	0.00	1499.74	25.68
Dijkstra PS	9.24	524.58	2.44	3503.89	48.46
BestFirstSearch PS	0.74	725.13	4.59	222.13	5.37
A* PS	3.84	538.21	2.63	1499.74	20.66
Basic Theta*	5.16	395.58	2427.87	743.20	20.40
Lazy Theta*	3.17	400.04	93.64	769.95	12.70

Table 4. World2 multi-target single-source

Algorithm	Time	Length	Line-of-sight	Closed	Trade-off
Dijkstra	1.88	1695.63	0.00	905.20	31.91
BestFirstSearch	0.30	2022.87	0.00	215.03	6.14
A*	0.99	1689.00	0.00	481.79	16.78
Dijkstra PS	2.05	1523.26	7.35	905.20	31.20
BestFirstSearch PS	0.38	1685.50	6.85	215.03	6.43
A* PS	1.04	1515.84	7.35	481.79	15.82
Basic Theta*	1.17	1331.34	988.08	344.22	15.51
Lazy Theta*	0.84	1330.46	52.40	356.08	11.15

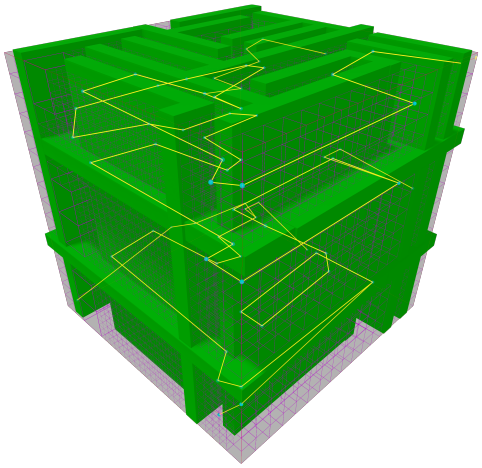


Figure 19. Word3 maze, Nodes 7271, max-depth: 5 2

Table 5. World3 single-target single-source

Algorithm	Time	Length	Line-of-sight	Closed	Trade-off
Dijkstra	5.20	4353.08	0.00	3634.50	226.15
BestFirstSearch	4.70	4670.05	0.00	2763.21	219.50
A*	5.46	4356.07	0.00	3487.35	237.85
Dijkstra PS	5.71	3987.99	26.47	3634.50	227.90
BestFirstSearch PS	4.77	4019.73	26.86	2763.21	191.84
A* PS	5.66	3977.41	26.47	3487.35	225.06
Basic Theta*	9.99	3810.55	6914.40	3456.55	380.75
Lazy Theta*	7.96	3811.35	318.73	3456.74	303.35

Table 6. World3 Multi-target single-source

Algorithm	Time	Length	Line-of-sight	Closed	Trade-off
Dijkstra	27.27	784.14	0.00	10 831.47	213.81
BestFirstSearch	1.03	4717.73	0.00	584.39	48.79
A*	1.18	4355.85	0.00	735.33	51.39
Dijkstra PS	1.28	3984.52	26.45	758.44	50.99
BestFirstSearch PS	1.20	4028.42	26.89	584.39	48.46
A* PS	1.38	3974.36	26.45	735.33	54.76
Basic Theta*	2.10	3810.08	1460.71	729.88	79.96
Lazy Theta*	1.72	3808.17	67.07	730.00	65.51

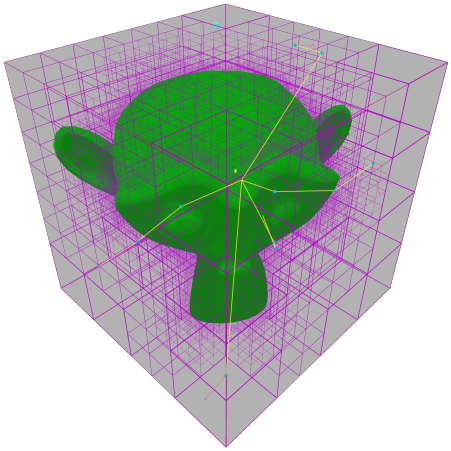


Figure 20. Word 4 Single large concave mesh shape, Nodes 9054, max-depth: 6 2

Table 7. World4 single-target single-source

Algorithm	Time	Length	Line-of-sight	Closed	Trade-off
Dijkstra	11.31	470.98	0.00	4525.71	53.28
BestFirstSearch	0.03	794.33	0.00	82.02	0.22
A*	3.11	472.72	0.00	1481.45	14.71
Dijkstra PS	11.15	424.61	1.14	4525.71	47.34
BestFirstSearch PS	0.04	490.60	1.70	82.02	0.21
A* PS	3.19	422.06	1.13	1481.45	13.45
Basic Theta*	4.89	276.84	2650.88	786.85	13.55
Lazy Theta*	2.87	281.70	54.58	832.03	8.09

