Efficient Octree-based 3D Pathfinding

Quentin Massonnat, Clark Verbrugge
McGill University
Montreal, Canada
quentin.massonnat@mail.mcgill.ca, clump@cs.mcgill.ca

***Abstract*—Even though many games feature complex 3D environments, 3D pathfinding remains a challenging problem. Representing large 3D maps can require a lot of memory, and pathfinding instances must be solved very quickly while the game is running. In this work we develop an efficient solution to 3D pathfinding by building a reduced, hierarchical grid representation within which we can extend traditional 2D navigation mesh (navmesh) pathing. Starting from an octree representation, we merge adjacent cells while preserving their convexity to obtain a coarser representation that greatly reduces path computation costs. We then build a navigation graph from this octree within which we can search for paths using the popular A* search algorithm. To increase the quality of the paths we obtain we implemented two forms of path refinement: a visibility-based path pruning heuristic, and a 3D extension of the classic "funnel" algorithm that computes minimal homotopic paths. We further extend our work to handle dynamic environments with local and efficient updates to the octree and the movement graph. Experiments on a variety of scenarios show that our approach remains fast and efficient even for very large 3D maps and could be used for real-time pathfinding in video games. A detailed comparison with the state-of-the-art JPS-3D algorithm shows that our approach produces shorter path lengths while being faster on long path instances. We implemented our work in Unity, one of the most popular game engines, as an effort to make pathfinding in 3D environments accessible to game developers.**

***Index Terms*—3D pathfinding, octree, navigation mesh, dynamic pathfinding**

I. Introduction

3D virtual environments are commonplace in modern games. Non-Player Character (NPC) movement planning, however, is still largely 2D, with off-plane or vertical movement modeled through limited, and frequently custom connections between 2D surfaces (e.g., ladders, jumping, and climbing points), or greatly simplified by a relative absence of obstacles, such as in traditional flight or space-simulations or for aerial attackers that generally move well above most obstacles. Even games that emphasize flying or floating enemies, such as *Marvel's Spider-Man*'s jetpack-enabled opponents, or the various flying monsters in *Elex 2* target primarily through line-of-sight and do not tend to take advantage of the many opportunities for more complex 3D pathing afforded by the intricate environments in which combat takes place.

Pathfinding in 3D, however, is expensive. The extra dimension of traditional grid or voxelized models greatly increases memory and time costs, while techniques for building and exploiting non-grid-like representations are also complex. On top of that, even though some tasks can be pre-computed, the active task of pathfinding must be done in real-time while the game is running, with a fraction of the computing resources available, and therefore it must be fast and cheap.

This work proposes a hybrid approach to 3D pathfinding, building a hierarchical representation of the environment to which we can extend traditional 2D and graph pathing using navigation meshes. We start from an octree representation, which is significantly more efficient than a voxel grid representation. From this representation, we create a movement graph (roadmap) to find short paths between two points. We improve this straightforward method in several ways: we use a Hertel-Mehlhorn approach [12] to merge adjacent cells and obtain a coarser representation that greatly reduces path computation costs, and implement two path refinement methods to reduce path length. The first one is a visibility-based heuristic inspired by Yang and Sukkarieh [24]. The second and more involved technique, based on the work of Massonnat [14], is an extension of the 2D funnel algorithm [11] that builds a shorter, 3D homotopic path.

We also extend our pathfinding to dynamic 3D environments containing moving obstacles. With some modifications, it is possible to efficiently update the octree and the associated roadmap at runtime. This creates new game design possibilities by allowing movement planning in more diverse scenarios.

We implemented our approach in Unity3D and created a custom set of 3D benchmarks to evaluate it, meant to be representative of topologically and navigationally complex environments in which 3D pathing may be useful. We also test our work on the more challenging dataset based on the game *Warframe* [4], which contains very large maps and pathing problems for each map that can be used to compare algorithms.

The main contributions of this work are as follows:

1) We implement the traditional octree data structure in Unity[1], and vastly improve it with the Hertel-Mehlhorn style merging and two forms of path refinement.
2) We extend this work to dynamic 3D environments with moving obstacles.
3) Our approach is evaluated on a set of (non-voxelized) 3D custom scenes inspired by modern game environments as well as more complex 3D benchmarks. We provide a thorough comparison of our approach and what is to our knowledge the state-of-the-art 3D pathfinding method, 3D Jump Point Search (JPS-3D) [18]. Our approach

[1]Our code is available at https://github.com/Qmassonnat/octree_decomp

provides more flexibility, and shorter path lengths, while maintaining feasible compute times.

II. Background and Related Work

Pathfinding in video games is a rich research topic that combines many different topics. Below we discuss specific approaches to 2D and 3D, the use of navigation meshes, path refinement, the use of octrees, and 3D pathfinding benchmarks.

1) Pathfinding in 2D: Most of pathfinding research focuses on 2D environments. Abd Algfoor, Sunar, and Kolivand [1] did a comprehensive study of pathfinding in games, mostly for 2D environments. Cui and Shi [6] review A* approaches for 2D pathfinding, underlining the fact that the quality and efficiency of the underlying data representation is essential. Sturtevant [22] creates a sparse grid representation of the environment in 2D and 3D games, although the 3D environments consist only of 2D walking surfaces connected in a 3D environment and therefore do not allow full 3D movement.

