Efficient Octree-based 3D Pathfinding

Quentin Massonnat, School of Computer Science

McGill University, Montreal

December, 2023

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of

Master of Computer Science

Abstract

Three-dimensional virtual environments are commonplace in modern games. Non-player character (NPC) character movement planning, however, is still largely 2D, with off-plane or vertical movement modeled through limited, and frequently custom connections between 2D surfaces. Naive extension of 2D pathfinding methods to 3D by building a full 3D grid (voxelization) is possible but easily becomes too expensive for non-trivial 3D spaces. In this thesis 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.

Our design begins with an octree representation, which already provides more flexibility than a more traditional voxelization. By merging adjacent cells while preserving their convexity, we 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 novel implementation of the classic "funnel" algorithm that computes minimal homotopic paths, in our case extended to our 3D environment. 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. 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.

Abrégé

La plupart des jeux vidéo récents offrent des expériences de jeu dans des environnements virtuels en 3 dimensions. La planification du mouvement des PNJ (personnages non joueurs) reste quant à elle majoritairement en 2 dimensions, et le mouvement vertical est souvent limité ou utilise des connections arbitraires. L'extension naïve de méthodes de recherche de chemins en 2D à la 3D est possible en construisant une grille en 3D (voxelisation), mais devient rapidement trop coûteuse pour des espaces 3D non triviaux. Dans cette thèse nous proposons une solution efficace à la recherche de chemins en 3D où nous construisons une représentation hiérarchique et réduite de l'environnement, à laquelle nous étendons des approches de mesh de navigations 2D.

Ce travail utilise des octrees, plus flexible qu'une simple voxelisation. Nous fusionnons ensuite des nœuds adjacents en préservant leur convexité pour obtenir une représentation plus efficace de l'environnement qui réduit grandement les temps de calcul. Nous construisons ensuite un graphe de navigation à partir de cet octree sur lequel nous cherchons des chemins avec l'algorithme A*. Pour améliorer la qualité des chemins obtenus, nous implémentons deux formes de raffinement de chemins : une heuristique basée sur la visibilité qui simplifie un chemin, et une nouvelle implémentation en 3D de l'algorithme de l' "entonnoir", qui calcule le plus court chemin homotopique, dans notre cas dans un environnement 3D. Nous étendons notre travail pour prendre en compte des environnement dynamiques, en mettant à jour localement et efficacement l'octree et le graphe de navigation.

Des expériences sur divers scénarios montrent que notre approche reste rapide et efficace même sur de très grandes cartes en 3D, et pourrait être utilisée pour la recherche de chemins en temps réel dans des jeux vidéo. Nous avons implémenté notre travail sur Unity, un des moteurs de jeux vidéo les plus populaires, dans l'optique de rendre la recherche de chemin en 3D accessible à des développeurs de jeux vidéo.

Acknowledgements

First and foremost I want to thank my supervisor, Clark Verbrugge, for his invaluable help and support throughout these last two years. This thesis would not exist without him.

I also thank Thomas Nobes for providing his results and code on the 3D JPS algorithm and his openness to discuss his work.

Thanks to my labmates, Wael al Enezi and Nolwenn Augras, as well as all the great friends I made at McGill for all the work and mostly non-work discussions that helped shape this thesis into the great experience it was.

Thanks to my family for their unwavering support and for always being there when I needed it.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Abbreviations

2D, 3D: 2-dimension, 3-dimension

AABB: axis-aligned bounding box

FOV: field of view

HPA*: hierarchical pathfinding A*

JPS: jump point search

OBB: oriented bounding box

NP: non-deterministic polynomial time

NPC: non-player character

RRT: rapidly exploring random tree

SSF: stupid simple funnel algorithm

SVO: sparse voxel octtree

Chapter 1

Introduction

Three-dimensional (3D) virtual environments are commonplace in modern games. They cover a variety of genres, from first-person shooters and role-playing games to adventure games. Even though most games feature 3D maps, movement in these environments is mostly 2-dimensional, with more or less emphasis on verticality. If a character can walk around but only jump or use ladders to change elevation, movement is not fully 3-dimensional.

As games get more complex, both from a graphical and mechanical point of view, game designers increasingly include complex 3D movement in their games. Any game featuring full 3D movement can benefit from this: flight or space simulations where the player can fly, but also games that give great vertical mobility to the player such as the recent *Armored Core 6*'s [5] jetpacks or *Marvel's Spider-Man*'s [3] web-swinging mechanic. There needs to be a robust pathfinding system to accompany such environments and mechanics in a game. Pathfinding can give the player guidance on the way to reach a destination, and more importantly is used to plan the movement of non-player characters (NPCs), such as companions following the player or moving towards an objective, or enemies hunting the player.

Most games, however, "cheat" when it comes to 3D movement planning. In games with limited vertical movement, pathfinding is often restricted to the horizontal plane,

and off-plane or vertical movement is handled by custom waypoints such as ladders or climbing points. In space simulations, since most of the environment is empty, the path to an objective is usually a straight line. In more complex settings, games use different tricks to simplify movement planning. In *Marvel's Spider Man*, flying enemies are in fact hovering well above obstacles and move mostly horizontally at this height. *Elex II* [4] features complex urban, industrial, or post-apocalyptic environments and flying enemies, but these enemies wander around until they have a line of sight on the player, at which point they fly straight at it. In *Armored Core 6*, the player can navigate through very intricate 3D environments full of narrow paths and holes, but enemies are mostly static or only attack and follow the player in large open rooms.

Since games have to run in real time and most of the computing resources go towards graphics and rendering, these simplified pathfinding methods work without breaking the player's immersion. With these few examples, we illustrate the fact that motion planning in full 3D environments remains an open problem for the game industry. Being able to solve 3D pathfinding in complex environments, where paths weave around obstacles, would open many opportunities for game design and more realistic movement.

The main problem lies in the fact that pathfinding in 3D is much more expensive than in 2D. 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 more complex than in 2D. 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.

1.1 Contribution

In this thesis, we propose 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. We then use a Hertel-Mehlhorn approach [28] to merge adjacent cells and obtain a coarser representation that greatly reduces path computation costs. From this representation, we create a movement graph (roadmap) to find short paths between two points, and use path refinement techniques to further reduce path length. The first one is a visibility-based heuristic inspired by Yang and Sukkarieh [46]. The second and more involved technique is an extension of the 2D funnel algorithm [27] that builds a shorter 3D homotopic path.

We also extend our pathfinding to dynamic 3D environments that contain moving obstacles. With some modifications, it is possible to efficiently update the octree and the associated roadmap during 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 benchmarks to evaluate it, including basic test levels as well as more complex game environments meant to be representative of topologically and navigationally complex environments in which 3D pathing may be useful. Timing and path length comparisons show that our octree approach is vastly superior to basic voxelization, and that merging octree cells significantly reduces the number of cells and path compute times. The visibility-based path pruning and the homotopic path refinement reduce path length successfully while being fast in comparison to the pathfinding itself, especially in larger and more complex environments.

The main contributions of this thesis are as follows:

1. We implemented the traditional octree data structure in Unity, and vastly improved it with the Hertel-Mehlhorn style merging

2. We describe and define a novel extension of Mononen's classic 2D funnel algorithm [35] to a 3D, octree context.

3. Our approach is evaluated on a set of (non-voxelized) 3D custom scenes inspired by modern game environments. This is a useful addition to existing pathfinding benchmarks which concentrate on 2D models and discretized voxel representations.

4. Experimentation shows that our approach is sufficiently efficient to be useful in complex game environments, drastically improving over a more naive voxelization.

5. We provide a thorough comparison of our approach and what is to our knowledge the state-of-the-art 3D pathfinding method by Nobes et al. [37]. We compare their performance on the large voxel-based maps from the game *Warframe* [2], converted into a 3D benchmark by Brewer and Sturtevant [10], as well as a more recent dataset by Nobes et al. [38] that covers a more diverse range of scenarios. Our approach provides more flexibility and shorter path lengths while maintaining feasible compute times.

1.2 Outline

This thesis comprises 6 chapters, and is organized as follows:

- In Chapter 1 we discussed the motivation behind this thesis and provided an overview of our work.

- We give some necessary background knowledge on graph, 2D, and 3D pathfinding and explore the related work in this area in Chapter 2.

- We present the voxel baseline, then describe the more efficient octree data structure and the octree merging process in Chapter 3.

- In Chapter 4 we explain how we use this octree structure to perform fast pathfinding and describe in detail the path pruning and the funnel algorithm that we use for path refinement.

- We discuss how to update the octree and the movement graph to handle dynamic environments in Chapter 5.
- We evaluate our approach on a handmade dataset in Chapter 6, and compare it to Nobes et al.'s approach [37] on the larger-scale *Warframe* dataset and the dataset created by Nobes et al. [38].
- Chapter 7 concludes our work and discusses possible directions for future work based on our observations.

Chapter 2

Background and Related Work

Pathfinding techniques are required in many genres of video games. In any non-trivial environment that contains obstacles, it can provide help to the player or be used to navigate non-player characters (NPCs).

The problem of pathfinding in a game consists of finding a path between a starting point and a target point, that should satisfy additional conditions depending on the use case:

- The path should avoid obstacles in the environment, such as walls, trees, etc.
- In most cases, we want the path to be optimal or close to optimal in length.
- For more realistic movement, especially in the case of moving vehicles, dynamical constraints such as inertia or a limited turning radius can be imposed.
- When the moving agents have a non-zero size, the path should maintain a safe distance from obstacles, for example by not grazing corners, to account for the size of the agent.
- Since our aim is pathfinding in video games, computational efficiency is also a major concern.

In this thesis, we will address the problem of finding obstacle-avoiding shortest paths in 3D environments.

In this chapter, we will provide the reader with the necessary background elements as well as an overview of the research related to this topic. We will first provide a summary of the main search algorithms in Section 2.1. These search algorithms typically work on a graph and, therefore need to be coupled with some environment decomposition method that builds a representation of the walkable space called navigation mesh (navmesh). From this navigation mesh, it is possible to create a graph, or roadmap, that abstracts the movement in the level. Various approaches have been developed for 2D environments, which we will explore in Section 2.2, and we will cover the extensions to 3D environments in Section 2.3. Building an accurate and efficient navigation mesh is essential, as it leads to a smaller roadmap, and the compute time of search algorithms increases with the size of the graph. We discuss some attempts at building efficient navmeshes in Section 2.4. In this thesis, we chose to use a hierarchical representation of the environment, the octree. In Section 2.5, we will give details about the quadtree data structure, its 3D extension, the octree, and how they can be used to perform fast pathfinding. After a path is found on the navmesh using one of these search algorithms, it is desirable to refine it, reduce its length, or make it more smooth. We discuss different methods to improve path quality in Section 2.6. Finally, comparing research and the performance of different approaches on the same benchmark is essential. Even though many benchmarks exist for 2D pathfinding, the number of 3D benchmarks remains limited. We describe these in Section 2.7.

2.1 Search Algorithms

Searching for shortest paths is typically done in a graph, as a simplified abstraction of the walkable space. In a 2D game, such a graph can be a grid of tiles the player can navigate, and that accommodates movement in 4 directions, or 8 if diagonal movement is allowed.

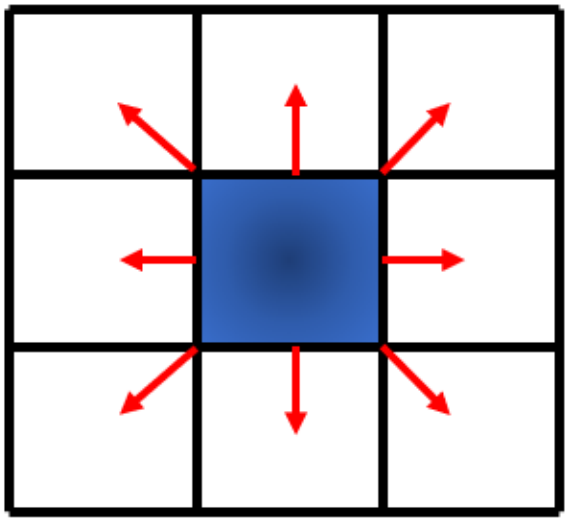

Figure 2.1: Visualization of the octile movement to the 8 possible neighbors of a tile in 2D. In 3D, it is also possible to access the 9 neighbors above a cell and the 9 below, resulting in a total of 26 neighbors.

In a 3D game, a similar 3D grid can be used, allowing movement in all 26 directions (from a given cell, the next move can be any of the 8 neighbors at the same elevation, the 9 neighbors that are one unit higher and the other 9 one unit lower, as illustrated in Figure 2.1). But from an abstract perspective, pathfinding can be done on any graph, by minimizing the sum of the costs of the edges along the path between the start and target point.

We describe the A* algorithm, a cornerstone of pathfinding search algorithms, in Section 2.1.1. There exist many variants and improvements to A*, but we will focus our attention on the jump point search algorithm (JPS) in Section 2.1.2, as it works well in game environments and its 3D extension is one of the fastest pathfinding methods for 3D games as of writing this thesis. We will end this section by giving a quick overview of some of the other important search algorithms in Section 2.1.3.

2.1.1 The A* Algorithm

The most well-known search algorithm is the A* algorithm [25], an optimization over the base Dijkstra's algorithm [15]. It uses a heuristic function, such as the Euclidean or Manhattan distance, to evaluate the distance between a node and the target node in the graph. This allows to guide the search towards nodes that are closer to the target point, and thus more likely to be on an optimal path.

For the following, if x and y are two nodes in the graph, we write $d(x, y)$ the cost of the edge connecting x and y. We call the start node s and the target node t. For a node n, $g(n)$ is the cost of the shortest path from s to n, $h(n)$ a heuristic estimation of the distance between n and t, and $f(n) = g(n)+h(n)$ the priority function that will determine the order in which we search nodes. The heuristic h can be the Euclidean distance separating n and t, the Manhattan distance, or any other distance function. If the heuristic function never overestimates the actual cost of the path between n and t, the A* algorithm is guaranteed to find the shortest path from s to t [25].

We give pseudo-code for this algorithm in Algorithm 1. At each iteration, we select the node n with the lowest f value, we update the g and f values of its neighbors n' if there is a new shorter path going from s to n' going through n. During the execution, we save and update for each node its predecessor, i.e., its neighbor connecting it on the shortest path to s, allowing us to reconstruct a path starting from t after the algorithm has run successfully.

2.1.2 The Jump-Point Search Algorithm

Another popular search algorithm is the jump-point search (JPS) algorithm [24] for video game spaces. This algorithm is an evolution of the A* algorithm and directly uses the grid structure of a 2D environment to reduce the number of searched nodes, resulting in faster searches. The intuition behind this is that, especially in open spaces, many paths between two points are equivalent in length, as shown in Figure 2.2.

Algorithm 1 Pseudocode for the A* algorithm

Input: A starting point s and a target point t
Output: The shortest path from s to t

```
Let openSet = { s }
while openSet is not empty do
    Let n be the node in openSet with the lowest f value
    if n == t then
        Reconstruct the path from s to t and end the search
    end if
    remove n from openSet
    for all neighbors n' of n do
        if g(n) + d(n, n') < g(n') then
            Let g(n') = g(n) + d(n, n'), update f(n') and add n' to openSet
        end if
    end for
end while
```

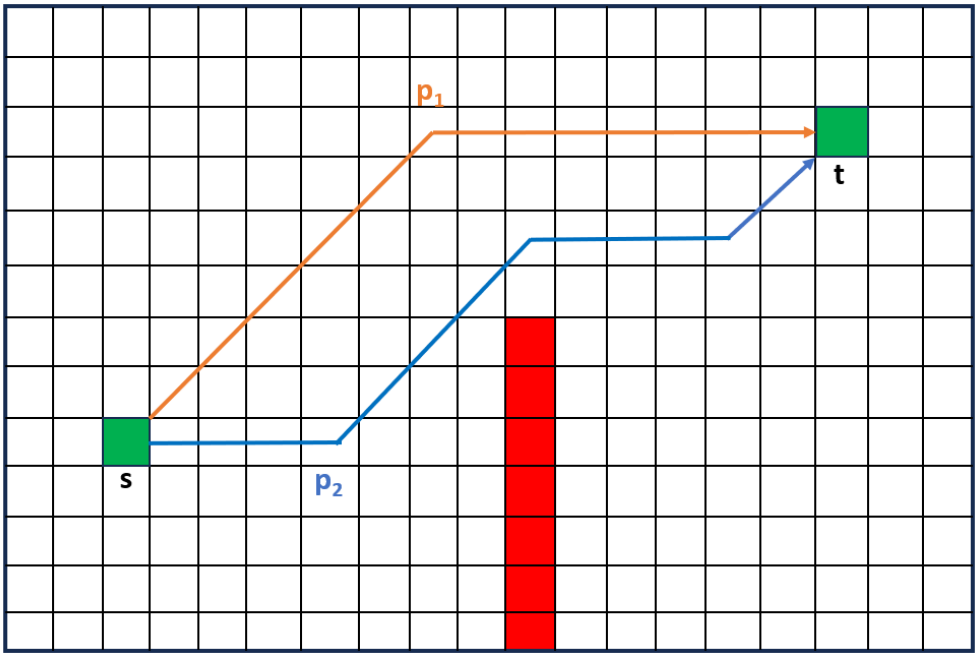

Figure 2.2: Example of 2 equivalent paths on a 2D grid. Here the two paths p_1 (orange) and p_2 (blue) connect the two points s and t with the same (optimal) length.

Since only one path of minimal length must be found, when opening a node and looking at its neighbor to continue a candidate path, it is possible to ignore most of its neighbors depending on the direction we are coming from, as they would result in equivalent paths or paths that would be explored by another candidate path. We give an

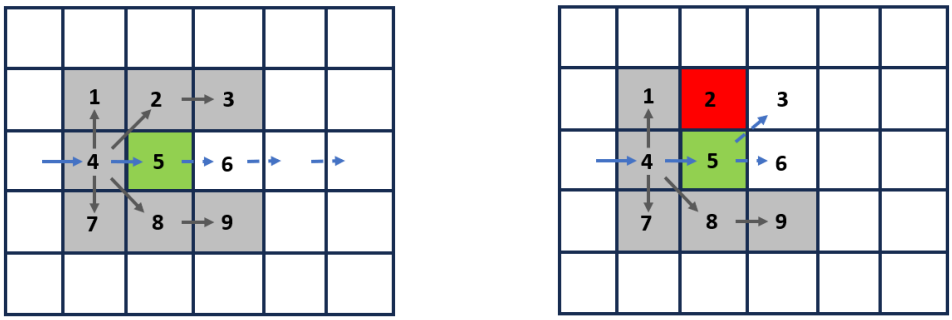

Figure 2.3: Explanation of the neighbor skips. On the left figure, if the path comes from (4), there is a shorter or equivalent path that goes to every neighbor of (5) except (6), thus we can ignore them and keep going straight. If there is an obstacle such as node (2) on the right figure, more nodes should be added to the queue, nodes (3) and (6) in this case.

example of this in Figure 2.3. In this figure, the green node (5) is opened coming from the left side (4). If there are no obstacles around the node (left figure), there is no need to open node (4) again since this is where the path comes from, and nodes (1), (2), (3), (7), (8) and (9) can all be accessed with a shorter or equivalent path that goes through the predecessor (4) but not the current node (5). Since (4) was opened, these grayed-out nodes were either already explored or nodes allowing to reach them were added to the open set. The only remaining option is to open node (6) and keep going straight. As long as there are no obstacles, this process will be repeated and the candidate path ”jumps” ahead in a straight line. If there is an obstacle (drawn in red in the right figure), more neighbors must be explored. Here, for example, there is no other equivalent path that leads to (3) since (2) is blocked off, therefore both (3) and (6) must be added to the queue.