Table 8. World4 Multi-target single source

Algorithm	Time	Length	Line-of-sight	Closed	Trade-off
Dijkstra	2.32	471.51	0.00	958.64	10.94
BestFirstSearch	0.02	813.88	0.00	23.79	0.15
A*	0.76	473.80	0.00	348.21	3.59
Dijkstra PS	2.36	424.61	1.13	958.64	10.03
BestFirstSearch PS	0.06	495.27	1.73	23.78	0.29
A* PS	0.81	422.55	1.13	348.21	3.42
Basic Theta*	1.35	279.35	689.43	208.53	3.76
Lazy Theta*	0.81	272.26	15.86	220.44	2.20

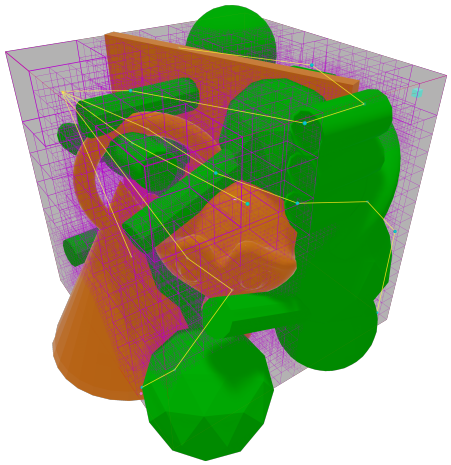


Figure 21. Word 5 Concave and convex mesh shape, Nodes 19797, max-depth: 6 2

Table 9. World5 single-target single-source

Algorithm	Time	Length	Line-of-sight	Closed	Trade-off
Dijkstra	23.54	820.46	0.00	9898.07	193.11
BestFirstSearch	7.74	1556.07	0.00	2731.87	120.47
A*	13.26	824.95	0.00	5688.43	109.42
Dijkstra PS	25.49	721.02	3.29	9898.07	183.78
BestFirstSearch PS	6.95	951.64	6.57	2731.87	66.10
A* PS	14.13	718.35	3.38	5688.43	101.52
Basic Theta*	43.68	556.77	12 823.50	4677.16	243.20
Lazy Theta*	26.83	564.05	510.16	4726.19	151.32

Table 10. World5 multi-target single-source

Algorithm	Time	Length	Line-of-sight	Closed	Trade-off
Dijkstra	8.01	820.47	0.00	3296.61	65.70
BestFirstSearch	3.97	1908.55	0.00	1571.16	75.77
A*	6.40	839.96	0.00	2492.96	53.76
Dijkstra PS	9.04	721.10	3.29	3296.61	65.22
BestFirstSearch PS	4.04	1113.65	7.83	1571.16	45.03
A* PS	6.17	725.79	3.42	2492.96	44.79
Basic Theta*	21.77	570.68	6074.38	2237.30	124.23
Lazy Theta*	13.09	576.02	243.36	2251.08	75.42

7.3 Octree VS 3D grid

Table 11. World1 single-target single-source **3D grid**

Algorithm	Time	Length	Line-of-sight	Closed	Trade-off
Dijkstra	27.64	751.49	0.00	10396.82	207.74
BestFirstSearch	0.01	817.70	0.00	90.12	0.10
A*	12.40	751.49	0.00	4306.43	93.22
Dijkstra PS	25.60	546.33	1.62	10396.82	139.85
BestFirstSearch PS	0.02	582.26	2.29	90.12	0.11
A* PS	9.97	535.68	1.60	4306.43	53.39
Basic Theta*	2.70	333.11	1860.69	621.09	8.98
Lazy Theta*	1.86	333.78	62.78	658.01	6.22