Botea, Müller, and Schaeffer [2] propose an improvement to the A* algorithm: HPA* (Hierarchical Pathfinding A*) that abstracts an environment to locally linked clusters. At a local scale, pathfinding inside a cluster can be done with traditional A* and at a global scale, it can be done by navigating from cluster to cluster along pre-computed routes. Hierarchical methods allow to store and reuse some intermediate results, making pathfinding faster to compute.

2) Pathfinding in 3D: Due to a higher branching factor and higher volumes, 3D pathfinding is a more difficult problem that can only be solved approximately. Canny and Reif [5] prove that finding the exact shortest path between 2 points in a 3D environment with polyhedral obstacles is NP-hard. All works for pathfinding in 3D then have to incorporate heuristics and show a trade-off between path quality and computational cost.

Many improved versions of the A* algorithm exist for 3D. Sislák, Volf, and Pechoucek [21] explore flight trajectory path planning, but do not incorporate a reduced representation of the environment such as navigation meshes. Frontera et al. [8] propose an algorithm with good empirical running time and solution quality, but their work is restricted to environments where obstacles are protruding vertical polyhedra (for example buildings in urban environments).

Li et al. [13] gives an approach to find safe paths for drones inside cluttered buildings, but the proposed approach uses voxels instead of a more efficient data structure and therefore suffers from high computational cost. Octrees are a more efficient hierarchical data structure, created from a large cube recursively split in 8 smaller cubes while it contains an obstacle. The work most similar to our approach is by Muratov and Zagarskikh [17], who use octrees and a clustering method similar to HPA*. Instead of creating clusters of a fixed size in a voxel grid, they use the depth level in the octree to create clusters. Each cluster has transitions to adjacent clusters. By precomputing all paths within any given cluster, they perform a hierarchical search similar to HPA*. Our proposed approach goes further by reducing the number of cells in the octree, leading to faster compute times, and incorporating non-trivial path refinement to increase the path quality and reduce their length. Their clustered octree search suffers significantly in terms of path length due to the clustering process, with paths on average 20% longer than optimal solutions.

Two works on 3D pathfinding stand out: sparse voxel octrees (SVOs) and 3D Jump-Point Search (JPS-3D). A highly optimized octree-based navigation is used by Brewer to perform real-time 3D pathfinding in the video game *Warframe* [3] using SVOs, based on the work of Schwarz and Seidel [20]. In SVOs, the data for each level of the tree is stored in a compact way using a *Morton Code order* [16], which allows for the enumeration of octree cells while keeping neighbor cells close to each other in memory. To capture fine details in the environment without building the octree all the way, they use small 4x4x4 voxel grids instead of regular cells as octree leaves when the minimum octree size is reached. Our work approaches octree navigation differently: these storage and implementation optimizations could be compatible with our work, and we propose a merging method to reduce the number of nodes in the octree, leading to faster compute times.

The JPS algorithm is a search algorithm like A* that maintains its optimality and is faster by an order of magnitude on 2D grids [9]. The same idea can be extended to 3D, even if it results in more switch cases as a cube has 26 potential neighbors and a square only has 8. The 3D extension of the JPS algorithm allowed Nobes et al. [18] to perform fast pathfinding in large 3D environments, and is a state-of-the-art method for 3D pathfinding. We will show a detailed comparison of this method and our work in Section IV.

3) Navigation meshes: The standard method to perform 2D pathfinding in a game environment is to use a form of *navigation mesh*, or navmesh. Any tiling, or partition of the navigable (interior and non-obstacle) environment that conforms to obstacles is a navigation mesh. Having convex tiles is especially useful in pathfinding applications since if a tile is convex and free of obstacles, two points in the same tile can always be connected by a straight line that does not intersect any obstacles.

Since moving between two points inside a free polygon can be done by going in a straight line, path-finding is reduced to moving from polygon to polygon, along a graph where each polygon is represented by a node, and neighbor polygons are connected by edges. Some tools that automatically create navmeshes, like Recast[2], are widely used in video game pathfinding.

4) Path refinement: After a (non-optimal) path is found, it can then be shortened or smoothed with various path refinement methods. To reduce a path's length, Yang and Sukkarieh [24] uses path pruning to skip a subset of points in the path, but the resulting path can be in a different homotopic class. Some works have studied the homotopic classes of paths: a simple characterization of these is that two paths are in the same homotopy class if they share the same start and end point and they can be continuously deformed to one another without

[2]https://github.com/recastnavigation/recastnavigation

intersecting any obstacles. There can be multiple classes due to the presence of obstacles. Hernandez, Carreras, and Ridao [10] uses homotopic variants of search algorithms like A* to find shortest homotopic paths in 2D scenarios. Hershberger and Snoeyink [11] introduces the *funnel algorithm* in 2D, used to find the shortest path in the same homotopic class. There are several implementations of the funnel algorithm, such as the Simple Stupid Funnel algorithm (SSFU) by Mononen [15] which inspired the extension of the funnel algorithm to 3D scenarios [14]. Erickson [7] uses path reduction and the funnel algorithm to compute the shortest homotopic path to a given path between two points in 2D environments.

5) 3D pathfinding benchmarks: Many pathfinding benchmarks exist for 2D environments, but to our knowledge, there are only three for 3D scenarios. Toll et al. [23] created a benchmark to compare 2D and 3D navigation meshes, but the 3D cases only correspond to 2D walking areas in a 3D environment, and thus only recreate a ”2.5D” scenario.