This optimization greatly reduces the number of nodes that must be opened, and in practice, the skipping of nodes results in horizontal, vertical, or diagonal ”jumps” during the exploration, to jump points located around obstacles, hence the name of the algorithm.

The JPS algorithm maintains the A* optimality and is faster by an order of magnitude on 2D grids [24]. 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. [37] to perform fast pathfinding in large 3D

environments, and is to our knowledge the state-of-the-art method for 3D pathfinding. We will discuss this approach in more detail in Section 2.3.

2.1.3 Other Navigation Algorithms

There is a variety of other search and navigation algorithms, especially if heuristics are introduced to perform faster non-optimal (in terms of length) searches.

- The rapidly exploring random tree (RRT) algorithm, first introduced by LaValle [32], randomly samples points in the environment and connects them to an expanding tree originating from the start node, until the target can be reached from a leaf to the tree. Starting from a map with obstacles, a start point, and a target point, points are sampled randomly. If a sampled point is outside of any obstacles, we try connecting it to the tree. We compute the nearest tree leaf, and if the straight line path connecting them is free of obstacles, we add the sampled point to the tree by connecting it to the leaf.

- Another popular category of approaches, especially for handling the simultaneous movement of a large number of agents, uses steering forces in an attempt to recreate the natural behavior of a flock of birds for instance. Reynolds [39] uses a simple vehicle model that allows agents to dynamically follow targets, flee from them, or reach target locations.

- Pathfinding using potential fields is mostly used in robotics [30], and associates attractive forces to the target and repulsive forces to obstacles to compute local forces on the agent and move in a way that maximizes the potential energy.

These last two methods are heuristic and most importantly work locally. This is an advantage as it greatly reduces computing times and performance would be similar regardless of the size of the environment, but by moving locally, there is no global path planning. This can result in longer paths or even in trajectories that get stuck in local optima and cannot reach the goal in the case of potential fields.

LaValle [31] provides a good, comprehensive review of many pathfinding techniques, such as A*, RRT, navigation meshes, and road maps that were discussed above.

2.2 Pathfinding in 2D Environments

Most pathfinding research focuses on 2D environments and uses the search algorithm described above. Abd Algfoor, Sunar, and Kolivand [6] did a comprehensive study of pathfinding in games, but mostly for 2D environments. Such a review does show the great variety of approaches that can be used to tackle pathfinding, such as grids, navmeshes, hierarchical or probabilistic methods to cite a few.

Cui and Shi [14] review A* approaches for 2D pathfinding, underlining the fact that the quality and efficiency of the underlying data representation are essential. Since A* is an optimal algorithm, optimizations should be made on other aspects that could facilitate the search, such as optimizing the underlying search space, exploring different heuristic functions, or reducing memory costs. In our case, if we perform pathfinding on a graph and not on a grid, reducing the number of nodes in the graph while maintaining a faithful representation of the environment will reduce pathfinding compute times.

Botea, Müller, and Schaeffer [8] propose an improvement to the A* algorithm: HPA* (hierarchical pathfinding A*) that abstracts an environment in local 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. This is especially useful when planning on very large maps. With a global method like A*, as the map and the underlying graph increase in size, compute times can quickly become too long for a real-time setting. With a hierarchical method, however, since the map is divided into smaller chunks, path queries can remain fast even on large maps.

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 intersecting any obstacles. There can be multiple classes due to the presence of obstacles. Hernandez, Carreras, and Ridao [26] introduce homotopic variants of A*, RRT, and the *Bug algorithm* (a common pathfinding algorithm in robotics) to find shortest homotopic paths in 2D scenarios, by restricting the search to a certain homotopy class. In this thesis, we will approach this problem in a slightly different way, as we will instead find a path, and then find the shortest homotopic path within its homotopy class.

Sturtevant [42] 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. Most 3D games use this "2.5D" logic, in what is sometimes called *Manhattan environments*, as they consist of a plane and vertical obstacles akin to buildings. We are interested in tackling problems in 3D environments with full freedom of motion, and will describe some interesting approaches in the next section.

2.3 Pathfinding in 3D Environments

Due to a higher branching factor and higher volumes, 3D pathfinding is a more difficult problem that in many cases can only be solved approximately. Canny and Reif [11] 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 some heuristics and show a trade-off between path quality and computational cost. Even if A* can find shortest paths in a graph, a trade-off has to be made at some point in building that graph. For example, when searching paths in a grid, the grid size might not exactly conform to the obstacles, and even if paths close to the shortest length can be found, we may not be able to reach the exact optimal length, as illustrated in Figure 2.4. Usually

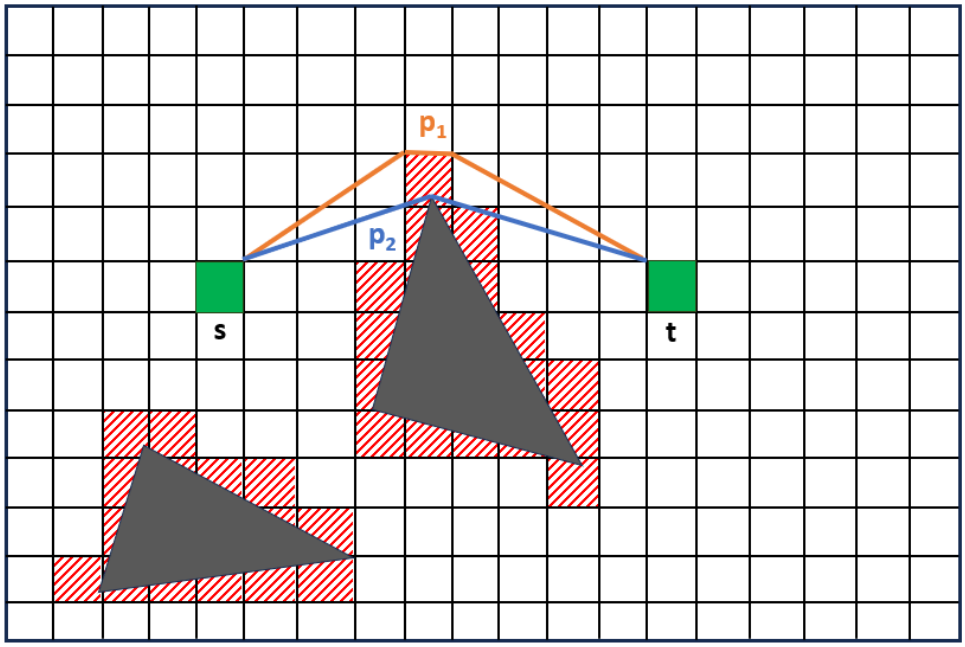

Figure 2.4: Example of the grid size leading to bad obstacle conformance. Since invalid (red) tiles can be larger than obstacles (drawn in gray) and also contain walkable space, the shortest path in a grid (here p_1) is not always the optimal path (p_2).

compromises in path length optimality are made to allow faster compute times, especially when considering real-time pathfinding applications such as video games or robotics.

Many improved versions of the A* algorithm exist for 3D. Sislák, Volf, and Pechoucek [41] explore flight trajectory path planning, but do not incorporate a reduced representation of the environment such as navigation meshes. Frontera et al. [21] 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. [33] give an approach to finding 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. The work most similar to our approach is by Muratov and Zagarskikh [36], 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.

As mentioned in Section 2.1.2, the best approach to date uses the 3D extension of the jump point search (JPS) algorithm [37]. They obtain fast compute times in very large and complex environments. We will show a detailed comparison of this method and our work in Section 6.4.

2.4 Navigation Meshes

Navigation meshes are a way to meaningfully encode the traversable areas (areas outside of obstacles) in an environment. Any tiling, or partition of the environment that conforms to obstacles is a navigation mesh. Having convex tiles is essential 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. The standard method to perform 3D pathfinding in a game environment is to use a form of navigation mesh. Some tools that automatically create navmeshes, like Recast[1], are widely used in video game pathfinding.

Navigation meshes can have many desirable properties and can be compared in a variety of ways [44], such as their coverage of the environment or the number of regions

[1] `https://github.com/recastnavigation/recastnavigation`

used. When using axis-aligned bounding boxes for instance, it is not possible to conform exactly to diagonal obstacles, resulting in some parts of the environment that should be walkable but end up outside of the navigation mesh.

One interesting approach to improve obstacle conformance is proposed by Hale, Youngblood, and Dixit [22] for 2D environments, where rectangles expand from seeds sampled in the level until they encounter an obstacle or another rectangle. This creates an efficient navigation mesh with convex regions. The same approach could be applied to 3D environments, where cubes would "grow" from seeds, with some level conformance as well. Building and growing the 3D structures, however, is complex, and we chose to use octrees, that can also be converted into a navigation mesh.

2.5 Quadtrees and Octrees

Finkel and Bentley [20] first introduced the notion of quadtrees, a hierarchical data structure. A quadtree is created from a square that is recursively split into 4 smaller squares while it contains an obstacle. The recursive splitting stops when a cell is obstacle-free or has reached a smallest allowed size. Quadtrees can be used to perform pathfinding in 2D environments, such as in the work of Hirt et al. [29].

The equivalent of quadtrees in 3 dimensions is octrees, where cubes are generated instead of squares. Fichtner et al. [19] use octrees to model the interior of buildings in the context of indoor navigation in multi-story buildings. A comparison to a point cloud baseline shows that thanks to their hierarchical structure, octrees have fewer points and can be leveraged to identify structures like floors, tables, or stairs.

A highly optimized octree-based navigation is used by Brewer to perform real-time 3D pathfinding in the video game *Warframe* [9] using sparse voxel octrees (SVOs), based on the work of Schwarz and Seidel [40]. In SVOs, the data for each level of the tree is stored in a very compact way using a Morton Code order, which allows to enumerate 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 4x4 voxel grids instead of regular cells as octree leaves when the minimum octree size is reached. This thesis approaches octree navigation differently: these storage and implementation optimizations could be compatible with our work, and we propose a merging method to directly reduce the number of nodes in the octree, leading to faster compute times.

2.6 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 [46] use path pruning to skip a subset of points in the path, by greedily skipping points in the path based on a visibility heuristic. Douglas and Peucker [16] present the recursive Ramer-Douglas-Peucker algorithm that can simplify a polygonal line (in our case the path), in a similar way to path pruning. These two methods can simplify a path and reduce its length, but the resulting path might be in a different homotopic class.

Another form of path refinement is path smoothing, which aims at making the path more natural and less jagged, allowing a vehicle with dynamic constraints and inertia to follow it. Yang and Sukkarieh [46] use rapidly exploring random trees (RRT) then path pruning and Bezier spiral curves to refine the path, mainly to account for turning radius constraints as they are interested in drone (UAV) navigation.

Cimurs and Suh [13] also use Bezier curves but optimize the control points to respect obstacle avoidance and kinematic constraints. Chaikin [12] introduces a fast algorithm to create a smooth curve from a set of points along a polygonal line. This algorithm works iteratively: for a line defined by a list of points, at each step, we sample two points for each segment of the line, more precisely at a 1/4 and 3/4 ratio between the two endpoints. All of these intermediate points form a new and smoother line, and the process can be repeated as much as needed.

When doing path refinement, it can be desirable to stay in the same homotopic class. This can ensure that a set of paths in the same class stay close to each other after smoothing, or that the path does not drastically change after being smoothed. Bhattacharya, Kumar, and Likhachev [7] use the *Cauchy Integral Theorem* to give a characterization of homotopy classes. Hershberger and Snoeyink [27] introduce 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 (SSF) by Mononen [35] which inspired our extension of the funnel algorithm to 3D scenarios. We describe the funnel algorithm in detail in Section 4.3. Erickson [17] uses path reduction and the funnel algorithm to compute the shortest homotopic path to a given path between two points in 2 dimensions.

2.7 3D Pathfinding Benchmarks

To establish a fair comparison between different methods, there is a need for standardized benchmarks. Many pathfinding benchmarks exist for 2D environments, but to our knowledge, there are only three for 3D scenarios. Toll et al. [45] 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 not to environments where free movement in all directions is possible. Since our work is aimed at solving full 3D motion, we will not run experiments on this benchmark, as approaches specially built for this ”2.5D” scenario would be more relevant to this setting and would likely outperform ours.

Brewer and Sturtevant recreated 3D maps from the video game *Warframe* [10], where jetpacks enable full 3D movement. In these very large maps, obstacles are represented by a list of the coordinates of all the voxels occupied by obstacles. These maps tend to contain hundreds of thousands to millions of obstacles and fit in a cube 1000 units long, wide, and high. 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. These maps

mostly consist of ships, debris, and asteroids floating in empty space. For a more diverse set of maps, Nobes et al. [38] followed the same data structure to extend the benchmark with maps presenting a more diverse range of scenarios, motivated by the real-life use cases of 3D pathfinding.

Chapter 3

Environment Decomposition

In this chapter, we will explain how, given a 3-dimensional (3D) map with obstacles, we build a meaningful representation of the environment that we can leverage to perform fast pathfinding. In Section 3.1 we will first introduce the baseline of environment decomposition in 3D, the voxelization. We describe the more efficient octree decomposition in Section 3.2. We give necessary definitions and motivate its use in Section 3.2.1, and give some important details about our implementation in Section 3.2.2. Finally, we explore octree pruning strategies in Section 3.3, which allow us to drastically reduce the number of nodes in an octree, resulting in faster pathfinding times.

3.1 Voxel Baseline

In this section, we will describe the simplest 3D decomposition that we used as a baseline, and introduce several important definitions and concepts that will be useful for this section as well as our octree approach.

The simplest way to perform environment decomposition is to use a voxel grid, voxels being the 3D equivalent of 2D pixels. This can be achieved by dividing the map into evenly sized cubes, called *voxels*. We define as *granularity* the size of these cubes. We take obstacles into account by making every voxel that intersects an obstacle *invalid*. Invalid

voxels cannot be traversed and any path has to go around them, through the remaining *valid* voxels. Such paths are therefore guaranteed to avoid all obstacles.

The similar 2D approach has been used extensively, as it is very simple to implement and it is very fast to create such a grid and remove the invalid pixels. In three dimensions, however, the number of voxels increases drastically. The main limitation of voxelization is its inefficiency in representing large open spaces, which are very common in 3D video game environments, especially ones designed to accommodate full 3D motion. An obstacle-free space that is n units wide, long, and high requires n^3 voxels to represent it at a granularity of 1, whereas as we will see in the next section such an open space can be represented by only 1 octree cell.

Voxel decomposition is easy to implement and fast to compute but results in a very large number of cells. As we will see in Chapter 4, this causes an increase in the number of nodes in the movement graph and therefore in the pathfinding compute times. The octree approach aims at representing the environment more efficiently, with a smaller number of cells, which in turn leads to a smaller graph and faster pathfinding. We give an example of this in Figure 3.1: the octree does not need as many cells to represent the environment, and this will be further enhanced by merging cells together as explained in Section 3.3.

3.2 Octree Decomposition

In this section, we will first give all the necessary definitions for our octree approach and motivate its use. We will then give more details about our implementation of it and some optimizations we added.

3.2.1 Definition and Advantages

To avoid the high cost of voxels we use the more efficient octree data structure. This decomposition is recursive: to build it, we start from a single cubic cell that encompasses

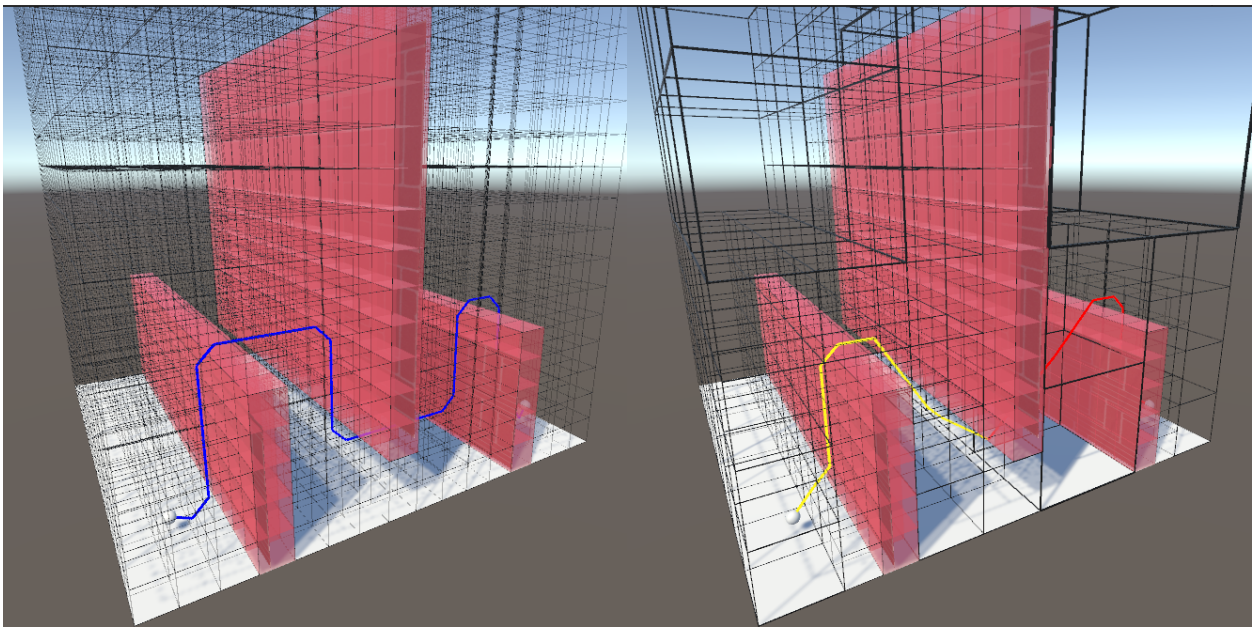

Figure 3.1: Example of a voxel and an octree decomposition on a simple map with three obstacles. Invalid cells are represented as red cubes, and the valid cells as transparent cubes with a black frame.

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 granularity. We split cells in a breadth-first order; that is we split all the cells of the same size that should be split before moving on to the smaller cells.

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.

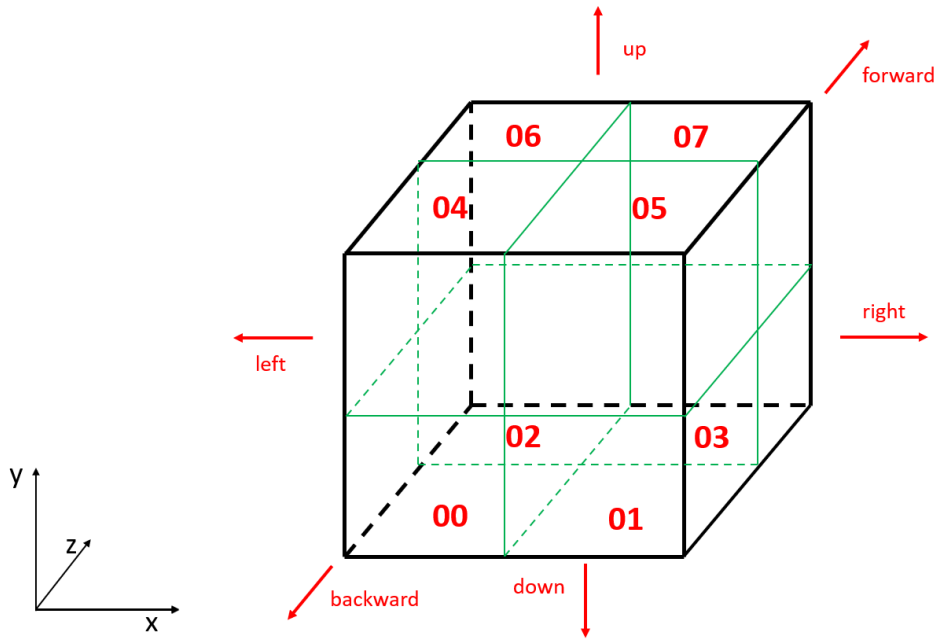

Figure 3.2: Indexing of the 8 children of the root, labeled ”0”. The indexes range from ”00” to ”07”.