Table 12. World1 single-target single-source **Octree**

Algorithm	Time	Length	Line-of-sight	Closed	Trade-off
Dijkstra	6.03	620.26	0.00	2693.02	37.38
BestFirstSearch	0.02	848.73	0.00	70.14	0.20
A*	2.42	622.69	0.00	816.32	15.08
Dijkstra PS	7.87	516.74	1.76	2693.02	40.68
BestFirstSearch PS	0.03	554.98	2.30	70.14	0.16
A* PS	1.90	514.27	1.79	816.32	9.78
Basic Theta*	2.50	337.90	1066.08	340.21	8.46
Lazy Theta*	0.96	349.33	46.68	351.59	3.36

7.4 Environment for testing

Unity version: 2021.3.4f1 Version C#: .NET 2.1.202
Surface book 2:

- **CPU**: Intel(R) Core(TM) i7-8650U CPU @ 1.90GHz
- **RAM**: 16GB RAM 1866Mhz LPDDR3
- **GPU**: NVIDIA® GeForce® GTX 1060 discrete GPU w/6GB GDDR5 graphics memory

7.5 Algorithm results

7.5.1 General results

As we can see in the results, the best trade-off in all situations is **Greedy Best-First search**. Even though the greedy approach is very fast, it is unemployable if we don't want agents to move in a robotic way. Consequently, the best option appears to be **Best-first-search PS**: although the average path length can increase quite a bit compared to **any angle path planning algorithms** or **A * PS**, but the very fast computation time makes it worth it. **Greedy Best-first-search** with post-smoothing has the largest improvement in travelled distance compared to **Dijkstra PS** and **A* PS**. As **Greedy Best-first-search** paths are very unrealistic, since it contains much more artifacts compared to the latter (for example moving up and down or in spiral), smoothing will also remove this problem by improving the path length.
In most cases, **edge constrained path planning algorithms** with post-smoothing result to have a better trade-off than **any angle path planning**, while maintaining realistic looking path. For this reason, **post-smoothing** is widely used in the videogame industry [4]. In my testing, this is particularly true for **Greedy Best-first-search PS**, while **Dijkstra PS** still is not comparable. A* PS on the other hand seems to have a worst trade-off than **any angle path planning algorithm**, in most cases is outperformed both in travelled distance and time. This is established also by [4] on grids. The testing confirms that any-angle algorithms in every world provide the best travelled distance, although Lazy Theta* due to the line-of-sight reduction will always have better computational time than **Basic theta*** while maintaining a very similar travelled distance. As established by source [4], "Lazy Theta* provides a nearly dominating trade-off over Basic Theta*", although you should keep in mind that source refers to findings on 3D grid.

A developer can interleave on different path planning algorithms based on distance from the source, for example he can use **Greedy best-first search** on a very distant target and **Lazy theta*** on a near target making them react more promptly.

7.5.2 Multi-target single-source results

Multi target optimization did work effectively in every world tested, for most of the algorithms in a large decrease in time without a large increase in path length. This is also very noticeable by the trade-off column in the tables. Sometimes the multiple-target optimization gives slightly larger time in computation; this is the case with **Greedy best-first-search** and **Greedy best-first-search PS**, and it could be happening because the **Greedy Best first search** is quite fast, and multisource requires sorting the targets, which is an additional overhead.

7.5.3 Octree VS 3D grid results

Test performed as the one above but with only one **source** and one **target**, refer to 1. As we can see from the result, using an octree permits to have in most of the cases a better trade-off. This is mainly thanks to the lower calculation, which in turn is due to the reduction of the nodes in the octree data structure. The trade-off does not work for of Best-first-search, but it could be test time approximation errors. If we consider the shortest path, there is no clear winner since the values are very similar. This makes the octree even more worth it, as it permits to have a lower computation time with similar distances results. These results are not comparable to the one in paper [6] because there has been tested a run on cubic graph but each node has 26 neighbours, while my grid implementation has 6 neighbours instead. Anyway, the results should give an insight of the performance gains given by using an octree.

8 Agents

8.1 Agent movement

This project does not focus on the agent 1 movement. Since movement and path planning are usually kept separate as subjects [4], agents are provided with simple movements:

Player (source) FPS movement

I implemented a simple first-person flying movement so that the player can move in the 3D space. Essentially, it get the mouse position on screen converts into a rotation and apply to the player object, while the movement keys **W,A,S,D** move the object forwards, left right or backwards based on the direction decided by the player.