Brewer and Sturtevant recreated 3D maps from the video game *Warframe* [4], where jetpacks enable full 3D movement. These voxel-based maps mostly consist of ships, debris, and asteroids floating in empty space, with obstacles represented by a list of the coordinates of all the voxels occupied by obstacles. These maps can be extremely large, with some containing hundreds of thousands to millions of obstacle voxels enclosed in an overall volume of 10^9 cubic voxels. Due to their very large size, they allow testing pathfinding methods in difficult scenarios and it is challenging to do real-time pathfinding on them. For a more diverse set of maps, Nobes et al. [19] followed the same data structure to extend the benchmark with maps presenting a more diverse range of scenarios, motivated by some real-life use cases of 3D pathfinding.

III. Methodology

In this section we will describe how our navigation mesh is built, the pathfinding and path refinement methods we implemented, and how dynamic environments are handled.

A. Building the Navigation Mesh

The simplest 3D navigation mesh is a voxel decomposition where paths can only go through obstacle-free voxels, which we call *valid* voxels. Octrees are much more efficient since large open spaces can be represented with only one cell. We build the octree recursively, starting from a single cubic cell that encompasses the whole level. If an obstacle intersects the cell, we split it into 8 cubes half the size of the cell in each dimension that will be stored as the children of the main cell. We recursively continue this process of splitting invalid cells until the cells have reached a specified minimal size, which we define as *granularity*. We call *children* the set of cells that was obtained from splitting one *parent* cell. With this analogy in mind, we call *siblings* two cells that share the same parent. A valid cell without any obstacle or one that has reached the minimal size is called an octree leaf. Increasing this granularity gives a more accurate decomposition of the environment, but at the cost of more compute time, as in a worst-case scenario, halving the granularity can multiply the number of octree leaves by 8. The main advantage of the octree is that it requires many fewer nodes than a voxel-based approach, as a large obstacle-free space can be represented as a single octree leaf. By definition, a leaf is free of obstacles and as a convex space (cube) the shortest path between two points in a leaf is just a straight line.

As we will discuss in Section IV, the computing time of the pathfinding algorithm is directly linked to the number of valid octree leaves, so we are interested in lowering this number. On many occasions, adjacent cells can be merged while maintaining convexity. We implemented a greedy merging procedure inspired by the Hertel-Mehlhorn algorithm [12], which computes a convex partitioning within $4\times$ optimal by merging cells starting from a 2D polygon triangulation, as long as the merged area remains convex. In our case we check for adjacent cells that would remain convex after merging, which happens if the transition surface covers the entire side of both octree leaves. When two cells are merged, this can create new merging opportunities for neighboring cells, so we check again if the new cell can be merged with any of its neighbors. This heuristic we use for merging the octree is not guaranteed to be optimal (in terms of the total number of cells after merging), but in practice gives good results and still consistently leads to a significant decrease in the number of octree cells.

B. Path Finding and Path Refinement

All valid octree cells by definition are free of obstacles. Because they are convex cells, the shortest path between two points within the same cell is just a straight line. With this in mind, we just need to know how to navigate from cell to cell. We define as *transition surfaces* the surfaces connecting two adjacent valid cells. Using the environment decomposition, we create a graph, where nodes are in the middle of transition surfaces, and two nodes are connected if their associated transition are sides of the same octree cell. We connect them with an edge of weight $d(t_1, t_2)$ where d is the Euclidean distance and t_1 and t_2 are the middle of the two transitions. We then add the start and target points in the graph and temporarily connect them to all the transitions of the cell they belong to. The voxel decomposition can also be used as a baseline for pathfinding. It can be seen as a fully split octree with all cells of size 1, and the same pathfinding techniques apply.

With this graph, the task of finding the shortest path in a 3D environment is reduced to finding the shortest path on this weighted graph. We use the A* algorithm to perform this task due to its simplicity and flexibility in solving point-to-point shortest path problems [25]. We use the A* algorithm and the Euclidean distance as the associated heuristic for estimating the remaining distance between a node and the target.

Even though paths found with A* are optimal within the graph, heuristics such as the placement of transitions can make paths sub-optimal in the 3D environment itself. A first approach to reduce path length is a simple path pruning following a visibility-based heuristic. We say that two points

Algorithm 1 Pseudocode for the path pruning algorithm

Input: A path between two points s and t ($s = x_0, x_1, ..., x_{k-1}, x_k = t$)
Output: The pruned path

```
Let anchor = t and prunedPath = [t]
for $i$ decreasing from $k-1$ to 0 do
    if $x_i$ is visible from anchor then
        Discard the node $x_i$
    else
        Add the node $x_{i+1}$ to prunedPath and let it be the new anchor
    end if
end for
Add s to prunedPath
return prunedPath in reverse order
```

are visible from one another if there are no obstacles between them. Following the idea of Yang and Sukkarieh [24], the idea is to keep a subset of the points of the original path to create a shorter path that still does not intersect obstacles. Starting from the target, we add the last node visible from it and restart the process from this node, as described in Algorithm 1.

We recall that two paths are in the same homotopic class if one can be smoothly deformed to the other without touching any obstacles. One important characteristic of path pruning is that unlike the method we will discuss next, homotopicity may not be preserved. If there is a game design motivation to go in between specific obstacles, for example staying behind cover in a stealth game, the path pruning can create a very different path and compromise certain qualities of the path.