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.

3.2.2 Implementation Details

In order to increase the speed of building the octree, several optimizations can be implemented. First of all, we need an efficient indexing method that assigns a unique index to each octree cell. This has many benefits, as we will discuss below. We start the indexing from the root, labeled ”0”, and for every cell we split, we create the index of the children by adding one digit (corresponding to one of the 8 possible child locations) to the parent’s index. We represent this indexing scheme in Figure 3.2 as we split the root, and the same logic is applied to every cell we split.

This indexing scheme offers several advantages. Having a unique index associated with each cell as well as storing cells in a dictionary with their index as the key allows fast access to any cell. By definition, the length of the index of a cell corresponds to its

depth in the octree, which is useful to determine if two cells are at the same depth or not. It also allows us to instantly test the adjacency of two cells simply by comparing their indexes. For instance, cells ”00” and ”01” are adjacent, as shown in Figure 3.2, but cells ”00” and ”07” are not. We give more details about this neighbor check method further below.

To fully benefit from the hierarchical octree structure, we want to perform local operations on the octree as much as possible. To do that, we need to have fast access to the neighbors of a cell, which can be especially useful when building the navigation mesh or pruning the octree as we will see in the next section. For each octree cell, we then store two adjacency lists of the valid and invalid neighbors it has in all six directions (left, right, up, down, forward, and backward). After each operation during the building phase of the octree, we update the affected cells' adjacency list accordingly.

- When splitting a cell into 8, the children will inherit from the neighbors of their parent cell they are still in physical contact with, as well as the other children adjacent to them. The parent's neighbors must also have their adjacency lists updated to reflect the split, which is done by removing the parent and adding the children adjacent to the neighbor.

- When a cell switches from valid to invalid, we remove it from its neighbors' valid adjacency list and add it to the invalid one, and perform similar operations if it switches from invalid to valid.

- We will explain in the next section what happens when we merge two cells.

With this indexing system, it is possible to instantly check if two cells are neighbors or not. If they are siblings, they will have the same index except for the last digit which will be different and indicate their placement in the parent cell. If the last digit of a cell is 1 for instance, it will be a neighbor of the other children with a last digit of 0 or 3, but not with 6.

For a parent index p, two children cells with indexes pi and pj will be neighbors if $i = j \pm 1$ (neighbors along the x axis), $i = j \pm 2$ (neighbors along the z axis), or $i = j \pm 4$ (neighbors along the y axis).

In some cases, we also have to perform neighbor checks between non-siblings cells. When we split a cell, we use the parent's neighbor list to compute the children's one. Given a parent with index p and a child with index pi, the child is still a neighbor with the neighbors of a parent in a given direction (for example to the left) only if the child is on the same (i.e. left) side of the parent cell. These direction checks can be performed as follows:

- pi is on the left side if $i \equiv 0 \mod 2$
- pi is on the right side if $i \equiv 1 \mod 2$
- pi is on the bottom side if $i < 4$
- pi is on the top side if $i >= 4$
- pi is on the backward side if $i \mod 4 < 2$ (i.e. if $i = 0, 1, 4, 5$)
- pi is on the forward side if $i \mod 4 >= 2$ (i.e. if $i = 2, 3, 6, 7$)

This direction check as well as the neighboring check between children of the same cell allows us to create and update the neighbor lists of every cell at every stage in the building of the octree.

Although this update process slightly lengthens the octree building time, this does not affect runtime pathfinding costs, which is the main constraint we face when solving video game pathfinding. Once the octree is built, we save it and can very quickly load it as the game starts for subsequent pathfinding requests. We then have to pay an offline cost during the map generation, but once a level is finished, the octree can simply be loaded with the map and used as it is to solve individual pathfinding queries.

One last thing to discuss is the way collision detection is implemented. Collision detection is used to determine if there is at least one obstacle in a given octree cell, and

thus if we need to split the cell. Since octree cells are already axis-aligned, we chose to treat all obstacles and cells as axis-aligned bounding boxes (AABB) for two reasons. This makes collision testing very fast, as we can just look at the minimum and maximum coordinates of two objects to check if their intersection has a strictly positive volume, meaning they collide. The second reason is that although we do not expect all obstacles to be cubic or rectangular-shaped, it is possible to voxelize any 3D shape to represent it as a collection of small cubes. This is what allows Brewer and Sturtevant [10] to represent complex shapes such as spaceships as a series of voxels. That being said, since collision detection is only used to determine if a cell should be split, it is a modular component that could be replaced in future work by a more advanced method. Such methods could allow for instance representing obstacles as oriented bounding boxes (OBB), or other, more precise bounding volumes, as some of the options explored by Ericson [18].

3.3 Octree Merging

As we will discuss in Chapter 6, 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 [28], which triangulates 2D polygons with holes. 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. As can be seen in Figure 3.3, a cell can have a side larger than the transition, and merging would compromise the convexity of the cell. When two cells are merged, this can create new merging opportunities for neighboring cells, which is handled by our greedy algorithm.

A more practical and equivalent way to check if merging two cells n_1 and n_2 maintains convexity is if n_1 only has one valid neighbor n_2 and no invalid neighbors in a given direction, and the same holds for n_2 when looking in the opposite direction. Since we

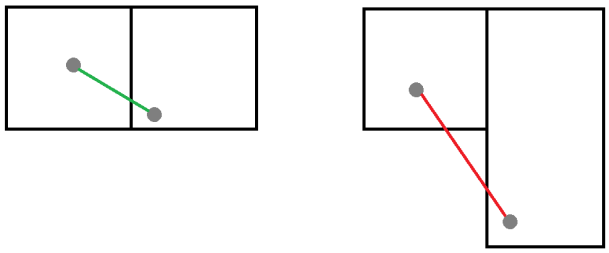

Figure 3.3: Two examples of potential cell merges, the right one being invalid as the convex property is lost.

Algorithm 2 Pseudocode for the octree pruning algorithm

```
Let validStack be a stack of all valid octree leaves
while validStack is not empty do
    Remove the first node n_1 from the stack
    for all 6 cardinal directions dir do
        if n_1 only has 1 valid neighbor n_2 and no invalid neighbors in direction dir, and
        the same holds for n_2 in oppositeDirection(dir) then
            Merge n_1 and n_2, updating their scale, position, and adjacency list accordingly
            Add the merged cell to validStack to check if this merge enabled further merges
        end if
    end for
end while
```

store and update a list of valid and invalid neighbors for each cell, we can check this very easily. We give the pseudocode for our merging procedure in Algorithm 2.

When merging two cells n_1 and n_2, we need to update their adjacency lists. The merged cell's list is simply the union of n_1 and n_2's lists, to which we remove any reference of n_1 or n_2. We also update their neighbor's list by removing n_1 and n_2 if they are present, and adding the merged cell.

This heuristic we use for merging the octree is not guaranteed to be optimal (in terms of the total number of cells after merging), because the order in which cells are merged is random and there might be another order that yields better results. Even in 2D, using a similar procedure does not guarantee optimality, as shown in Figure 3.4. In this example, if we start by merging cells 1 and 2 the best merging that can be obtained leaves 3 quadtree cells, even though a better merging that gives 2 cells exists. In practice, however, this

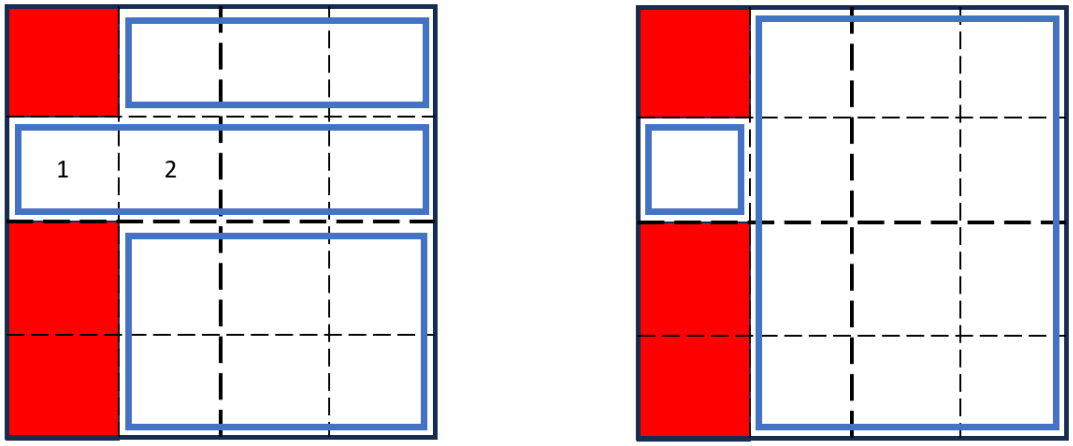

Figure 3.4: An example of imperfect quadtree merging: the original quadtree (dotted lines) is obtained after detecting the obstacles (red squares). If we start by merging cells 1 and 2, at best we obtain 3 quadtree cells (left side), but a better solution with 2 cells exists (right side).

approach gives good results and still consistently leads to a significant decrease in the number of octree cells.

Chapter 4

Path Finding and Path Refinement

In this chapter, we will detail how we can leverage the environment decomposition we built in the last chapter to perform fast pathfinding. If we consider these decompositions as a set of convex connected cells, the only difference between a voxel grid and an octree is the shape and number of such cells, so they can be treated in the same way. In Section 4.1, we explain how we can create a movement graph from the environment decomposition, and how we use it to find short paths. We then explain two path refinement techniques, namely path pruning in Section 4.2 and the 3D funnel algorithm in Section 4.3. These techniques allow us to reduce the length of a path even further while only paying a small compute time when compared to the pathfinding time.

4.1 Path Finding on the Movement Graph

Regardless of the environment decomposition method used, we obtain a set of cells that can be traversed freely. 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 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.

With this movement 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. The A* algorithm is good at solving a point-to-point shortest path problem [47], and many state-of-the-art algorithms are extensions of it.

By using the Euclidean distance as the A* heuristic, we guarantee path optimality within the graph. Note that as the graph is an abstraction of the environment, a shortest path in the graph does not necessarily correspond to a shortest path in the 3D map. Heuristics such as the placement of nodes in the middle of transitions, or the strictly positive granularity make the path sub-optimal. Having a finer granularity or more transition points for each transition surface could decrease slightly the length of paths but at the expense of more compute time. Since our goal is to solve real-time pathfinding in games, this tradeoff is not beneficial; we will explore other ways to reduce path length in Sections 4.2 and 4.3.

One key factor to consider is that the number of nodes in the graph is proportional to the number of cells in the environment decomposition, and the pathfinding cost (in our case the cost of the A* algorithm) increases with the graph size. Because of that, if we want to solve real-time pathfinding in a video game setting, we want to minimize the number of nodes in the environment decomposition.

To find a path from a start point to a target point, we first have to insert them in the graph. To insert a point, we find which octree cell it belongs in, then we add the node to the graph and connect it to all transitions starting from the cell it is in. Once the path is found and we want to connect a new pair of points, it is important to delete the old start and target points and the edges pointing to them in the graph. Inserting a point would require going down the octree: starting from the root, we would have to check for every child if it contains the point, then go one level deeper in this child and repeat the

process until a valid cell is found. Thanks to the hierarchical indexing system detailed in Section 3.2.2, it is instead possible to know directly in which children of a cell the point is, which makes node insertion faster.

One last thing to discuss is the size of the NPC agent. The octree decomposition and pathfinding presented earlier assume the agent is a zero-size dot or an object of small size compared to the obstacles. In many use cases, however, agents can have a non-zero size and should not partially clip through walls. One way to address this problem is during the environment decomposition phase, by virtually expanding all obstacles to account for the size of the agent, following the work of Lozano-Pérez and Wesley [34]. For two sets S and S', the Minkowski difference of S and S' is

$$S \ominus S' = \{x - y | x \in S, y \in S'\}$$

Let O be the subset of $\mathbb{R}^3$ covered by obstacles and A the subset covered by the agent when it is centered at the origin. The problem of detecting collisions between the agent of non-zero size and O is equivalent to the one of detecting collisions between an agent of size 0 and $O \ominus A$. To prove this, let $z \in \mathbb{R}^3$.

$$z \in O \ominus A \iff \exists x \in O, y \in A, z = x - y \iff \exists x \in O, y \in A, z + y = x \iff (z + A) \cap O \neq \varnothing$$

In other words, z is in the expended set of obstacles $O \ominus A$ if and only if the agent intersects O when it is centered in z. During the octree decomposition phase, expanding all obstacles this way then reduces the pathfinding problem to the one with a zero-size agent.

A more advanced way to handle non-zero sized agents is presented by Harabor and Botea [23]. In this octree-based approach, every cell is labeled with its obstacle clearance, defined as the cell's minimal distance to an obstacle. These labels allow handling agents of various sizes without additional computation, as only cells with an obstacle clearance larger than the agent size can be considered valid.

4.2 Path Pruning

As explained in the last section, paths found by the A* algorithm are optimal in length in the movement graph, but can be sub-optimal if we allow free movement in the 3D map itself. A first approach to reduce path length is based on the triangle inequality. For three points u, v and w, $d(u, w) \leq d(u, v) + d(v, w)$ where d is the Euclidean distance. This means that the path going from u to w is always shorter than the one going first from u to v, then from v to w. From this inequality, we gather that if we can skip intermediate waypoints in a path while still avoiding obstacles, the path length will be shortened.

Following the idea of Yang and Sukkarieh [46], we implement a visibility-based heuristic to prune paths. The goal is that given a set of waypoints describing a path, we find a subset of points that gives a shorter path that still avoids obstacles.

We say that two points are *visible* from each other if the segment connecting them does not intersect any obstacles. Starting from the target, we add the last node visible from it and restart the process from this node, as described in Algorithm 3. We illustrate the process (in a 2D environment for better readability but the same principles hold for 3D) and describe the main steps of the algorithm in Figure 4.1. Starting from the original path in step (a), since the second point is visible but the third one is not, we add the former to the pruned path and restart the process with it as the new anchor in step (b). We repeat this process in step (c) and obtain the pruned path in step (d).

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 this algorithm is that unlike the method we will discuss in the next section, the pruned path can be in a different homotopic class. In Figure 4.2, the original path and the pruned path belong in different homotopy classes. If there is a game design motivation to go in between these 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.