Enemies (targets) movement

The target to reach the source will follow one after the other the node's position defined in the path, simple movement towards points in the path is based on $speed * time$ calculating its next position each frame. The speed of targets is randomized to make for more interesting situations in path planning, thus having different speed targets need a form collision avoidance. Note that a target doesn't need to reach a position perfectly has a range in which the position is acceptable and goes to the next.

8.2 Collision avoidance between targets

Actually there is no collision avoidance system in the project, a target is able to surpass other slower targets simply by pushing it away, only a target with larger speed can push another, a target with lower speed cannot push a target with larger speed, the push is calculated by taking the hit position between two targets (with a Unity build-in function $OnCharacterControllerHit$), define the direction of push by subtracting the position of the target with larger speed and the hit position (also hit position gets a bit randomized), apply the push direction as force to the slower target. Because of pushing is possible that a target gets stuck, there is re-planner algorithm that gets executed every 30 seconds and recompute the path for stuck targets.

9 Conclusion

In this project I have managed to create a working system for navigation in 3D in unity by effectively discretize the 3D space with an octree with both **convex** and **concave shape meshes** 2.3.2 . I also defined a navigable dual graph for

path planning. Programmed a multiple path planning algorithm with different characteristics exploitable in different worlds or situations, adaptable in terms of run-time, and with a realistic looking with **post smoothing** and **any-angle algorithms**. I effectively optimized the computation for multiple-target single-source, which is a recurring problem in videogames. Finally, I tested the octree representation against 3D cubic grid in which the described algorithm was designed for, with positive outcome due to octree reduction of nodes

References

[1] K. Daniel, A. Nash, S. Koenig, and A. Felner. Theta: Any-angle path planning on grids. *The Journal of artificial intelligence research*, 39:533–579, 2010.

[2] J. J. Hasbestan and I. Senocak. Binarized-octree generation for cartesian adaptive mesh refinement around immersed geometries. *Journal of Computational Physics*, 368:179–195, 2018.

[3] P. McCarthy. Nuget package meanger for unity, 2018. License: `https://github.com/GlitchEnzo/NuGetForUnity/blob/master/LICENSE`.

[4] A. Nash. *Any-Angle Path Planning PhD thesis*. PhD thesis, Department of Computer Science, University of Southern California Los Angeles (California), 2012.

[5] A. Nash and S. Koenig. Any-angle path planning. *AI Magazine*, 34(4):85–107, Winter 2013. Copyright - Copyright Association for the Advancement of Artificial Intelligence Winter 2013; Document feature - Tables; ; Graphs; Equations; Diagrams; Last updated - 2021-09-09.

[6] A. Nash, S. Koenig, and C. Tovey. Lazy theta*: Any-angle path planning and path length analysis in 3d. volume 1, 01 2010.

[7] I. U. o. C. Niels Kokholm, Peter Sestoft. C5 package for c#, 2006. License: `https://www.itu.dk/research/c5/LICENSE.txt`.

[8] Nvidia. Convex mesh, 2021. [Online; accessed 20-june-2022].

[9] A. Patel. Heuristics from amit's thoughts on pathfinding.

[10] Shaders Laboratory. Wireframe shader. License: `http://www.shaderslab.com/index.html`.

[11] U. Technologies. Unity mesh combinemeshes [unity manual], 2021.

[12] U. Technologies. Unity physics overlapbox [unity manual], 2021.

[13] U. Technologies. Unity physics spherecast [unity manual], 2021.

[14] U. Technologies. Unity, 2022. License: `https://store.unity.com/compare-plans`.

[15] Unity. Unity nav-mesh.

[16] Wikipedia. The Free Encyclopedia. Octant — Wikipedia, the free encyclopedia, 2021. [Online; accessed 20-june-2022].

[17] Wikipedia. The Free Encyclopedia. A* search algorithm — Wikipedia, the free encyclopedia, 2022. [Online; accessed 20-june-2022].

[18] Wikipedia. The Free Encyclopedia. Best-first search — Wikipedia, the free encyclopedia, 2022. [Online; accessed 20-june-2022].

[19] Wikipedia. The Free Encyclopedia. Euclidean-graph — Wikipedia, the free encyclopedia, 2022. [Online; accessed 19-june-2022].

[20] Wikipedia. The Free Encyclopedia. Octree — Wikipedia, the free encyclopedia, 2022. [Online; accessed 19-june-2022].