The funnel algorithm is a path refinement algorithm that, given a 2D convex decomposition (e.g. a triangulation) of a polygon with holes and a path between two points, returns the shortest path belonging in the same homotopic class. We extended one of its 2D implementations by Mononen [15] to 3D environments by using the octree as the convex decomposition. Due to space limitations, we will not discuss this further, but detailed explanations can be found in Massonnat's thesis [14].

C. Path Finding in Dynamic 3D Environments

Finally, all of the work presented above can be adapted to dynamic environments, where obstacles can move, appear, or disappear. One of the main advantages of our work is that the octree can be pre-computed, and loaded quickly during runtime, which makes real-time pathfinding feasible. While this works well for static environments, if the map is updated or if obstacles move during the execution, the octree no longer reflects the environment accurately.

Re-building the octree from scratch every time the environment changes would be way too expensive, but since the octree is a hierarchical structure, most of the time a local change in the environment will only affect a small part of the octree. Because of that, we want to update the octree with local methods as much as possible. To do so, we keep track of the position and scale of dynamic obstacles in the level. Changes in the environment can only lead to two update scenarios:

- If an obstacle moves into a cell that was previously free (valid), this cell becomes invalid. If this cell is larger than the octree granularity, we have to split the cell further. This can potentially cause several sequential splits until the octree granularity is reached.
- If obstacles move out of an invalid cell and it no longer contains any obstacle, it becomes valid. It may then be possible to merge octree cells that were previously split.

Naively merging cells dynamically with the previously described greedy method causes an increase over time in the number of cells, as arbitrary choice of fine-grain cell merges can restrict higher level merging opportunities. To ensure the stability of the octree over time, we adopt a new merging strategy. When a cell is updated and becomes valid, if all of the children of the cell's parent are valid, it is possible to repair the split parent by merging the 8 valid children as a single merge operation. It might then be also possible to repair the parent's parent, and so on, going as high as possible in the hierarchy. With this merging strategy, even after many updates, the total number of cells remains stable and the octree is still efficient.

After the octree is updated, we must reflect these changes in the movement graph to finalize the update process. A naive way to do this would be to delete the old graph and recompute the new graph from scratch. This approach, however, especially on larger maps, quickly becomes inefficient and slow. Because updates in the octree are very localized (an obstacle moving will only affect a few cells at a time for instance), the number of octree cells that are updated at one moment in time is much smaller than the total number of cells in the octree. To be more efficient, we instead devise local graph update methods, that update only the nodes in the graph affected by changes in the octree.

The two types of graph updates needed are the creation and deletion of transition points. When creating a transition point between two cells, we also have to connect it to all of the other transition points starting from one of the two cells. Likewise, when deleting a transition point, it is important to delete all the edges pointing to it. With these two types of updates in mind, after each update in the octree, we update the surrounding transitions accordingly:

- If a cell was invalid and becomes invalid, we delete all transitions that started from this cell.
- If a cell was invalid and becomes valid, we create transition points between this cell and all of its valid neighbors.
- When splitting a cell, we delete all transitions that started from this cell and create the necessary transitions between the children and the neighbors, and between the children themselves.
- When restoring a parent, we delete all the transitions involving its children and add connections with all of the parent's neighbors.

With these rules in mind, we successfully reduce the number of operations on graph nodes and edges required for a given octree update, making the graph update process faster, as we will show in Section IV.

IV. Experiments and Results

In this section, we will give visual examples of important aspects of our work, evaluate the benefits of merging octrees and doing path refinement, and show a detailed comparison of our algorithm against the state-of-the-art JPS-3D algorithm [18].

To evaluate our work, we first created a series of 6 smaller maps directly in Unity3D that represent common structures and navigational challenges encountered in video games. To simulate more challenging scenarios on much larger maps, we used the 3D benchmark by Brewer and Sturtevant [4] recreating maps from the game *Warframe* published by Digital Extremes. Snapshots of maps from these datasets can be found in Figure 1. We excluded from this dataset the C-named maps that feature many more obstacles than other maps and are also excluded by other works using this dataset, and also excluded for space reasons some maps that were just very similar variants of another map, resulting in 19 maps. Each of these maps comes with 10,000 "test scenarios", consisting of a start and a target point. The task is finding the shortest path between them. Because results, especially compute time, tend to vary drastically with path length, we bucketed these points according to path length, splitting them into 10 evenly-sized buckets. Testing different algorithms on the same scenarios makes it possible to compare their performance. We are mainly interested in two factors in our experiments. Of course, the length of the path connecting two given points is important, as finding exact shortest paths in 3D is NP-hard [5] and although some compromises can be made, it is desirable to find a path with a length close to the optimal one. The other deciding factor, potentially even more important, is the compute time. Because we are interested in 3D pathfinding in video games, queries will be made in real-time and it is essential for a smooth player experience that finding a path is as fast as possible. Pathfinding costs should not exceed a second, and ideally should remain in the hundreds or tens of milliseconds. All experiments were done using Unity3D v.2021.3.13f1 on a medium-range laptop (Intel Core i7-12700H with a 2.30 GHz clock speed and 16 GB of RAM).