Algorithm 3 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
```

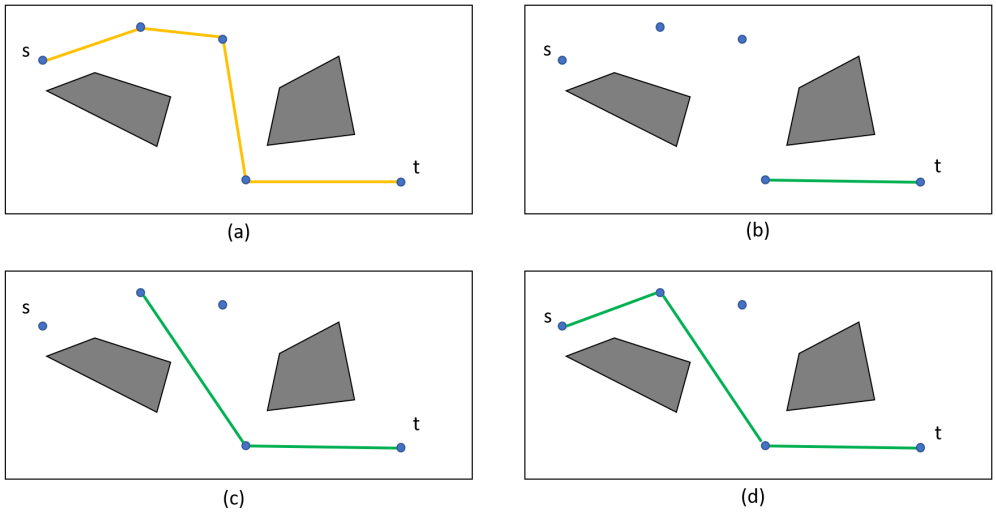

Figure 4.1: An example of path pruning, with the existing path in yellow and the pruned path in green. The original path is shown in step (a). Starting from the target point t, the second point is visible from t but the third is not, so we add the second point to the pruned path in step b. From this new point, the next two points are visible but s is not, so we add the point before s in step c. The resulting pruned path visible in step d skipped a node compared to the original path and is therefore shorter.

4.3 The Funnel Algorithm

We now describe another, more involved form of path refinement, which is a 3D extension of the funnel algorithm, a homotopic 2D path refinement algorithm. In this section, we present one implementation of the base 2D algorithm and explain how this work can be

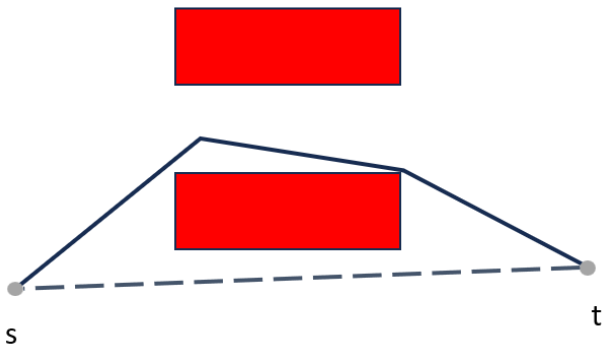

Figure 4.2: Example of the path pruning algorithm changing the homotopy class of a path. The original path (full line) goes in between the obstacles, but the pruned path (dotted line) goes under the obstacles.

extended to a 3D setting. We then give details about some important implementation details.

4.3.1 Presentation of the Algorithm

The funnel algorithm, first introduced by Hershberger and Snoeyink [27], is a 2D path refinement algorithm, which given a path in a 2D environment with obstacles and a triangulation of the environment, finds the shortest homotopic path. In this section, we will explain how this idea can be extended to 3 dimensions.

We recall that the homotopy class of a path P is the set of paths that can be obtained by deforming P without intersecting any obstacles. Thus two paths are in the same homotopy class if and only if one can be continuously deformed to the other without intersecting any obstacles. As illustrated in Figure 4.3, the blue and green paths are in the same class as one can be deformed into the other. The yellow path, however, is in a different class: as it goes on the other side of the obstacle (drawn in red), there is no way to deform it into the blue or green path without going through the obstacle.

In pathfinding applications, there are some advantages to staying in the same homotopic class while doing path refinement. It ensures that the refined path is not drastically different from the original path. For instance, if the original path goes a

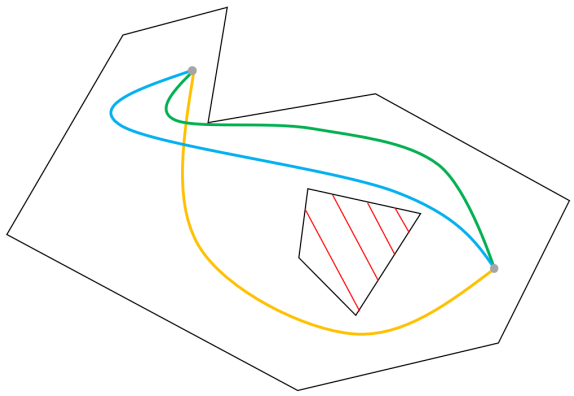

Figure 4.3: Illustration of homotopy classes: the blue and green paths are in the same homotopy class, but the yellow one is in a different class as it goes on the other side of the obstacle.

certain way to stay close to a beneficial resource or avoid an enemy, staying in the same homotopy class ensures this property is maintained. Being able to compute the different homotopic classes is also relevant for visualization purposes, to illustrate the different general routes that connect two points.

The SSF (simple stupid funnel) algorithm [35] is a simple implementation of the funnel algorithm in 2D. It takes a path between two points in a 2D environment as input and returns the shortest path in the same homotopic class. It uses a navigation mesh (e.g., triangulation) and works on the portals (edges) between adjacent partitions (triangles) along the input path. Figure 4.4 illustrates the way the algorithm works. The intuition is that, for a path that goes through a series of portals (edges in the 2D case), there will be 2 funnel "arms" going from the starting (anchor) point.

The funnel arms are incrementally moved to each vertex of the next portal. For each portal, if the funnel shrinks (as in step b), the arms' angles are updated. If a funnel arm would broaden the funnel by going to the new portal, it is not updated. If there is a bend that forces one funnel arm to cross the other one (such as the orange arm in step d crossing the blue arm), we add the corner of the previous portal (the extremity of the other funnel arm) to the refined path and restart the process from this new anchor point.

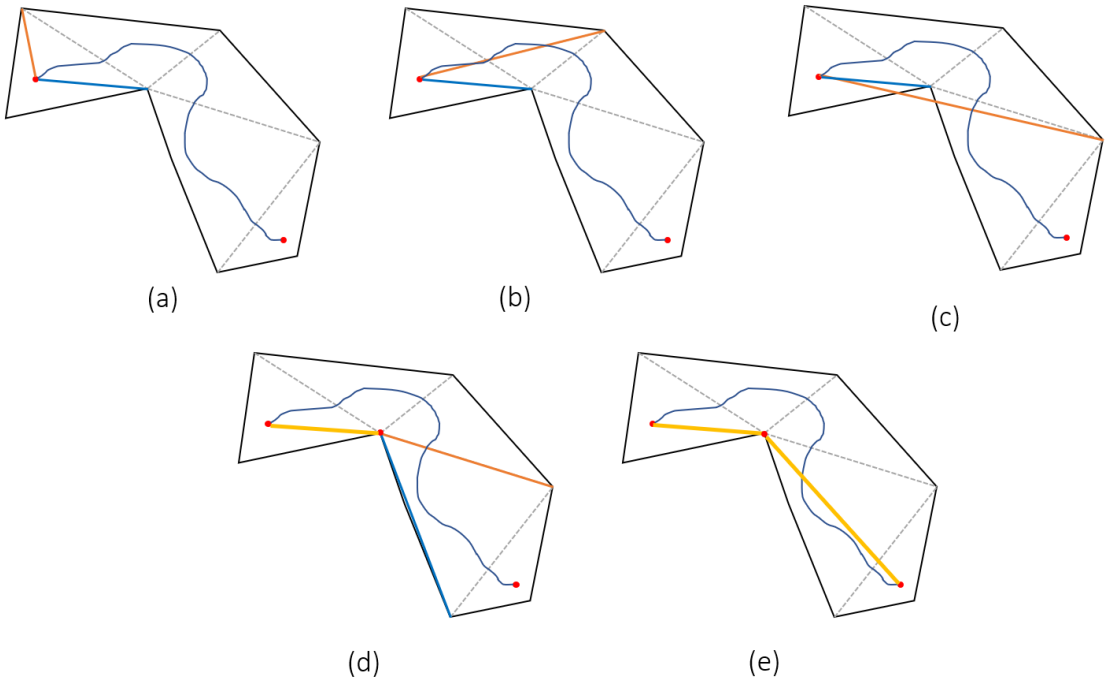

Figure 4.4: Illustration of the first steps of the 2D funnel algorithm. The funnel arms (the blue and orange lines) are initialized to the first transition (dashed gray line) in step (a). The funnel shrinks as we go through the next transition in step (b). In step (c) the arms cross each other, making us restart from a new point in step (d). We complete the refined path in step (e), as we reach the triangle the target is in.

To extend this algorithm to 3 dimensions, we need to shift perspective. Another way to think of this algorithm is to see the funnel as a 2D triangle originating from the anchor point. For each transition, we intersect the triangle formed by the edge and the anchor point with the existing funnel. The funnel can only shrink, and if there is an edge so that the triangle associated with it does not intersect the funnel, we restart the process from the point of the previous edge closest to the current edge.

This idea can be extended to 3D, where the portals are the rectangular surfaces between adjacent octree cells, and the funnel is a polyhedral cone volume (instead of a triangle) originating from the anchor point. This cone can be seen as the field of view (FOV) of available space for the path when viewing it from the anchor point.

We recall that points in the path correspond to the center of transitions (portals) between 2 adjacent cells. For a rectangular transition T defined by the list of its vertices x_1, x_2, x_3, x_4, we write $\mathcal{C}(anchor, T)$ the infinite cone starting from the anchor point and intersecting said vertices. This forms a pyramidal cone, but in general the FOV can be a cone formed by the anchor point and any polygonal shape.

The funnel algorithm is described in Algorithm 4. We iterate over the successive transitions the path goes through, and narrow the FOV progressively until it no longer intersects the cone formed by the anchor and a transition, at which point we restart the process from a new anchor point. The target point t can also be seen as the last transition of the path, with the exception that it has an area of 0.

This "ideal" behavior of the algorithm is illustrated in Figure 4.5. As we will see later, numerous corner cases can arise and must be treated separately. If the intersection is empty, we must select a new anchor point within the intersection of the FOV and the previous transition. To do so, we use the heuristic that it should minimize d(old_anchor, new_anchor) + d(new_anchor, next_transition), where d is the Euclidean distance, and the distance from a point to the next transition is defined as the distance between that point and the closest point to it in the transition: $d(x, T) = \min_{y \in T} d(x, y)$. This heuristic minimizes the length of the sub-path (old_anchor, new_anchor, next_transition), and provides good results in practice.

To better understand how the funnel algorithm works, we illustrate the first steps of the algorithm viewed from the starting point in a simple case in Figure 4.6. In this Unity3D scene we are looking up (the white dot near the center is the sun), and the path, represented as a red line, has to bend over a wall in the bottom right corner. This path was computed using an octree, as explained in Section 4.1. The path crosses the yellow transition in Step 1, and the FOV is initialized. In Steps 2, 3, and 4, we intersect the FOV with the cyan, blue, and grey transitions, and the FOV (represented in black) shrinks at every step. In Step 5, the path crosses the white transition, but the cone delimited by this transition and the anchor does not intersect the FOV. The algorithm then restarts at a new

Algorithm 4 Pseudocode for the funnel algorithm in a 3D environment

Input: A path between two points s and t and the associated transitions $T_0, T_1, ..., T_k = t$
Output: The refined path

```
Let anchor = s, FOV = C(s, T_0), and refinedPath = s
for 1 ≤ i ≤ k do
    Compute the intersection of the current FOV and C(anchor, T_i)
    if the intersection is empty then
        Let anchor be the point x in FOV ∩ T_{i-1} that minimizes the distance D(anchor, x) +
        D(x, T_i), and FOV = C(x, T_i)
        Add the new anchor to refinedPath
    else
        Let FOV be the computed intersection
    end if
end for
Add t to refinedPath
return refinedPath
```

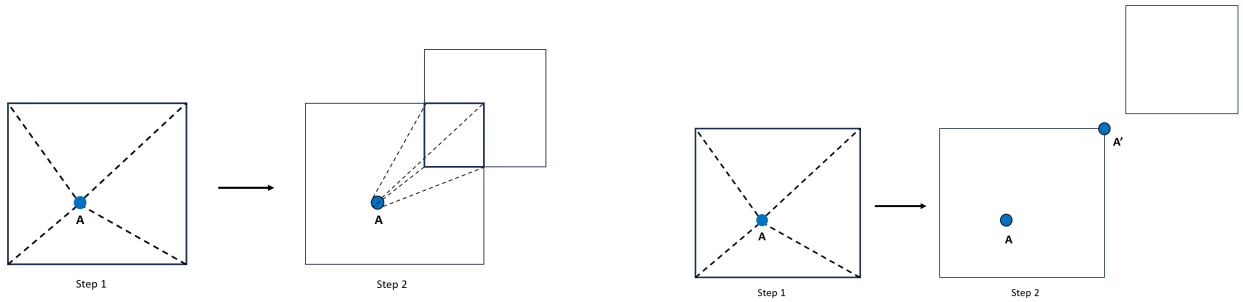

Figure 4.5: Two possible steps of the 3D funnel algorithm. The anchor point is the blue dot A, and the black squares are transitions. On the left side, the FOV (dotted lines) intersects the next transition and shrinks. On the right side, there is no intersection, so we restart the process from a new anchor point, A', that minimizes path distance.

anchor point (the green dot on the figure), which will be on the previous transition (the gray one). We choose the point x to restart from in the intersection of the FOV and T_{gray} so that it minimizes the distance $D(anchor, x) + D(x, T_{\text{white}})$.

With this algorithm, we can find the shortest path within the series of octree cells the original path goes through, even if sometimes a faster path going through another set of cells might exist in the same homotopy class. This method thus does not strictly

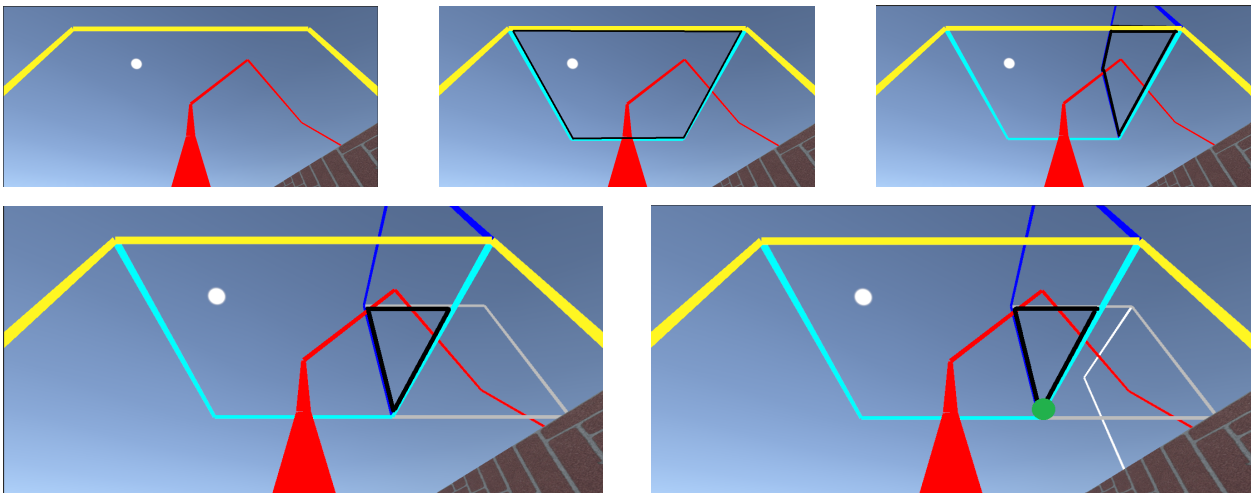

Figure 4.6: Perspective view (from the anchor point) of the first 5 steps of the funnel algorithm. The original path is in red, the successive transitions are represented as colored squares and the FOV as a black polygon.

guarantee path optimality within the homotopic class, but we will see in Chapter 6 that it still noticeably decreases path length.

Although the funnel algorithm attempts to find the shortest homotopic path, the overall optimality of that path is limited by both how well the octree fits the terrain as well as the relative optimality of the sequence of octree cells given by the A* search of the movement graph (the A* search is based on portal-center to portal-center distances). Path pruning can thus still be applied after the funnel algorithm to further improve the path, as pruning takes a subset of the path positions and the funnel algorithm improves the overall quality of these positions.

4.3.2 Implementation Details

In this section we will first discuss how we efficiently compute the intersection of two 3D cones at each step of the funnel algorithm, then illustrate some of the corner cases that can arise and how we treat them.

The FOV and the transitions can be represented as a list of 3-dimensional vectors, which is the list of the corners of the polygon. This list and the anchor point generate a 3D

Algorithm 5 Pseudocode for computing the intersection of 2 convex polygons

Input: Two convex polygons P and P'
Output: The intersection P'' of the two polygons

```
Let P'' = {}
for all edges (u, v) in P do
    Let H be the half plane delimited by (u, v) where P is in
    for all edges (u', v') in P' do
        Let x be the intersection (if it exists) of the segment [u', v'] and the line (u, v)
        if u' ∈ H and v' ∈ H then
            Add u' to P''
        else if u' ∈ H and v' ∉ H then
            Add u' and x to P''
        else if u' ∉ H and v' ∈ H then
            Add x to P''
        else if u' ∉ H and v' ∉ H then
            Do not add anything to P''
        end if
    end for
end for
return P''
```

cone extending from the anchor. For computing intersections we also store "normalized" versions of the FOV and transitions: for a point x and the anchor a, the normalized point $\bar{x}$ is

$$\bar{x} = \frac{x - a}{\|x - a\|} - a \in \mathcal{S}(0, 1).$$

These normalized points all belong on $\mathcal{S}(0, 1)$ (the surface of the sphere of center 0 and radius 1). If we consider the anchor as the origin, they correspond to the angle at which we see the points from the anchor, which justifies our use of the term "field of view".

By normalizing the FOV and transitions, we essentially simplify 3D volume computations to computations on the surface of a sphere, which allows us to use 2D methods. We explain the computation of the intersection of 2 convex polygons in Algorithm 5. We iterate over all edges of the first polygon P and intersect the second polygon P' with the half-plane delimited by the edge and that P belongs in (this is done in the inner loop). By successively doing this for all edges of P, we obtain the intersection of P and P'.

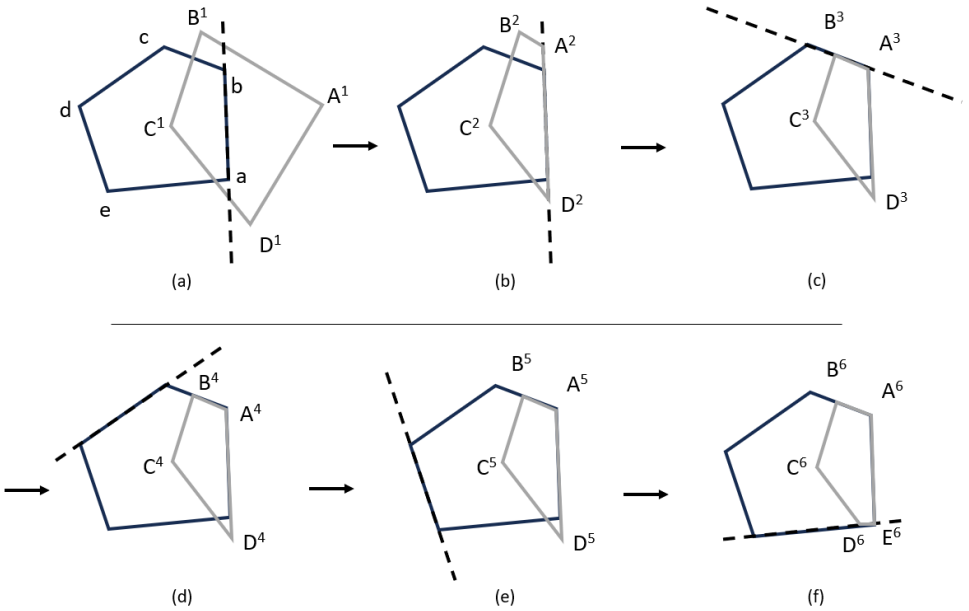

Figure 4.7: Computing the intersection of $P = (a, b, c, d, e)$ and $P' = (A^1, B^1, C^1, D^1)$ (drawn in grey). We start by intersecting P' with the half-plane delimited by (a, b) containing P (drawn in black) in steps (a) and (b), and repeat the process with the over edges of P in steps (c) to (f).

The intersection process for two polygons P and P' is illustrated in Figure 4.7. We first compute the intersection of P' and the half-plane defined by (A, B) in steps (a) and (b). Starting from (A', B'), since $A' \notin H$ and $B' \in H$, we add the intersection of (A, B) and $[A', B']$, A'', to P''. For the next edge, since $B' \in H$ and $C' \in H$, we add B'. Next, $C' \in H$ and $D' \notin H$ so we add both C' and D'' to P''. For the last edge, since neither D' nor A' are in H, we do nothing. Repeating the same process for all the half-planes defined by the edges of P in steps (c) to (f), we obtain the intersection of P and P'.

This algorithm for 2D convex intersection is useful since all switch cases are based on points being inside or outside hyperplanes. As explained earlier, we deal with convex polygons on the surface of a sphere, so even though points are constrained to a 2D space they reside in the 3D space. Hyperplane tests still work in this scenario, and allow us to simplify the problem of intersecting two 3D cones originating from the anchor to the one of intersecting their polygonal bases in a 2D setting.

Due to the grid structure of the octree, corner cases arrive often. We recall that the FOV as well as the cone defined by a transition are usually defined by the anchor point and 4 points forming a rectangular base, but we can account for more complex polygonal bases with more than 4 points, or on the opposite have the ”cone” base collapse to a segment with 2 points, or even a single point, in which case the cone is reduced to a line. We will now detail the most common corner cases that can happen and how they are handled:

- If the anchor point is in the same plane as a rectangular transition, from the point of view of the anchor the transition will be seen from the side and appear as a line instead of a rectangle. In this case, the cone defined by the transition collapses to a triangle, and we compute its intersection with the FOV by performing a polygon/segment intersection of the two bases.

- After several steps and intersections, the base of the FOV can be reduced to a line or even a single point. If it is a line, we compute a line/polygon intersection with the next transition, as explained in the point above. If it is a point, the FOV is reduced to a straight line. In this case, the next transition intersects the FOV if the FOV line goes through it, and we simply check if the point appears inside of the transition when viewed from the anchor (point/polygon intersection).

- The last transition in our implementation of the funnel algorithm is the target, which we see as a transition defined by only one point, hence the cone associated with it is a line. We handle this as explained above by doing a point/polygon intersection.

To summarize, even though various corner cases can happen, they can all be handled by not doing the usual polygon/polygon intersection but doing segment/polygon, point/polygon, or even segment/segment or segment/point intersection tests if multiple corner cases happen at the same time.

Chapter 5

Dynamic 3D Path Finding

In this chapter, we will describe how the environment decomposition and pathfinding techniques can be adapted to handle dynamic environments. The two main components we use for pathfinding are the octree and the associated movement graph. One of the main advantages of this technique is that the octree can be pre-computed, and loaded quickly during run-time, which makes real-time pathfinding feasible. While this works well for static environments, if the map is slightly updated or if obstacles move during the execution the octree no longer reflects the environment faithfully. Many games can benefit from dynamic environments, with changes as diverse as:

- Relatively small obstacles may be moving in the otherwise static level, such as asteroids floating in space.
- Enemies or guards patrolling through the level, that can be modeled as obstacles with a certain sphere of influence (their range) corresponding to the obstacle size.
- Drastic changes in the level layout, such as an explosion creating holes in a wall that allow new paths.

We will now describe in detail how we can dynamically update the octree that we built earlier to reflect these changes. In Section 5.1, we will explain the general update process for the octree. Updating a merged octree is more challenging, as in merged octrees cells

no longer always have 8 children, and the hierarchy starts to lose its meaning. Although we have to view the merged octree in a different way to update it dynamically, it is still possible to do so as we will show in Section 5.2. We tackle the second part of the update process, the graph update, in Section 5.3, and discuss some potential use cases of dynamic octrees in Section 5.4.

5.1 Updating the Octree

In the entirety of this chapter, we assume that the octree and associated graph were pre-computed, and we will describe how we can update them dynamically. Since the octree is a hierarchical structure, most of the time a 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 2 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. In this case, it may be possible to merge octree cells that were previously split.

One very important problem to address is pruning the octree as we update it. If a small obstacle enters a large cell, thus making it invalid, we want to split the large cell as only a fraction of it is actually invalid, and splitting it allows most of the cell to remain valid and traversable. Because of that, the number of nodes quickly increases in a dynamic environment, as large cells are repeatedly split. After a long time, the octree would lose its edge against the voxel baseline, as all the larger cells would be split.

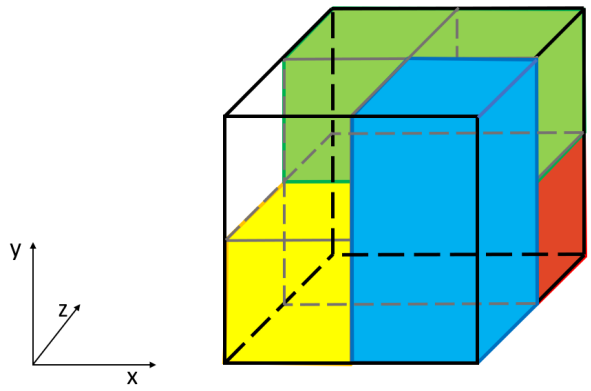

Figure 5.1: 3D drawing of an imperfect dynamic merging caused by an obstacle (red cube) enters the cell in $(x = 1, y = 0, z = 1)$. The cell must be split, and one possible result of greedy merging is the three rectangular-shaped cells (blue, yellow, and green cells). The last cell in $(0, 1, 0)$ cannot be merged. In this scenario, cells cannot be merged anymore without losing convexity, and this remains true even if the obstacle leaves the cell. The parent cell thus remains split in 5 even after the obstacle is gone.

To counteract this splitting, we must implement a procedure that merges cells that become valid back together. We first experimented with reusing the greedy merging procedure detailed in Section 3.3. When an obstacle moves out of a cell and it becomes valid, it can often be merged with its valid neighbors to form a larger cell. Unfortunately, even after trying various heuristics for the greedy merging, this did not give satisfactory results. Because the merging does not follow a specific rule, greedy merging was unable to restore cells to their original structure. We give an example of this in Figure 5.1: as an obstacle causes a split in the octree, siblings can be merged together imperfectly in the split cell. Even if the obstacle leaves the cell, it may not be able to be merged back to its original state, as in this instance there is no way to merge cells two by two while maintaining convexity.

If an obstacle goes through a large cell, causing it to be split, and then exits it, greedy merging would reduce the number of nodes after the obstacle exited the cell but be unable to restore the one large cell. There remain some "artifacts" of the passage of the obstacle. When looking at the octree as a whole, as obstacles move around, the number of valid cells

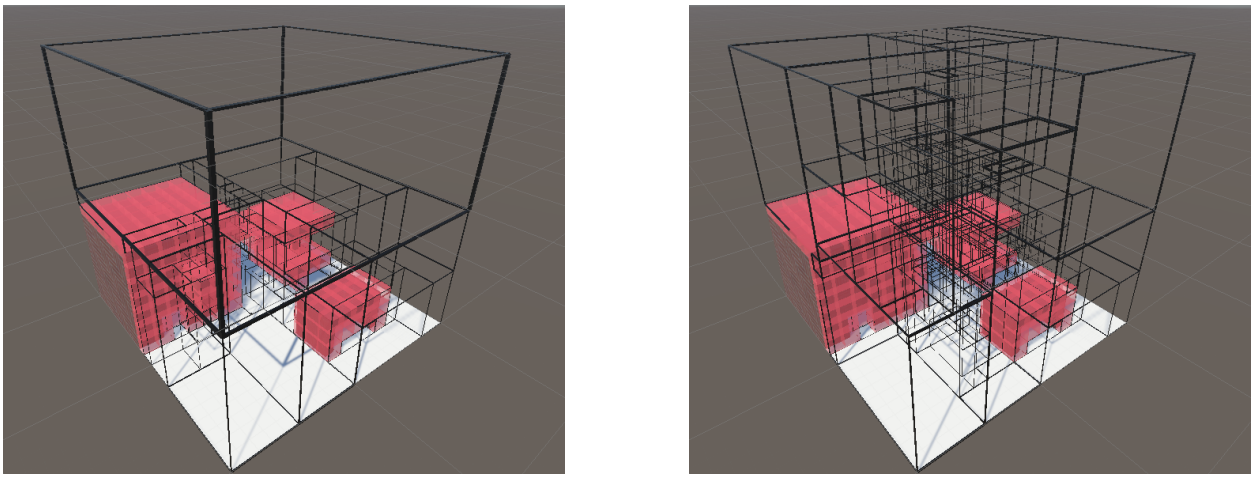

Figure 5.2: Illustration of ”artifacts” appearing in the octree. After the initial computation (left side), if the octree is updated dynamically after a while cells are inefficiently merged, and their total number increases.

should remain approximately constant, but because of these artifacts not all splits can be repaired and this number keeps increasing as time passes. As illustrated in Figure 5.2, where an obstacle was moving freely through the level for some time, the original octree structure including the very large open cell at the top of the level is quickly lost, resulting in a larger number of cells and a less efficient octree.

A second approach that yielded much better results was repairing cells. As a cell is updated and becomes valid, if all of its ”siblings” (the children of the cell’s parent) are valid, it is possible to repair the split parent by merging the 8 valid children together. If the parent’s siblings are also all valid, we can in turn repair its parent, and so on, going as high as possible in the hierarchy. This solves the previous problem of one small obstacle entering and then leaving a large cell: no matter how much it was split, after the obstacle has exited the cell, all of the split cells can be repaired until the large cell is restored. The scenario presented in Figure 5.1 can no longer happen: instead in this example the obstacle would cause the split and the 7 valid cells would not be merged, but as soon as the obstacle leaves the cell the parent cell is restored.

During all the dynamic operations, we also update the adjacency lists of all the affected cells in the same way that we did during the building of the octree. For example, when restoring a parent by merging its 8 children, the parent's list is the set union of all the children's lists.

We will compare these two update policies in Section 6.5, and Figure 6.13 will confirm experimentally a gradual increase in the number of valid cells in the octree when using the greedy merging update method. On the other hand, repairing parent cells when all of their children are valid leads to a stable number of cells in the octree. We therefore choose this repair policy to update dynamically the octree.

5.2 Updating the Merged Octree

In this section, we will explain in greater detail the differences between a regular and a merged octree and why updating it dynamically requires more attention. First of all, as we will see in Chapter 6, a merged octree contains significantly fewer cells and enables faster pathfinding than a regular one. Because of this, in a real-time game setting, we recommend the use of merged octrees. It is therefore important to describe the subtle differences in the update procedure. Note that the general update idea still holds - keeping track of the position of dynamic obstacles, changing cells from valid to invalid or vice-versa, splitting them, and merging them.

The main differences that merging an octree as detailed in Section 3.3 makes is that the octree hierarchy loses meaning and that cells are not cubic anymore. As cells are merged, parents no longer necessarily have 8 children, and we lose the relationship between the depth of a cell and its size. If a small cell is merged multiple times with its neighbors for instance, it will end up being a large cell even though it can be deep in the tree. On top of that, cells may not be cubic and can end up being rectangular. For these reasons, we can no longer use the hierarchical indexing described in Section 3.2.2, and can no longer assume the fact that a parent has exactly 8 children. We can thus no longer test the

adjacency of two cells simply by looking at their hierarchical index, we instead have to test if the two cells share a contact surface of strictly positive area (as in the case where they share just an edge or a corner they are not considered neighbors).

In this setting, it makes more sense to view the finalized octree (after building but before any dynamical updates) as a set of convex cells covering the space rather than a hierarchical structure. To update it dynamically, we can then split these cells or restore them, as we did in the last section, although the splitting procedure is slightly different. Because we save the original octree and split/repair cells according to this original structure, the octree remains stable with updates, and its general shape does not change too drastically even after many changes have occurred.

Merged octree cells are not necessarily cubic, but thanks to the minimum cell size (granularity), they can be decomposed into cubes of the smallest size. In other words, if we assume that the granularity is 1, cells in a merged octree can be rectangular, but their length, width, and height will be integers (multiples of 1). With this in mind, when dynamical updates cause a split in elongated cells, splitting them in 8 could create increasingly elongated cells and odd shapes. Instead, if a rectangular cell must be split, we split it into evenly-sized cubes, of size the greatest common divisor (GCD) of the parent cell's length, height, and width. This minimizes the number of cubes created by splitting the cell. For example, with that granularity of 1, if a rectangular cell of size 4x1x1 must be split, this allows us to split it into 4 1x1x1 cubes instead of 8 2x0.5x0.5 rectangles, and not have children with one of their dimensions smaller than the granularity.

5.3 Updating the Navigation Graph

After the octree is updated, we must reflect these changes in the movement graph to finalize the update process. One very simple 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 with a large number of octree cells, 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 actually 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 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 6.5.

5.4 Potential Use Cases

We explained in detail the dynamic update process in the last three sections, let us now discuss some of its potential applications. After giving some context, we will discuss