A. Impact of Merging and Path Refinement

To illustrate the octree decomposition and how merging affects the structure of the octree, a visualization of a finished octree in its regular and merged version on a small map can be found in Figure 2. It is clear that merging the octree greatly reduces the number of cells. The number of valid cells in the octree is crucial, as it is proportional to the number of nodes in the movement graph and therefore is the main cost of the pathfinding using A*. To assess in greater detail the impact of merging on the number of cells and compare it with the voxel baseline, we report the number of valid cells on several maps in Table I. The octree is as expected much more efficient than a simple voxel decomposition, and merging the octree reduces the number of cells even further. As can be seen in Figure 3, this reduction in the number of octree cells lowers

TABLE I
Comparison of the number of valid cells across several maps using either the voxel baseline, the regular, or merged octree.

Map name	Voxel	Unmerged	Merged
Building_1	28,190	4,226	**303**
Building_2	28,628	3,975	**147**
Building_3	29,816	4,243	**182**
Cave	29,510	8,190	**316**
Industrial	31,833	1,671	**157**
Zigzag	31,488	2,130	**52**
Complex (Warframe)	8.3M	41,385	**10,552**

in turn the pathfinding time (similar trends can be observed on all handmade maps but were omitted for space reasons). One last thing to discuss is the increase in compute times with the voxel baseline that can be seen for paths between 10 and 15 units long. We attribute this to the level geometry: short paths are easier to compute as the A* algorithm will explore fewer cells, and long paths tend to be in large open parts of the maps. The characteristic length in between (about half of the map size) could correspond to the typical length of more challenging paths that weave in between obstacles.

We illustrate how path refinement works in Figure 4. In this example, we use an octree to find a path between the start (green sphere) and the target (blue sphere) that avoids stalactites in a cave-like environment. The red path is the "default" path found by executing the A* algorithm on the movement graph, but because nodes are always at the center of the transitions between cells, there can be some superfluous movement. The yellow path shows the result of the funnel algorithm, which makes the path more direct. Path pruning can then be applied to make the path even shorter, as shown with the green path. Testing on 10,000 points for all 6 maps showed a 5-10% decrease in average path length after performing path refinement, and the most improvement by combining path pruning and the funnel algorithm.

Let us now discuss the variance of the results obtained. For all the experiments in this section, the observed variance is similar. By sampling a large number of pairs of points and especially by bucketing these points according to path length, we aim to reduce the variance. Within a single bucket, the variance in path length is very small. When it comes to compute time, the variance is larger. The first and third quartiles for a given bucket are generally 30% lower and higher than the median. Even if computing the same path over and over would give nearly identical compute times, within a single bucket, paths of similar length can be of varying complexity, with some paths being straight lines in large open spaces and others weaving between obstacles.

To summarize our findings in this section, studying the baseline shows that naive voxel grids are not suitable for 3D pathfinding. Octrees allow for faster pathfinding, and our merging algorithm vastly improves performance. If there is a special motivation to respect homotopy classes, the funnel algorithm is relatively cheap and reduces path length, and if

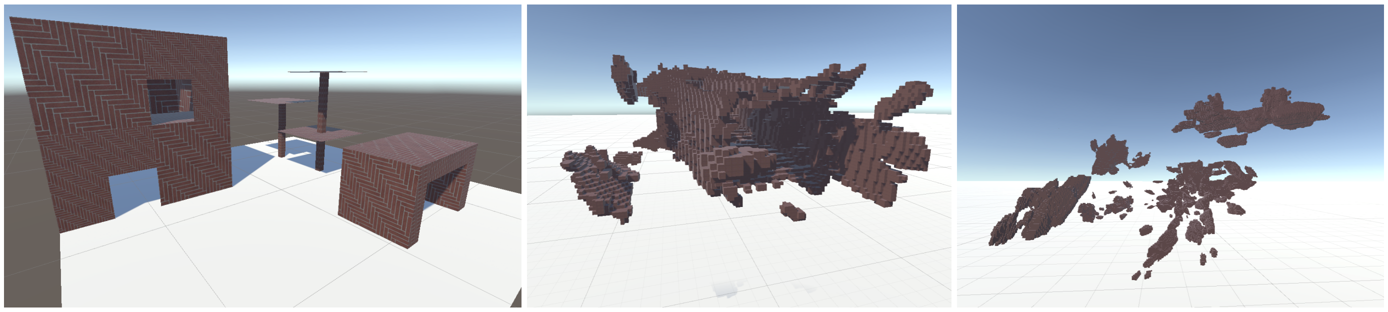

Fig. 1. Snapshot of the *Industrial* map from our handmade dataset (left), and the *Complex* (middle) and *Full4* (right) maps from the *Warframe* dataset.

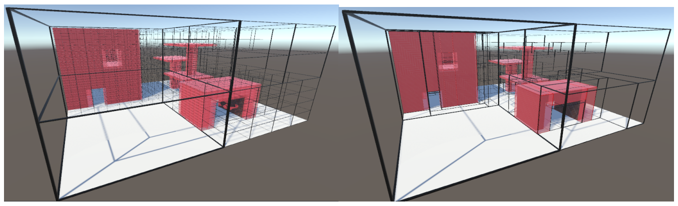

Fig. 2. Example of an Octree decomposition before (left side) and after (right side) merging. Invalid cells are represented in red, and the black frames indicate the location of valid octree leaves.

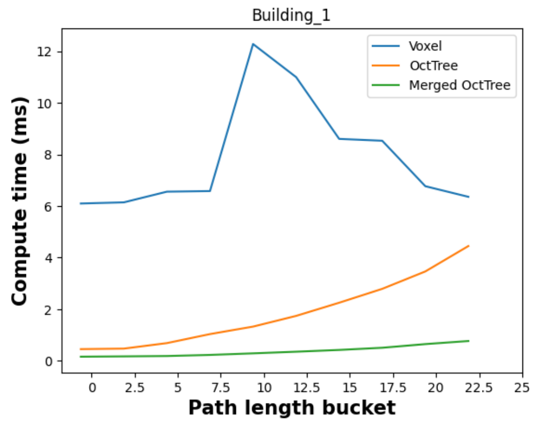

Fig. 3. Average pathfinding times on the Building_1 map with the voxel baseline, the regular and merged octree. Test scenarios are split into 10 buckets according to path length to reduce variance.

the main concern is path length, combining path pruning and the funnel algorithm gives the best results. On larger maps especially, path pruning only represents a fraction of the total pathfinding cost, with the main cost originating from the A* algorithm. From these results, we gather that using merged octrees along with both path pruning and the funnel algorithm gives the best results, and it is this approach that we will evaluate on larger maps in the rest of this section.

B. Experiments on the Dynamic Octree

Although no benchmark exists for 3D pathfinding in a dynamic setting, we can reuse our previous dataset and add randomly moving obstacles while maintaining and updating a given path. Table II shows average update costs over 1,000

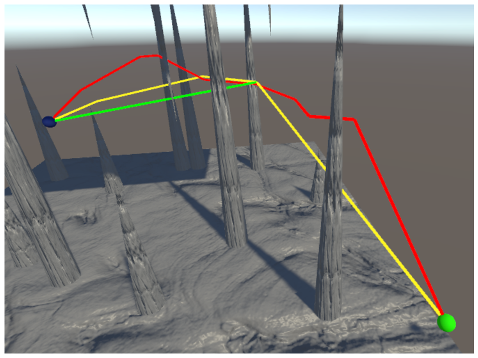

Fig. 4. Shortest path between two points found using the octree structure (red), after applying the funnel algorithm (yellow), and after funnel and path pruning (green).

TABLE II
AVERAGE COST OF THE OCTREE UPDATE, GRAPH UPDATE, AND PATH RECOMPUTING IN MILLISECONDS OVER 1,000 UPDATES ON THE INDUSTRIAL MAP, USING EITHER REGULAR (O) OR MERGED OCTREES (MO), AND LOCAL GRAPH UPDATES (LGU) OR REBUILDING THE GRAPH

Update method	Octree update	Graph update	Path recomputing	Total time
O + no LGU	1.36	0.92	1.82	4.10
O + LGU	**0.22**	**0.03**	1.26	1.51
MO + no LGU	0.44	0.34	0.72	1.50
MO + LGU	0.39	**0.03**	**0.55**	**0.97**

updates with 3 moving obstacles on the *Industrial* map. Updating a merged octree is faster since it contains fewer cells, and locally updating the graph is much faster than recomputing it entirely. Using local methods, updates can be made in just one millisecond on average, making real-time updates to the octree and pathfinding in dynamic environments feasible.

C. Comparison with JPS-3D

To test our approach in more realistic settings, we will now run experiments on the 19 maps selected from the *Warframe* dataset and compare our results against the JPS-3D algorithm. The median and average compute times on all maps can be found in Figure 5. The results obtained vary with the maps, mainly because of their size and the number of obstacles they contain. On most maps, the average compute time remains

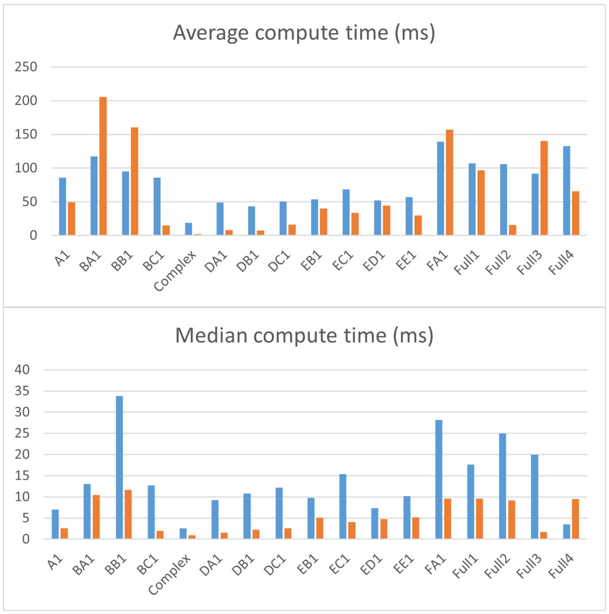

Fig. 5. Average and median compute time of merged octrees (in blue) and JPS-3D (in orange) on the 3D benchmark *Warframe*.