a variety of settings where dynamic pathfinding would be interesting and link them to potential game scenarios. We then explain how dynamic octrees can be used for path replanning, and propose compromises that can be made to improve performance at runtime.

While this method can work in any dynamic setting, the frequency and compute time of updates will increase with the number of moving obstacles, as well as the time needed to track the movement of all the obstacles and detect changes that they would cause in the octree. This limitation on the number of dynamic obstacles that can be tracked can be interpreted in two ways:

- If we aim at modeling a full 3D dynamic environment where every part of the environment can move, there must be a compromise between the number of obstacles and the compute time and frequency desired for dynamical updates.

- On the other hand, this compromise can be exploited by creating a large environment where most of the obstacles are static, and only a fraction of them are moving. By only considering a reduced number of obstacles for dynamical updates, such scenarios would allow for dynamic elements even in large-scale maps.

Following this second idea, having a small number of moving obstacles in a mostly static level makes sense from a game design perspective. These moving obstacles could be physical obstacles moving around, or enemies that the agent should keep away from.

These moving obstacles can behave in many ways, in this paragraph we will give a few examples. These examples will be useful for designing test cases to evaluate the behavior and performance of our work, and linking them to potential game applications helps provide more context. Obstacles can have a linear motion with a random but constant velocity, mimicking asteroids floating in space for instance, and can disappear as they exit the map or bounce on its edges.

Although our current implementation does not allow to easily separate static and dynamic obstacles, in the future we aim at clearly differentiating them. This would allow

us to scale our approach to very large maps with a few moving obstacles. A prime setting to do so would be adding obstacles with a constant velocity to the *Warframe* maps to simulate floating ship debris and asteroids, as these maps take place in space and involve flying.

To imitate flying enemies, such as the flying enemies present in *Elex 2*, we can also have obstacles behave with erratic movement by going to a random position in the level and orbiting for a while around it before going to another position. This creates unpredictable movement for the player where enemies patrol by going to a zone, searching around this zone and moving on. In order to recreate more realistic movement in the level, it is also possible for enemies to move by following the octree's navigation graph and the path it gives between two locations. This creates more interesting enemy movement, where they move through the parts of the level a player would go through.

Dynamic octrees can also be used for path replanning. From the perspective of the moving agent (that can be an NPC companion or a help to the player that shows him the recommended path), dynamical pathfinding can be used to maintain a path between two fixed points in a dynamic environment, which can be useful if traveling along this path is instantaneous or the environment does not change often. A more useful application is that of computing a path, moving along it, and rerouting if an obstacle gets in the way. Before the agent starts moving on the path, we compute the list of cells that the path goes through and their distance along the path. As the agent starts moving along the path, if an obstacle intersects a cell ahead of the agent, we update the octree and movement graph, and recompute the path with the new starting point being the present position of the agent. In doing so, the agent will deviate from its initial path and successfully avoid the obstacle. This "just-in-time" approach is commonly used as a replanning method in game contexts to handle dynamic changes in agent planning.

To conclude this chapter, we propose a few compromises that can be made to this approach to improve performance. In order to improve the performance at runtime, we want to reduce the number of updates that are made. This can be achieved in a variety of

ways. First of all, since it is a safe assumption that obstacles should move continuously at a slow or medium speed, any given obstacle will only move slightly between two frames. For this reason, keeping track of the obstacles' movement and updating the octree accordingly can be made only once every few frames instead of every frame. This naturally creates a compromise between the update time cost (averaged over time) and the "reactivity" to changes to the environment.

Another consideration that can be made is that, in any large level, if we compute and want to update a path between two points, most of the changes to the environment will occur in regions that are very far from the path, and will not affect the path in any way thanks to the hierarchical nature of the octree. A second compromise that can be made is that of updating the octree only when it is necessary, meaning when an obstacle moves in the way of the previously computed path. To do so, we keep track of the potential changes that should be made, such as the cells that should become valid or invalid, and only when one of the cells the path goes through is affected, we apply all of these changes. The advantage of this method is that most changes will cancel each other out over time: for example, if an obstacle enters a cell far from the path, the cell should become invalid, but after the obstacle has exited the cell, this change is not necessary anymore. With this "last moment" update method, even if the octree has not been updated for a long time, the number of changes that will actually be made when we update it remains reasonable. This makes updates a little bit more expensive than updating the octree as soon as any part of it changes, but since updates occur far less frequently the update cost smoothed over time is greatly reduced.

Chapter 6

Experiments and Results

In this chapter, we will evaluate our work in a variety of scenarios and assess the impact of every refinement we made to the basic octree algorithm. The general approach we used to evaluate 3D pathfinding algorithms is the same throughout this chapter. We follow the same idea as existing 3D pathfinding benchmarks ([10], [38]): we create several 3D environments, and for each one of these maps generate a large number of ”test scenarios” by randomly sampling pairs of points in the walkable space of the level. These scenarios consist of a start and a target point, and the task is finding the shortest path between them. By testing different algorithms on the same scenarios, it is possible to compare their performance.

We are mainly interested in 2 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 [11] 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. Games can ”cheat” and hide some compute time in start-up animations for example, but pathfinding costs should not exceed a second, and ideally should remain in the hundreds or tens of milliseconds.

We will first give visualizations of some important aspects of our work in Section 6.1. We include a brief presentation of the different scenarios we will test our work on in Section 6.2. In Section 6.3, we will compare the voxel baseline, the regular octree, and the merged octree on a variety of small, handmade maps. This comparison will establish that merged octrees are by far the fastest pathfinding method of the three. To provide results on more difficult scenarios, we use the more challenging *Warframe* dataset [10] and its extension [38], and report our findings in Section 6.4. These datasets feature much larger maps and complex obstacles in a variety of settings that can be linked to real-world applications. We also include a comparison of our work and what we believe is the state-of-the-art 3D pathfinding method by Nobes et al. [37]. Finally, we will evaluate the performance of our dynamic octree update method in Section 6.5.

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). This setting is interesting as it corresponds to the real use case for our work. Unity is one of the largest game engines in the world, and since people tend to play games on a personal machine rather than a supercomputer, our results show what the expected performance in an actual video game would look like.

6.1 Visualisation of the Results

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 handmade map can be found in Figure 6.1. It is clear that pruning the octree greatly reduces the number of cells.

The number of valid cells in the environment decomposition 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

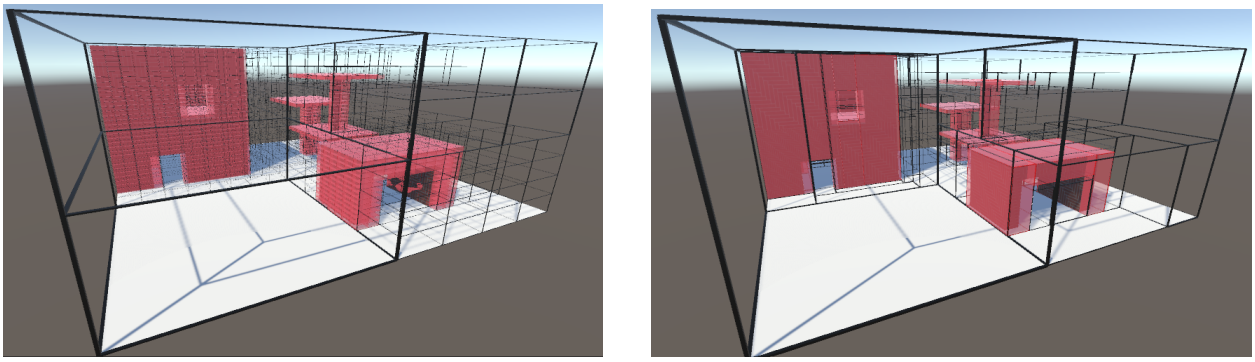

Figure 6.1: Example of an Octree decomposition before (left side) and after (right side) merging. Invalid cells are represented in red, and the black frames indicate the location of valid octree leaves.

Table 6.1: Comparison of the number of valid cells across several maps using either the voxel baseline, the regular, or merged octree.

Map name	Voxel	Unmerged octree	Merged octree
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**

cells on several maps in Table 6.1. On the small hand-made maps (namely the variants of Building, Cave, Industrial, and Zigzag), we can see that even the unmerged octree is able to represent the environment using far fewer nodes than the voxel baseline, since it only needs a few cells to represent large open spaces. Our merging strategy further reduces the number of valid cells in the octree. We also included the theoretical number of voxels in the smallest map from the *Warframe* dataset, the *Complex* map. It would require more than 8 million voxels, resulting in a huge graph and impractical pathfinding compute times. The octree and especially the merged version on the other hand only need a relatively small number of cells. For this reason, for all of our experiments on larger maps, we only use merged octrees.

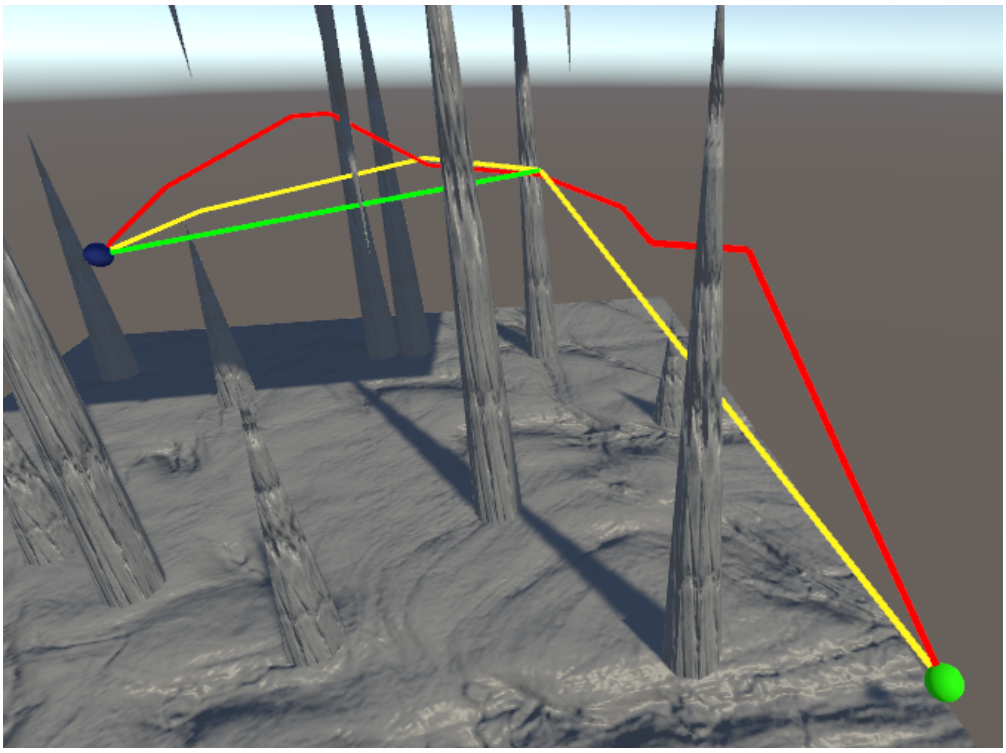

Figure 6.2: Shortest path between two points found using the octree structure (red), after applying the funnel algorithm (yellow), and after funnel and path pruning (green).

We illustrate how path refinement works in Figure 6.2. In this example, we use an octree to find a path between the start (green sphere) and the target point (blue point) 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.

6.2 Overview of the 3D Benchmarks

For the rest of this chapter we perform experiments on three datasets: a handmade dataset comprising smaller maps representing diverse scenarios, the *Warframe* dataset published by Brewer and Sturtevant [10], and the newer 3D benchmark by Nobes et al. [38].

We started by designing different test scenarios to test our pathfinding algorithms, we provide snapshots of these maps in Figure 6.3. The map Zigzag consists of 3 obstacles that test the ability to weave between obstacles. Building_1, Building_2, and Building_3 are multi-story buildings with holes in the walls and floors that create interesting traversal possibilities. Cave and Industrial are maps that have complex structures that a game with full 3-dimensional movement could have, namely a cave interior with protruding rock formations and an industrial complex with buildings, a bridge, and other complex structures. All of these maps are much smaller than the other datasets, but allow us to compare the voxel baseline and the unmerged and merged octree in simple scenarios.

For each of these maps, we sampled pairs of start/target points to test our pathfinding algorithm on. After discarding pairs of points that could not be connected by any path (this can happen if a point is inside an obstacle or if the two points are unreachable from each other), we obtain 10,000 pairs of points. 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.

The *Warframe* dataset [10] is a recreation of maps from the space combat game *Warframe*, published by Digital Extremes. Each map is represented as a list of occupied voxels (obstacles) in a 3D grid. These maps are much larger, ranging from a few hundred to more than a thousand units long, high, and wide, and can contain hundreds of thousands or millions of occupied voxels. These maps are designed by a series of letters (e.g., A, BA, BB, etc.) and a number. Maps with the same letter and different numbers are very similar to each other, and therefore due to the long compute times to run experiments on a single map, we will only perform experiments on one map of each series, resulting

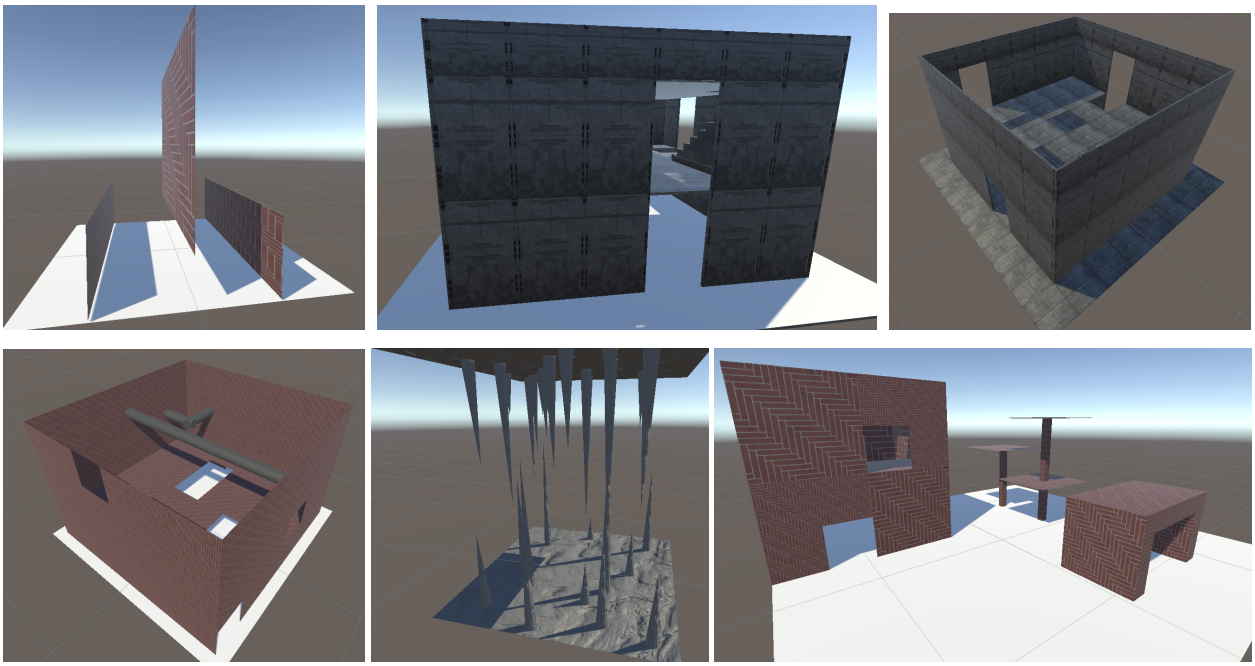

Figure 6.3: Snapshot of the different scenarios, from top left to bottom right: Zigzag, Building_1, Building_2, Building_3, Cave, Industrial.

in a total of 19 maps and 190,000 pathfinding instances. We provide a snapshot of some of these maps in Figure 6.4. Most maps consist of asteroids and debris scattered over a large open space, with two notable exceptions. The *Complex* map is one large and hollow broken spaceship, and is therefore interesting thanks to its different structure, but is smaller than the other maps, resulting in faster compute times. The *C* named maps on the other hand are much larger than other maps, and are always omitted in other work due to their size and unpractical compute times.

Each map comes with 10,000 pairs of points. These points are sampled to be very close to obstacles (their distance to the nearest obstacle is always less than 5). This forces interesting paths, because if they were sampled randomly, considering maps contain very large empty spaces, in most cases they could be connected by a straight line, resulting in trivial solutions.

Following the same methodology as for the *Warframe* dataset, Nobes et al. created another 3D dataset [38], motivated by potential pathfinding applications. The *Descent*

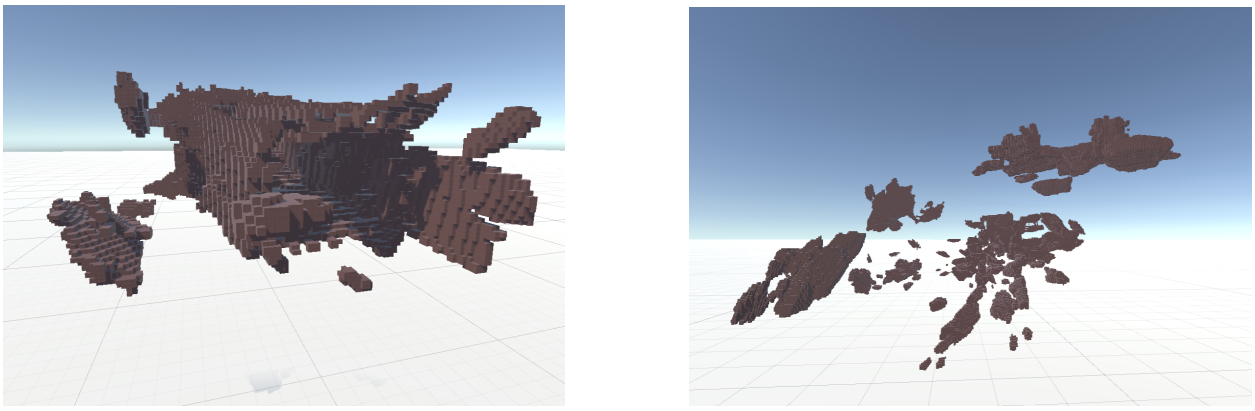

Figure 6.4: Snapshot of some *Warframe* maps. Some, like *Complex* (left side), feature broken, hollow spaceships, while others like *Full4* (right side) consist of many asteroids and debris floating in space.

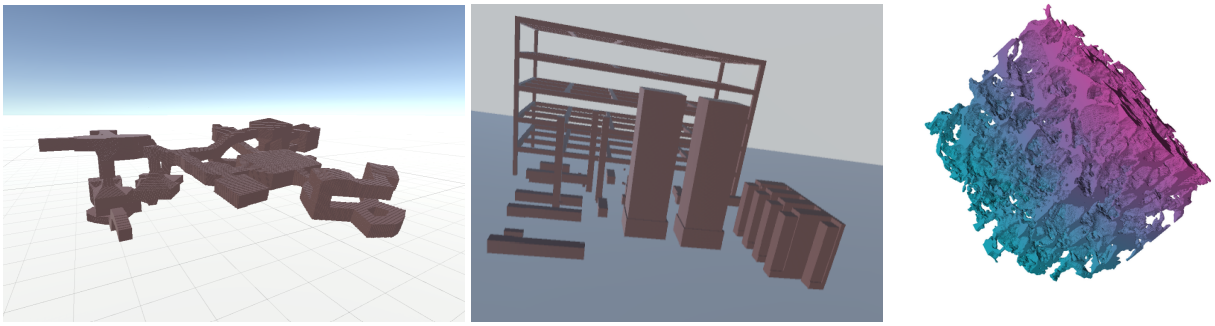

Figure 6.5: Snapshots of one map of the *Descent* (left), *Plant* (middle), and *Sandstone* (right, snapshot provided by Nobes et al. [38]) datasets.

maps are a recreation of levels from the FPS game *Descent* [1] developed by Parallax Software, featuring rooms and hallways twisting in a 3D space. The *Plant* dataset is inspired by real-life industrial plants. They feature complex structures and tasks such as optimal pipe routing deal with similar environments. The *Sandstone* dataset is inspired by porosity scans of real-life sandstone samples and consists of large cubes filled with small holes and crevasses. We give an example for each of these datasets in Figure 6.5 (with the snapshot of the *Sandstone* dataset provided by Nobes et al., for computational reasons we were not able to provide our own visualization).

All maps in the *Plant* and *Descent* datasets are mostly similar in structure and size. The *Sandstone* dataset maps however, even if they are small (they are 400x400x400 cubes,

while some *Warframe* maps reach 1000 units), have an extremely high concentration of obstacles. To put it in perspective, the map C1 from the *Warframe* dataset requires 2.5M voxels to represent all its obstacles, and maps in the *Sandstone* dataset require more than 12M voxels. Storing all these voxels in memory and iterating over them for all operations in the building of the octree is not feasible, and this shows a limit in this type of voxel representation for benchmarks. Future work could improve the efficiency of obstacle description; for example, our work handles axis-aligned bounding boxes of arbitrary size, and a much smaller number of AABBs would be required to describe the obstacles. We thus exclude *Sandstone* maps from our experiments. The *Descent* and *Plant* datasets respectively contain 31 and 5 maps, and each map in this benchmark comes with 2,000 test points.