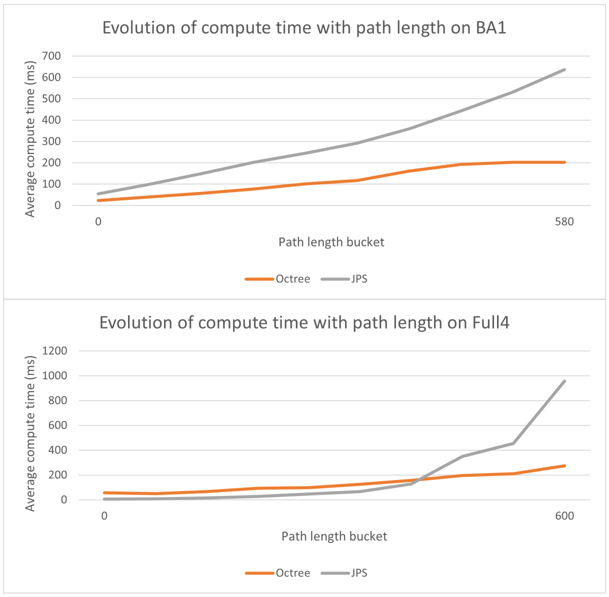

Fig. 6. Evolution of the compute time with path length on the BA1 and Full4 maps.

under 150 milliseconds, which remains feasible for real-time applications.

Our path refinement consistently produces shorter paths than those obtained with JPS-3D and leads to a 5% decrease in the average path length.

The JPS algorithm is faster on most maps. We can compare these average times in terms of speed-up factor of JPS over our work, and the geometric mean over all 19 maps is 2.05 for average and 2.77 for median times. Although tested on the same machine, JPS is implemented in C++ and our work in Unity, so overhead and rendering costs could affect the octree performance. We chose Unity to promote applicability in video game development. We will now give more details about these results to gain more insight into the behavior of these two algorithms.

The median times are much lower than the average times, meaning that the compute time distribution is skewed to the right. Certain path instances are very long or complex and can cause very high compute times. If we look at maximum compute times, certain instances can take over a second to solve with octrees, and up to 10 seconds with JPS. As we saw on the handmade dataset, pathfinding times can vary significantly with path length. We order the data points by path length and report the average compute time for each 10-percentile on two maps from the *Warframe* dataset in Figure 6. The octree is on average two times faster than JPS on the BA1 map and two times slower on the Full4 map, but both maps show the same trend. As expected, pathfinding times increase with path length, but this increase is much less noticeable with merged octrees than with JPS. On the longest instances especially (paths in the last quartile), compute times significantly increase with JPS, which is not the case with octrees.

To conclude, our work produces very consistent results, regardless of the path length or the environment, and finds paths in a few hundred of milliseconds, which is viable for many real-time video game applications. While the JPS-3D algorithm is faster on average, merged octrees and the addition of path refinement produce shorter paths and are faster in certain scenarios, mainly for finding longer paths. A more optimized implementation of our work in C++ would likely also greatly reduce the performance gap.

V. Conclusion and Future Work

3D pathfinding is crucial for NPC path planning in games and being able to exploit the full range of 3D movements can open many new possibilities in game development. However, it remains computationally expensive compared to 2D pathfinding. Efficient representation of 3D environments is key, leading us to develop a hierarchical octree approach with a successful greedy merging method, reducing the number of valid cells in the octree by up to an order of magnitude. Since the octree and the graph can be computed and saved offline, and loaded at run-time, we achieve very fast compute times. On smaller maps paths can be found in just a few milliseconds, and even on very large maps such as maps from the *Warframe* dataset compute times typically range only in the hundreds of milliseconds.

The visibility-based path pruning approach and the 3D extension to the 2D funnel algorithm lower path length by 5 to 10% on average while being cheap to compute compared to the A* algorithm, and in the case of the funnel algorithm also maintains the homotopy class of a path.

A detailed comparison with the state-of-the-art JPS-3D algorithm [18] shows that, even if our approach is slower on average, our compute times remain in the same order of magnitude and are faster for longer paths. Our path refinement methods also produce shorter paths and provide more flexibility.

We extended all of this work to dynamic environments by locally updating the octree and the graph as changes in the environment are detected. By changing the dynamic octree method to a cell-repairing approach instead of the greedy merging we ensure a stable representation and number of octree cells even after many updates. Thanks to its locality the update process remains fast, with average costs of less than a millisecond.

We identified several interesting directions that future work on this subject could take. First, we used an axis-aligned bounding box representation for obstacles, mainly because it suited the existing voxel benchmarks. Other, more involved forms of obstacle representation could also be considered. In some 3D benchmarks, for example, maps require extremely large numbers of voxels to be represented, and a less constrained representation, polyhedral representation, even if still discretized, may reduce the cost of initially building the octree.

Our design is also limited by the way the octrees most naturally fit a cubic volume with each dimension a natural power-of-2. Some maps in the *Warframe* dataset were very elongated, and in general, game environments may not easily fit in a large octree cube. One way to address this would be to start the octree building process with a rectangular cell instead of a cube, and split the first rectangular cell into evenly-sized cubes, which can then be individually built like a regular octree. This method would also add another form of hierarchy, in which different octrees can be built for different regions, and in very large game environments one could only load the local octree relevant to the region the player is in.

While we focused on the shortest path problem, in many games other constraints should be taken into account. If the moving agent is a spaceship for example, paths should incorporate dynamic constraints such as inertia or a limited turning radius and smoother changes of direction to allow vehicle movement. The graph initially constructed based on partition transitions in the octree may provide some flexibility to consider agent volume or turn radius, with some additional limitations on granularity and connectivity. Extensive research has already been conducted on this subject, and an approach such as using Bezier curves to obtain smoother paths [24] could also be used, although additional work would be needed to ensure that smoothed paths do not collide with obstacles.

Acknowledgment

We thank Thomas Nobes for sharing their JPS-3D code.

References

[1] Zeyad Abd Algfoor, Mohd Shahrizal Sunar, and Hoshang Kolivand. "A Comprehensive Study on Pathfinding Techniques for Robotics and Video Games". In: *International Journal of Computer Games Technology* 2015 (Apr. 2015), pp. 1–11.

[2] Adi Botea, Martin Müller, and Jonathan Schaeffer. "Near optimal hierarchical path-finding." In: *J. Game Dev.* 1.1 (2004), pp. 1–30.

[3] Daniel Brewer. "3D Flight Navigation Using Sparse Voxel Octrees". In: *Game AI Pro 3*. 2017, pp. 265–274.

[4] Daniel Brewer and Nathan R. Sturtevant. "Benchmarks for Pathfinding in 3D Voxel Space". In: *Symposium on Combinatorial Search (SoCS)* (2018), pp. 143–147.

[5] John Canny and John Reif. "New lower bound techniques for robot motion planning problems". In: *28th Annual Symposium on Foundations of Computer Science (sfcs 1987)*. 1987, pp. 49–60.

[6] Xiao Cui and Hao Shi. "A*-based Pathfinding in Modern Computer Games". In: *International Journal of Computer Science and Network Security* 11 (Nov. 2010), pp. 125–130.

[7] Jeff Erickson. *Shortest Homotopic Paths*. Lecture notes from CS 598 at University of Illinois. 2009.

[8] Guillermo Frontera et al. "Approximate 3D Euclidean Shortest Paths for Unmanned Aircraft in Urban Environments". In: *Journal of Intelligent & Robotic Systems* 85 (Feb. 2017), pp. 353–368.

[9] Daniel Harabor and Alban Grastien. "Online Graph Pruning for Pathfinding on Grid Maps". In: *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*. 2011, pp. 1114–1119.

[10] Emili Hernandez, Marc Carreras, and Pere Ridao. "A comparison of homotopic path planning algorithms for robotic applications". In: *Robotics and Autonomous Systems* 64 (2015), pp. 44–58. ISSN: 0921-8890.

[11] John Hershberger and Jack Snoeyink. "Computing minimum length paths of a given homotopy class". In: *Computational Geometry* 4.2 (1994), pp. 63–97. ISSN: 0925-7721.

[12] Stefan Hertel and Kurt Mehlhorn. "Fast triangulation of simple polygons". In: *Foundations of Computation Theory: Proceedings of the 1983 International FCT-Conference*. Springer. 1983, pp. 207–218.

[13] Fangyu Li et al. "Universal path planning for an indoor drone". In: *Automation in Construction* 95 (2018), pp. 275–283. ISSN: 0926-5805.

[14] Quentin Massonnat. "Efficient Octree-based 3D Pathfinding". MSc thesis, McGill University, 2024.

[15] Mikko Mononen. *Digesting Duck: Simple Stupid Funnel Algorithm*. http://digestingduck.blogspot.com/2010/03/simple-stupid-funnel-algorithm.html. 2010.

[16] G. M. Morton. *A Computer Oriented Geodetic Data Base; and a New Technique in File Sequencing*. Tech. rep. IBM Research, 1966.

[17] Timur Muratov and Aleksandr Zagarskikh. "Octree-Based Hierarchical 3D Pathfinding Optimization of Three-Dimensional Pathfinding". In: *Proceedings of the 2019 3rd International Symposium on Computer Science and Intelligent Control*. 2020, pp. 1–6. ISBN: 9781450376617.

[18] Thomas K. Nobes et al. "The JPS pathfinding system in 3D". In: *Proceedings of the International Symposium on Combinatorial Search*. Fifteenth InternationalSymposium on Combinatorial Search 1 (2022), pp. 145–152.

[19] Thomas K. Nobes et al. "Voxel Benchmarks for 3D Pathfinding: Sandstone, Descent, and Industrial Plants". In: *Proceedings of the Sixteenth International Symposium on Combinatorial Search*. 2023, pp. 56–64.

[20] Michael Schwarz and Hans-Peter Seidel. "Fast Parallel Surface and Solid Voxelization on GPUs". In: *ACM SIGGRAPH Asia 2010 Papers*. 2010, pp. 1–10. ISBN: 9781450304399.

[21] David Sislák, Premysl Volf, and Michal Pechoucek. "Flight Trajectory Path Planning". In: *Proceedings of the 19th International Conference on Automated Planning & Scheduling (ICAPS)*. 2009, pp. 76–83.

[22] Nathan Sturtevant. "A Sparse Grid Representation for Dynamic Three-Dimensional Worlds". In: *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* 7.1 (2011), pp. 73–78.

[23] Wouter van Toll et al. "Comparing navigation meshes: Theoretical analysis and practical metrics". In: *Computers & Graphics* 91 (2020), pp. 52–82. ISSN: 0097-8493.

[24] Kwangjin Yang and Salah Sukkarieh. "3D smooth path planning for a UAV in cluttered natural environments". In: *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2008, pp. 794–800.

[25] W. Zeng and R. L. Church. "Finding shortest paths on real road networks: the case for A*". In: *International Journal of Geographical Information Science* (2009), pp. 531–543.