6.3 Experiments on the Handmade Dataset

We first compare the different approaches we presented in the previous chapters on our smaller, handmade maps. As explained earlier, for each map we generate 10,000 pairs of points and test different pathfinding algorithm variants on the same set of points. Because results, especially compute time, vary significantly with the path length, to better understand the impact of path length on performance, we split these 10,000 pairs of points into evenly-sized buckets according to the path length. For each pair of points, we ran a fast merged octree pathfinding to obtain an approximation of the shortest path length between these two points, and assign the pair to the corresponding interval.

To start, we compare the compute times of the voxel baseline, the regular octree, and the merged octree. To ensure a fair comparison, the voxel grid size corresponds to the octree granularity (minimum cell size). All maps have been created to have the same size (20x20x20 units), and we used a granularity of 0.625 for these experiments, which corresponds to an octree depth of 6. We will discuss the impact of granularity more later in this chapter.

We report the results across all maps in Figure 6.6. The same trends appear across all maps: the voxel baseline is consistently much more expensive than both the regular and merged octree. Even on these small maps, voxel compute times can reach several dozens of milliseconds, except for the Industrial map, where compute times reach hundreds of milliseconds. The regular octree, however, is viable: compute time increases with path length, but on average stays below 10 milliseconds. The merged octree is significantly faster than both alternatives, with average costs never exceeding a few milliseconds. These results show that the addition of merging the regular octree significantly reduces compute times. 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 of course 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.

The other important metric of pathfinding algorithms is the path length. We compare the average path length of the voxel baseline, the regular and merged octree, as well as with the addition of the path refinement algorithms, namely the path pruning algorithm presented in Section 4.2 and the 3D funnel algorithm presented in Section 4.3. All of these results can be found in Table 6.2, and several conclusions can be drawn. Across most maps, the average path length using a voxel grid is higher, because in this baseline we only allow movement in cardinal and diagonal directions, whereas especially with path refinement more precise movement in all directions is possible. Without any form of path refinement, merged octrees lead to slightly longer paths than regular octrees, as cells are larger and the reduced navigation graph contains fewer nodes, forcing paths to take some detours to travel along the graph. This difference can be offset by using path refinement. The funnel algorithm slightly reduces average path length, while maintaining the homotopy class of a path. Path pruning reduces path length even more, and the best results are obtained by combining both algorithms (i.e., first running the funnel algorithm

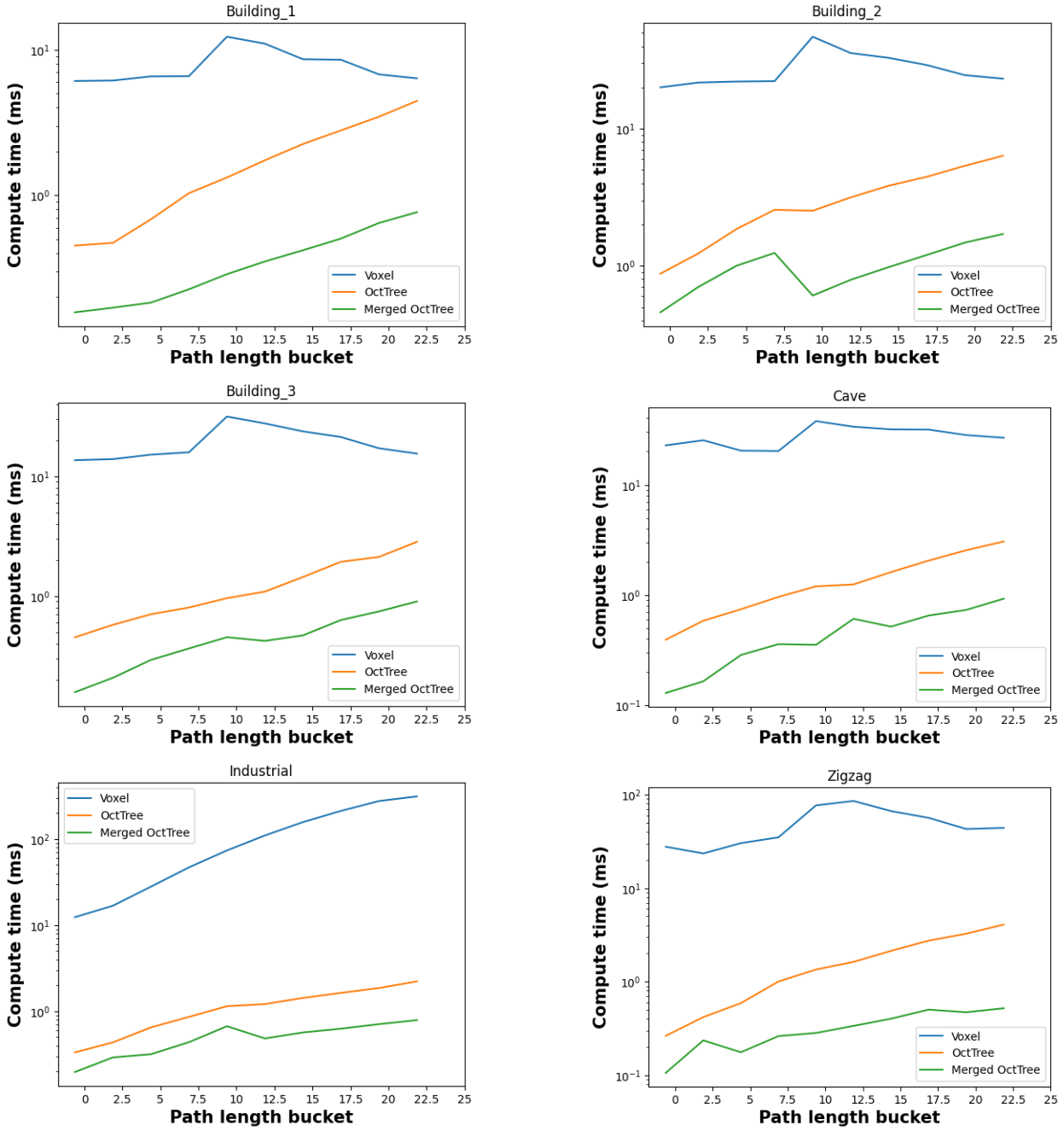

Figure 6.6: Compute time comparison of the voxel baseline, the regular, and the merged octree on all handmade maps. For better readability, the y axis is a logarithmic scale of the compute time.

and then applying path pruning to the obtained path), leading to a 5 to 10% decrease in the average path length.

Table 6.2: Comparison of the path length of the voxel baseline, the regular and merged octree on all handmade maps.

Pathfinding method	Building_1	Building_2	Building_3	Cave	Industrial	Zigzag
Voxel	11.82	13.76	14.06	13.52	13.38	14.70
Octree	12.58	13.42	13.70	12.82	12.30	13.79
Octree + pruning	11.68	12.32	12.33	**11.84**	11.28	12.65
Octree + funnel	11.77	12.45	12.48	12.02	11.42	12.80
Octree + pruning + funnel	**11.61**	**12.30**	**12.29**	11.85	**11.27**	**12.64**
Merged octree	12.51	13.09	14.04	13.71	12.23	16.61
Merged octree + pruning	11.97	12.56	12.55	12.20	11.33	13.88
Merged octree + funnel	11.79	12.43	12.64	12.31	11.43	13.63
Merged octree + pruning + funnel	11.73	12.37	12.41	12.06	11.30	13.42

Let us now discuss the variance of the results obtained. For all the experiments in this section and in Section 6.4, 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, with 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 fully assess the performance of our path refinement algorithms, it is also important to look at how costly they are. We give detailed results of the total pathfinding time of the regular and merged octrees on all maps in Figure 6.7, using one of: no refinement, path pruning, the funnel algorithm, or a combination of funnel and pruning.

Once again, similar trends appear on most maps. As expected, the compute time increases with the path length, as longer and more complex paths will result in more

nodes explored by the A* algorithm and more involved path refinement. As previously observed, using merged octrees is consistently faster than using regular octrees. This mostly affects the A* cost, however, and using path refinements adds a similar cost for both approaches. We observe that this cost is lowest for the funnel refinement, and increases if we use a combination of pruning and funnel refinement, with path pruning alone being the most expensive. While this may seem surprising, path pruning is simple but expensive, as the visibility checks used in the algorithm are more costly than the geometric operations of the funnel algorithm. The cost of path pruning scales with the number of transitions (points) in the path, and since the funnel algorithm significantly reduces this number (only "anchor" points are kept), pruning the path obtained after running the funnel algorithm is cheaper.

In general, on these small maps, path refinement is a significant part of the total pathfinding cost. On larger maps, however, such as the ones in the *Warframe* benchmark, the fraction of pathfinding cost due to path refinement decreases, going to less than 10% of the total cost on average. One possible explanation for this tendency is that the A* algorithm's cost increases with the graph size, but both the pruning and funnel algorithm iterate over the transitions along the path. We expect their cost to increase with the number of intermediate points in the path, and even in very large maps, this number does not increase too much. As we will show in Figure 6.9, on a larger map (the map *Complex* from the *Warframe* dataset), path refinement, especially using the funnel approach, represents a mall fraction of the total pathfinding cost.

Finally, we analyze the impact of granularity (or the voxel grid size for the baseline) in Figure 6.8. We compare the baseline, the regular and the merged octree on the Cave map (that fits in a 20x20x20 cube) using a granularity of 0.3125 (fine-grained), 0.625 (normal), or 1.25 (coarse). The voxel approach quickly becomes infeasible, as average compute times exceed 200 milliseconds when using the fine-grained grid size. The compute time increase with a regular octree is much more manageable, and once again merged octrees give us the best results. Compute time only slightly increases with the granularity in merged

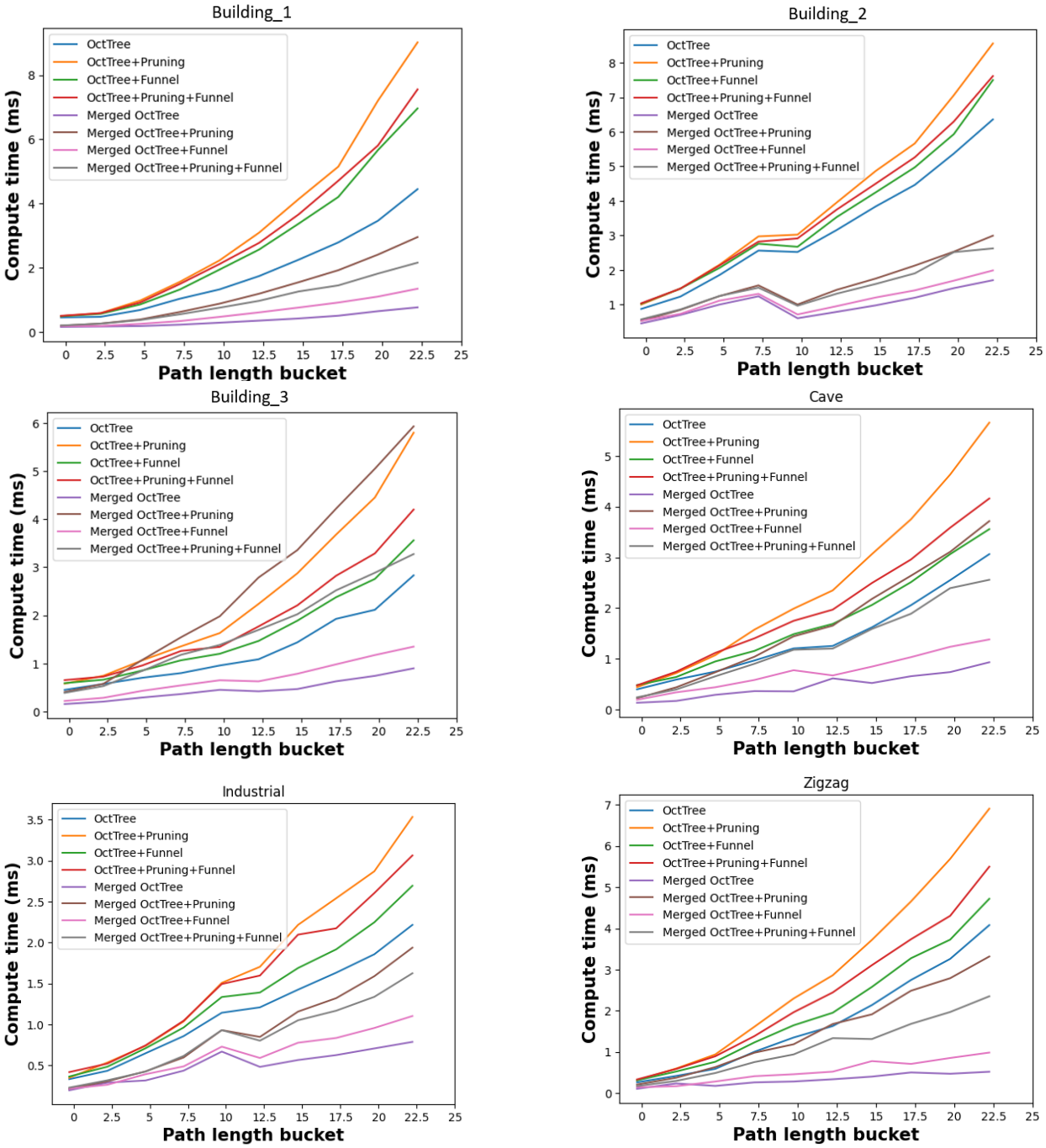

Figure 6.7: Compute time comparison of the octree and merged octree on all maps using different forms of path refinement.

octrees. Even if the potential level of detail of the octree gets higher, merging cells will greatly reduce their number, and this reduction is mostly limited by the complexity of the

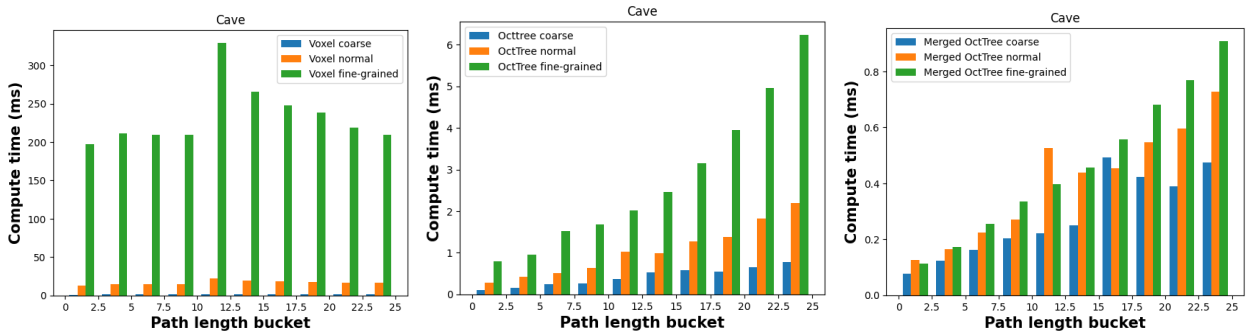

Figure 6.8: Average compute time for the voxel (left), regular (middle), and merged octree (right) on the Cave map, using a coarse (1.25), regular (0.625) and fine-grained (0.3125) granularity.

map (i.e., how detailed the obstacles are for a map of the same size, or how large a map is for obstacles typically of the same size).

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, at the cost of a small increase in path length. This increase can be offset for the most part by using path refinement methods. If there is a special motivation to respect homotopy classes, the funnel algorithm is relatively cheap and reduces path length, and if 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 chapter.

6.4 Large-Scale Experiments

To test our approach in more realistic settings, we will now run experiments on the *Warframe* dataset and the analogous datasets created by Nobes et al. Because of the long

runtime of each scenario, we limit our experiments to one map in each category of the *Warframe* dataset, since some maps are just variants of other ones. As explained in Section 6.2, we also exclude the C-named maps in the *Warframe* dataset as well as maps from the *Sandstone* dataset. To improve readability and because maps from the same dataset share similar structures, we report experiments mainly on the *Warframe* dataset, because this benchmark is the most explored in other works, as well as a few maps from the *Descent* and *Plant* dataset [38]. With this in mind, we obtain 22 maps: 19 for the *Warframe* dataset, 1 for the *Descent* dataset, and 2 for the *Plant* dataset. Each of these maps comes with 10,000 pairs of test points for the *Warframe* dataset and 2,000 for the other two.

Because obstacles (voxels) are described as integer coordinates and have a size of 1x1x1, and that test points are sampled to be close to obstacles, the octree must be created with a granularity of 1 to exactly match the obstacles and avoid scenarios where a test point would be in an invalid cell even if it is not in an obstacle. For this reason, the octree size must be a power of 2 and large enough to include all obstacles. For most maps it is either 512x512x512 or 1024x1024x1024.

We will first examine the impact of using path refinement on the total compute time to justify the previous claim that, on larger maps, path refinement only adds a small cost compared to the A* compute time. The results for three maps - a small and a large one from the *Warframe* dataset, as well as one from the *Plant* dataset - can be found in Figure 6.9. The results on path length are similar to the ones on the handmade dataset: the average path length is reduced for all paths regardless of their length. Unlike on small maps, path refinement is now much cheaper compared to the A* cost. Using path pruning alone adds a significant cost, but that is because our implementation of visibility checks is not well optimized when there is a very large number of obstacles. In general, the combination of path pruning and the funnel algorithm once again gives the best path length with a small compute time cost.

We now compare our best-performing approach (merged octree, path pruning and the funnel algorithm) to the 3D JPS algorithm [37] on the 22 maps described above. The

Figure 6.9: Evolution of the pathfinding times (left side) and path length (right side) using merged octrees and different forms of path refinement on the Complex, plant02 and BA1 maps.

median and average compute times on all maps can be found in Figure 6.11. 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 under 150 milliseconds, which remains feasible for real-time applications. On the *level01* map from the *Descent* dataset, compute times are much larger and reach almost half a second. This is because paths are much longer than on other maps (more than 700 on average compared to less

than 300 for other maps), and the environments in this dataset consist of rooms and twisting hallways that require many octree cells to represent. The A* algorithm opens on average 90,000 nodes on the level01 map while this average is closer to 10,000 on other maps. An approach based on sparser representations such as medial skeletons [43] would be much more adapted to this kind of environment.

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 22 maps is 2.29 for average and 2.71 for median times. 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. We will investigate in greater detail the impact of path length on compute times in Figure 6.12, and establish that our work is faster on longer instances.

The comparison of the average path length obtained by JPS and the octree can be found in Figure 6.11. Our work consistently computes shorter paths, with the path length over all maps being 5% lower.

As we saw on the handmade dataset, the pathfinding time can vary significantly with path length. We order the data points by path length and report the average path length for each 10-percentile on two maps from the *Warframe* dataset in Figure 6.12. 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, and 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 or milliseconds, which is viable

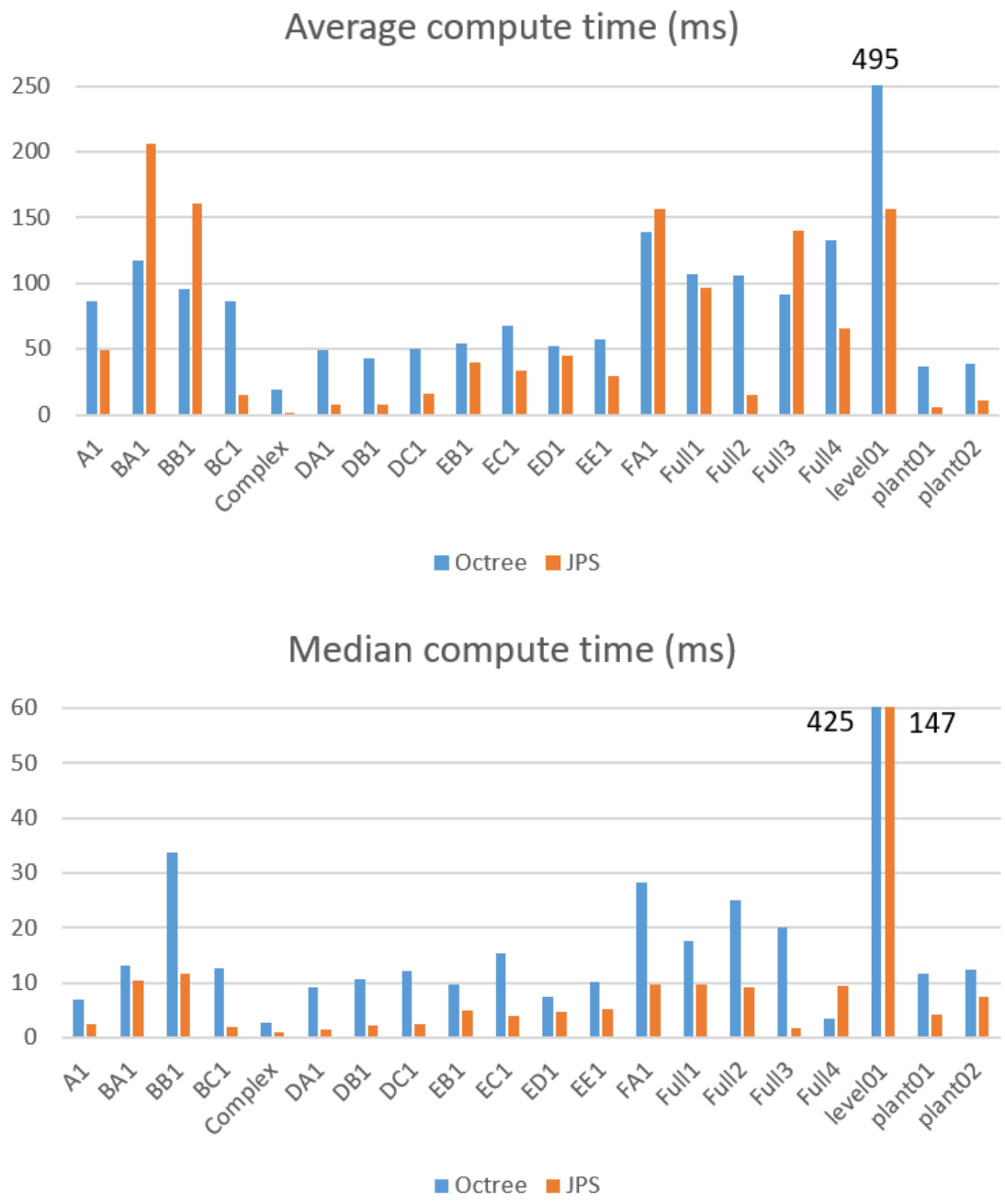

Figure 6.10: Average and median compute time of merged octrees and 3D JPS on the 3D benchmarks *Warframe*, *Plant*, and *Descent*.

for real-time video game applications. While the 3D JPS 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.

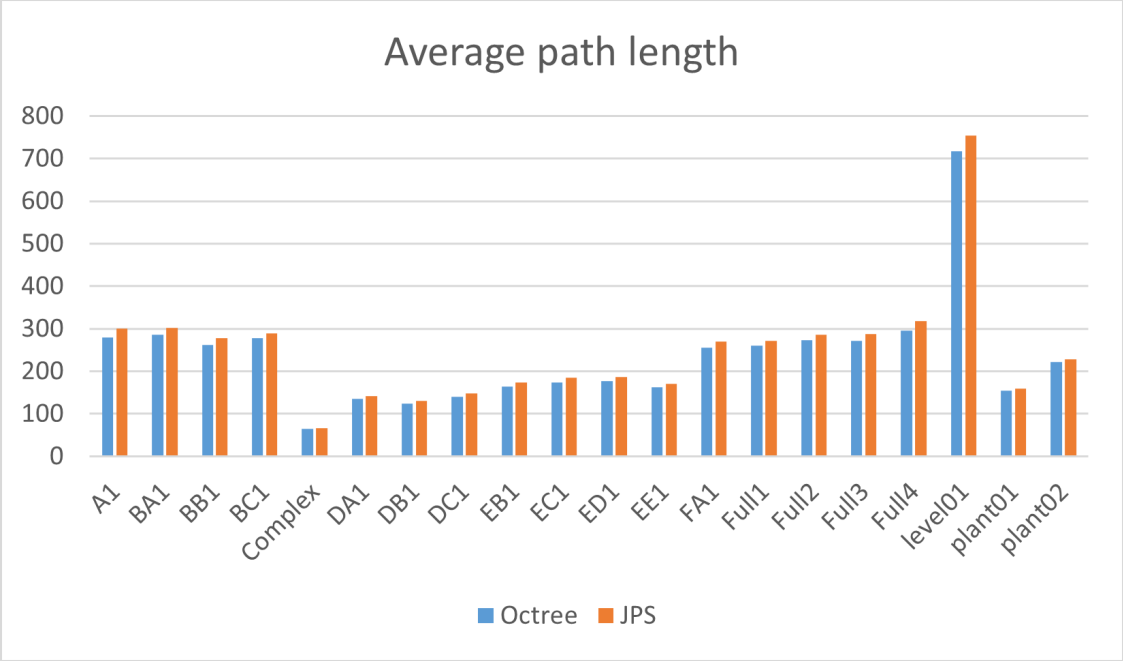

Figure 6.11: Average path length of merged octrees and 3D JPS on the 3D benchmarks.

6.5 Experiments on the Dynamic Octree

To conclude our experiments, we will now analyze the performance of the dynamic update methods presented in Chapter 5. Whenever changes occur in the environment, the cost of updating the path to take these changes into account can be split into three parts: updating the octree, updating the navigation graph, and recomputing the path on the updated graph.

To analyze the distribution of cost between these three categories, we introduce randomly moving obstacles in maps from our handmade dataset, and average the compute times over 1,000 updates. We present our results on the Industrial map, using an octree granularity of 0.625, in Table 6.3. We try to maintain a path between two static points in the level while an obstacle moves randomly. The obstacle goes to a random location in the level by following a path in the octree, and then moves to a new location and so on, to roughly imitate the behavior of an enemy patrolling through the level. The first thing to notice is that the average update time is a few milliseconds, which is the same order of magnitude as the pathfinding time. With this result and the fact that the

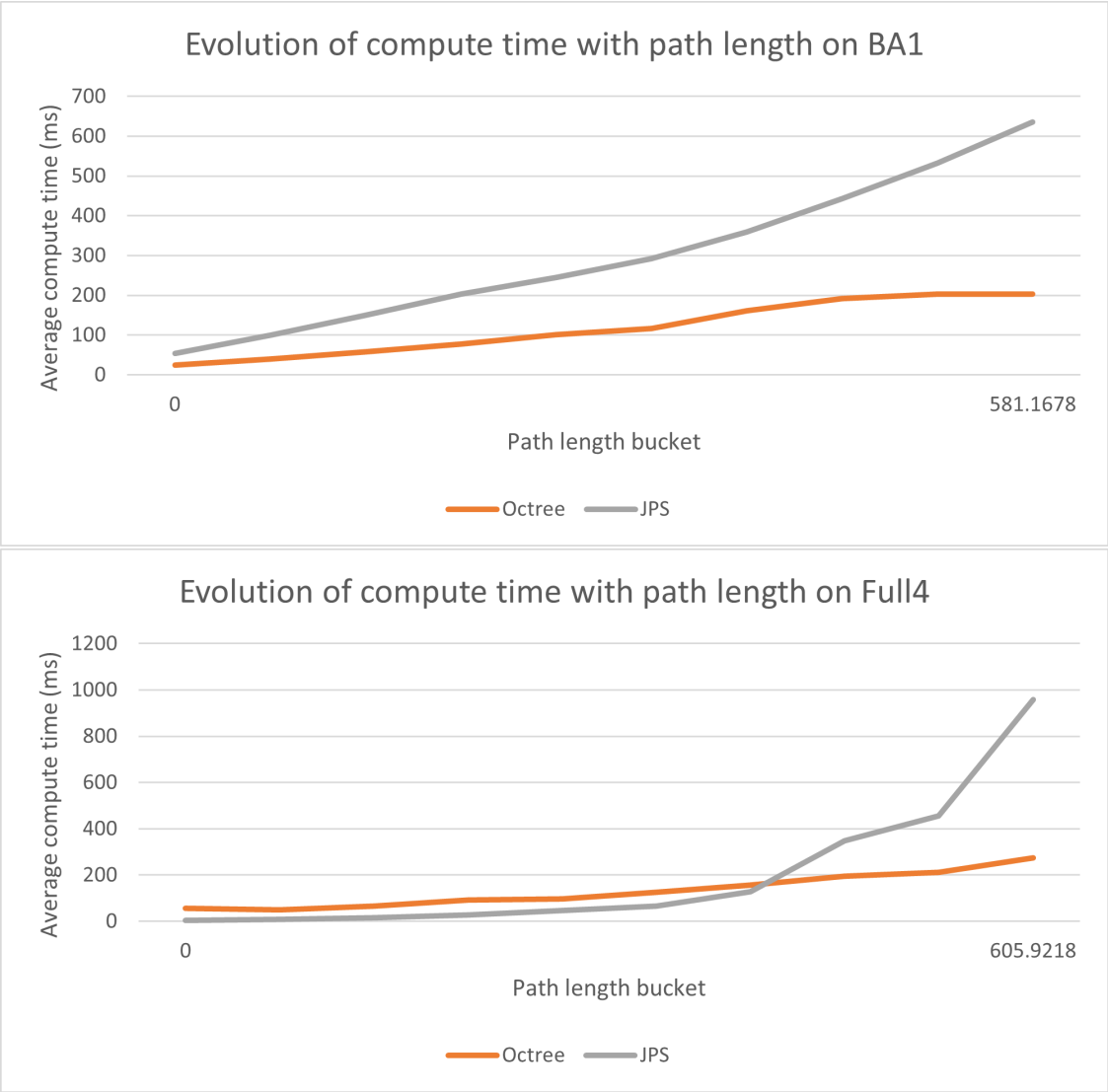

Figure 6.12: Evolution of the compute time with path length on the BA1 and Full4 maps.

octree does not need to be updated every frame in mind, we gather that real-time updates to the octree in order to achieve pathfinding in dynamic environments are feasible.

The cost distribution between the different aspects of the updates is also interesting. If we update the navigation graph by deleting the old one and recomputing it anew, it adds a cost similar to the one of updating the octree. Unlike the cost of local updates, the cost of recomputing the entire graph will also increase with the map size and prevent scalability. Moreover, recomputing the graph at every update implies allocating and de-allocating a

Table 6.3: Average cost of the octree update, graph update, and path recomputing in milliseconds over 1,000 updates on the Industrial map, using either regular or merged octrees, and locally updating or recomputing the whole navigation graph.

Update method	Octree update time	Graph update time	Path recomputing time	Total time
Octree + graph building	1.36	0.92	1.82	4.10
Octree + local graph update	**0.22**	**0.03**	1.26	1.51
Merged octree + graph building	0.44	0.34	0.72	1.50
Merged octree + local graph update	0.39	**0.03**	**0.55**	**0.97**

lot of memory, and after some time has passed this causes severe lag in the simulation. Locally updating the graph eliminates all of these problems.

If we compare updating a regular or a merged octree, as expected the path recomputing time decreases when using merged octrees. Updating the octree is slightly more expensive, because as we mentioned in Section 5.2 the process is a bit more complex, but overall the total update cost remains cheaper when using merged octrees. The non-merged octree update cost while recomputing the graph is more expensive but that is because of the memory allocation issue previously mentioned severely affecting performance.

In Section 5.1, we presented two heuristics for pruning the octree as we dynamically update it: greedily merging cells like when building the octree for the first time, or repairing parent cells if all of their children have become valid. We compare those two heuristics for updating the octree in Figure 6.13. As time passes, with the greedy merging heuristic, more and more cells are split and unable to be merged back together, leading to a total number of cells even higher than an unmerged octree and comparable to the voxel baseline. The cell-repairing approach on the other hand leads to a stable number of valid cells and can be used sustainably.

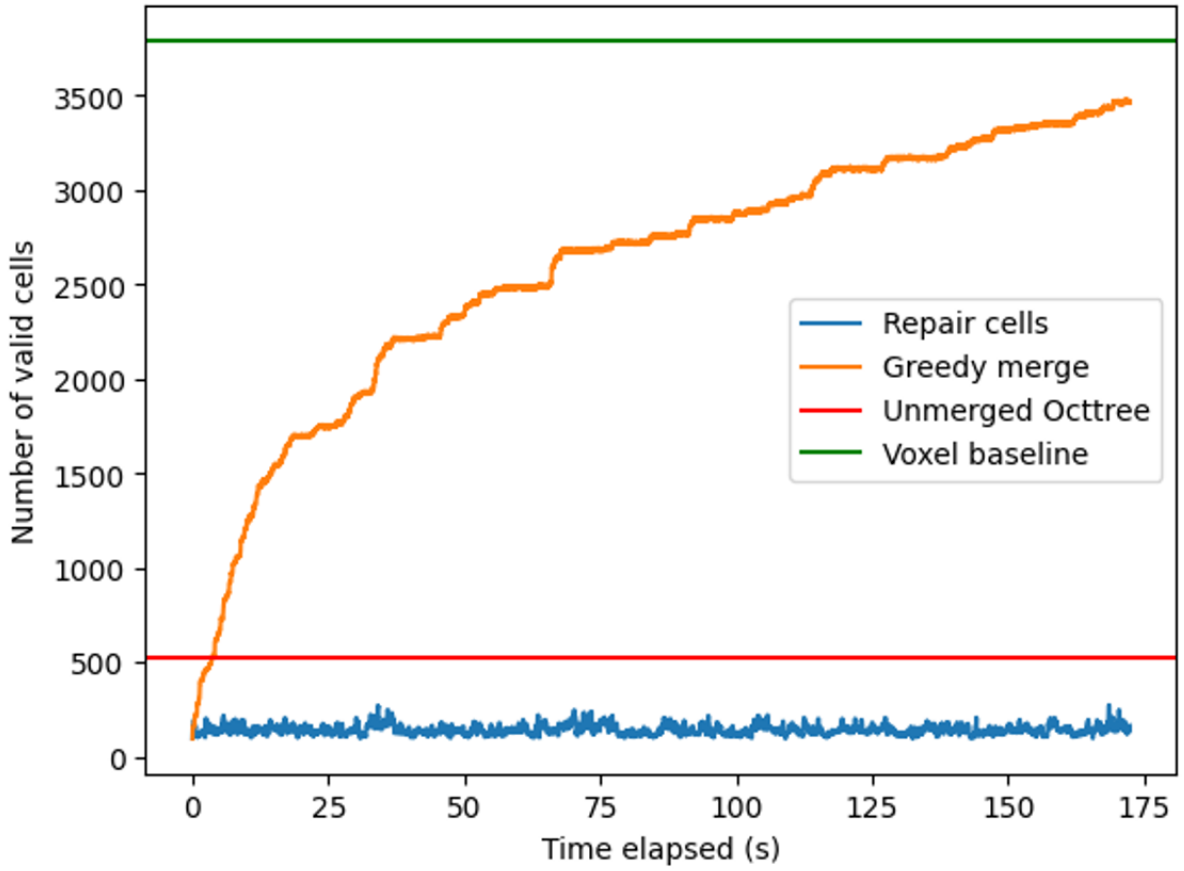

Figure 6.13: Evolution of the number of valid cells as the octree is updated with the greedy merge strategy (orange) and the repairing cells strategy (blue). The number of valid cells in the unmerged octree prior to any updates and the number of nodes in the voxel baseline are also indicated for comparison.

Finally, we described compromises that could be made to update the octree less often in Section 5.4. The first compromise that can be made is updating the octree every n frames instead of every frame. This naturally will reduce the update cost averaged over time, but by doing so some important changes in the environment could be detected late. The choice of update frequency will then depend on the type of game, how frequent changes in the environment are, and how responsive to changes the pathfinding must be.

The other compromise that can be made is only updating the octree when changes occur that would affect the computed path, and only then updating the octree and replanning the path. With this ”last moment” update method, we can ignore changes

Table 6.4: Update frequency and average update time with the last-moment update policy versus updating the octree as soon as it changes.

	Always update	Last-moment update
Update frequency	56 updates/second	0.6 updates/second
Octree update time (ms)	0.47	4.81
Graph update time (ms)	0.03	0.03
Path recomputing time (ms)	0.56	0.6
Total update time (ms)	1.17	5.44
Amortized update cost per second (ms)	65.5	3.26

that happen far away from the computed path and only update the path when necessary. To evaluate the impact of this update policy, we add three randomly moving obstacles to the Industrial map and keep track of the frequency and cost of updates over one minute in Table 6.4. Similar to the previous experiment, they move from one random location in the level to another following a path in the octree, and the movement of the obstacles interferes with the path roughly once per second. With three moving obstacles, the environment constantly changes and the octree is updated 56 times per second, and this frequency is cut down to less than one update per second with the last-moment policy. The updates are more expensive, mainly because of the octree update cost. Since updates are less frequent, more changes happen in the environment in between updates, resulting in more expensive updates. When looking at the amortized cost over time however, the last-moment update policy is far superior, with an amortized cost per second 20 times lower than the default update policy.

In this section, we saw that dynamically updating the octree in real-time game environments is feasible, and that performance can be greatly improved by locally updating the navigation graph instead of recomputing it, and updating the path at the last moment only when required. An interesting extension of this work would be to apply

it to larger maps, namely maps from the *Warframe* dataset, by having a small number of moving asteroids and ship debris floating around the otherwise static map. While this would be possible, our current implementation is not well optimized and does not allow us to differentiate static and dynamic obstacles. Future work could be made to have the octree built around static obstacles, and then only update the octree based on the movement of a reduced amount of dynamic obstacles.

Chapter 7

Conclusion and Future Work

Pathfinding is an essential component of many games for NPC path planning, and even if most 3D games restrict movement in some way or use a ”2.5D” representation planning with full 3D motion is highly desirable and would open many new possibilities in game development. Compared to 2D pathfinding, the problem of 3D pathfinding is much more computationally expensive and remains an open question. Representing a 3D environment with a naive voxel grid is too expensive, and thus producing an efficient and sparse representation of the environment is essential.

Following this idea, we developed an extension of the hierarchical octree approach in this thesis. Starting from an octree structure, we implemented a Hertel-Mehlhorn-style merging by merging adjacent cells in the octree as long as their merge would remain convex. This greedy merging approach proved very effective, reducing the number of valid cells in the octree by up to an order of magnitude.

We then created a navigation graph by connecting the centers of transition surfaces between adjacent cells together and used the A* algorithm to perform pathfinding on this graph. 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.

To improve the path quality and reduce their length we implemented a visibility-based path pruning approach and developed a 3D extension to the 2D funnel algorithm. These path refinement methods lower the 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 maintain the homotopy class of a path.

A detailed comparison with the state-of-the-art 3D JPS algorithm [37] 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. To limit the frequency of updates we propose several compromises that greatly reduce the amortized update cost over time.

We identified several interesting directions that future work on this subject could take. First of all, certain aspects of our code could be optimized further. The visibility checks used by the path pruning algorithm are expensive, and more efficient visibility checks or a different heuristic for path pruning could speed up the process. Our implementation currently does not allow us to easily distinguish static and dynamic obstacles, which limits performance on large maps with many obstacles. By separating the two kinds of obstacles, pathfinding in larger dynamic environments with a limited amount of moving obstacles could be achieved.

We used an axis-aligned bounding box representation for obstacles, especially because it suited the existing voxel benchmarks, but other, more involved forms of obstacle representation could be added. In some benchmarks such as the *Sandstone* dataset [38], maps require extremely large numbers of voxels to be represented, and further work

could be made on describing these maps more efficiently, for example with rectangular obstacles of arbitrary size instead of cubes of identical size.

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 use the splitting procedure introduced in Section 5.2 for the first split. This would 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. Extensive research has already been conducted on the subject, and an approach such as using Bezier curves to obtain smoother paths [46] could be used, although additional work would be needed to ensure that smoothed paths do not collide with obstacles.

Bibliography

[1] Descent, developed by Parallax Software. 1995.

[2] Warframe, developed by Digital Extremes. 2013. URL: https://www.warframe.com.

[3] Marvel's Spider-Man, developed by Insomniac Games. 2018. URL: https://insomniac.games/game/spider-man-ps4.

[4] Elex II, developed by Piranha Bytes. 2022. URL: https://www.elexgame.com.

[5] Armored Core 6: Fires of Rubicon, developed by From Software. 2023. URL: https://en.bandainamcoent.eu/armored-core/armored-core-vi-fires-of-rubicon.

[6] 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. DOI: 10.1155/2015/736138.

[7] Subhrajit Bhattacharya, Vijay Kumar, and Maxim Likhachev. "Search-Based Path Planning with Homotopy Class Constraints". In: *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*. AAAI'10. Atlanta, Georgia: AAAI Press, 2010, pp. 1230–1237.

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

[9] Daniel Brewer. "3D Flight Navigation Using Sparse Voxel Octrees". 2017. URL: http://www.gameaipro.com/GameAIPro3/GameAIPro3_Chapter21_3D_Flight_Navigation_Using_Sparse_Voxel_Octrees.pdf.

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

[11] 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. DOI: 10.1109/SFCS.1987.42.

[12] George Merrill Chaikin. "An algorithm for high-speed curve generation". In: *Computer Graphics and Image Processing* 3.4 (1974), pp. 346–349. ISSN: 0146-664X. DOI: https://doi.org/10.1016/0146-664X(74)90028-8.

[13] Reinis Cimurs and Il Hong Suh. "Time-optimized 3D Path Smoothing with Kinematic Constraints". In: *International Journal of Control, Automation and Systems* 18 (Jan. 2020), pp. 1277–1287. DOI: 10.1007/s12555-019-0420-x.

[14] 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.

[15] Edsger W Dijkstra. "A note on two problems in connexion with graphs". In: *Numerische mathematik* 1.1 (1959), pp. 269–271.

[16] David H. Douglas and Thomas K. Peucker. "Algorithms for the reduction of the number of points required to represent a digitized line or its caricature". In: *Cartographica: The International Journal for Geographic Information and Geovisualization* 10 (1973), pp. 112–122.

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

[18] Christer Ericson. *Real-time collision detection*. CRC Press, 2004.

[19] Florian Fichtner, Abdoulaye Diakité, Sisi Zlatanova, and Robert Voûte. "Semantic enrichment of octree structured point clouds for multi-story 3D pathfinding". In: *Transactions in GIS* 22 (Jan. 2018), pp. 233–248. DOI: 10.1111/tgis.12308.

[20] Raphael Finkel and Jon Bentley. "Quad Trees: A Data Structure for Retrieval on Composite Keys." In: *Acta Inf.* 4 (Mar. 1974), pp. 1–9. DOI: 10.1007/BF00288933.

[21] Guillermo Frontera, David Martín, Juan Portas, and Da-Wei Gu. "Approximate 3D Euclidean Shortest Paths for Unmanned Aircraft in Urban Environments". In: *Journal of Intelligent & Robotic Systems* 85 (Feb. 2017), pp. 353–368. DOI: 10.1007/s10846-016-0409-1.

[22] D. Hale, G. Youngblood, and Priyesh Dixit. "Automatically-generated Convex Region Decomposition for Real-time Spatial Agent Navigation in Virtual Worlds." In: *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference*, Jan. 2008.

[23] Daniel Harabor and Adi Botea. "Hierarchical Path Planning for Multi-Size Agents in Heterogeneous Environments". In: Dec. 2008, pp. 258–265. DOI: 10.1109/CIG.2008.5035648.

[24] Daniel Harabor and Alban Grastien. "Online Graph Pruning for Pathfinding on Grid Maps". In: *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*. AAAI'11. San Francisco, California: AAAI Press, 2011, pp. 1114–1119.

[25] Peter Hart, Nils Nilsson, and Bertram Raphael. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107. DOI: 10.1109/tssc.1968.300136.

[26] 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. DOI: https://doi.org/10.1016/j.robot.2014.10.021.

[27] 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. DOI: https://doi.org/10.1016/0925-7721(94)90010-8.

[28] Stefan Hertel and Kurt Mehlhorn. “Fast triangulation of simple polygons”. In: *Foundations of Computation Theory: Proceedings of the 1983 International FCT-Conference Borgholm, Sweden, August 21–27, 1983 4*. Springer. 1983, pp. 207–218.

[29] Julian Hirt, Dominik Gauggel, Jens Hensler, Michael Blaich, and Oliver Bittel. “Using Quadtrees for Realtime Pathfinding in Indoor Environments”. In: *Research and Education in Robotics - EUROBOT*. May 2010, pp. 14–29. ISBN: 978-3-642-27271-4. DOI: 10.1007/978-3-642-27272-1_6.

[30] Y.K. Hwang and N. Ahuja. “A potential field approach to path planning”. In: *IEEE Transactions on Robotics and Automation* 8.1 (1992), pp. 23–32. DOI: 10.1109/70.127236.

[31] S. M. LaValle. *Planning Algorithms*. Available at http://planning.cs.uiuc.edu/. Cambridge, U.K.: Cambridge University Press, 2006.

[32] Steven M. LaValle. *Rapidly-exploring random trees : a new tool for path planning*. Tech. rep. 98-11. Iowa State University, 1998.

[33] Fangyu Li, Sisi Zlatanova, Martijn Koopman, Xueying Bai, and Abdoulaye Diakité. “Universal path planning for an indoor drone”. In: *Automation in Construction* 95 (2018), pp. 275–283. ISSN: 0926-5805. DOI: https://doi.org/10.1016/j.autcon.2018.07.025.

[34] Tomás Lozano-Pérez and Michael A. Wesley. “An algorithm for planning collision-free paths among polyhedral obstacles”. In: *Commun. ACM* 22.10 (1979), pp. 560–570. ISSN: 0001-0782. DOI: 10.1145/359156.359164. URL: https://doi.org/10.1145/359156.359164.

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

[36] 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*. ISCSIC 2019. Amsterdam, Netherlands: Association for Computing Machinery, 2020, pp. 1–6. ISBN: 9781450376617. DOI: 10.1145/3386164.3386181.

[37] Thomas K. Nobes, Daniel Harabor, Michael Wybrow, and Stuart D. C. Walsh. "The JPS pathfinding system in 3D". In: *Proceedings of the International Symposium on Combinatorial Search*. Fifteenth InternationalSymposium on Combinatorial Search 1 (2022). Ed. by Lukás Chrpa and Alessandro Saetti, pp. 145–152. URL: https://sites.google.com/unibs.it/socs2022.

[38] Thomas K. Nobes, Daniel Harabor, Michael Wybrow, and Stuart D.C. Walsh. "Voxel Benchmarks for 3D Pathfinding: Sandstone, Descent, and Industrial Plants". In: *Proceedings of the Sixteenth International Symposium on Combinatorial Search*. AAAI Press, 2023, pp. 56–64.

[39] Craig Reynolds. "Steering Behaviors For Autonomous Characters". In: *Proceedings of the Game Developers Conference* (1999), pp. 763–782.

[40] Michael Schwarz and Hans-Peter Seidel. "Fast Parallel Surface and Solid Voxelization on GPUs". In: *ACM SIGGRAPH Asia 2010 Papers*. SIGGRAPH ASIA '10. Seoul, South Korea: Association for Computing Machinery, 2010, pp. 1–10. ISBN: 9781450304399. DOI: 10.1145/1866158.1866201.

[41] 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. URL: https://api.semanticscholar.org/CorpusID:15841913.

[42] 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. DOI: 10.1609/aiide.v7i1.12438.

[43] W.G. van Toll, A.F. Cook IV, M.J. van Kreveld, and R. Geraerts. "The Explicit Corridor Map: Using the Medial Axis for Real-Time Path Planning and Crowd Simulation". In: *International Computational Geometry Multimedia Exposition*. 2016, 70:1–70:5.

[44] Wouter van Toll, Roy Triesscheijn, Marcelo Kallmann, Ramon Oliva, Nuria Pelechano, Julien Pettré, and Roland Geraerts. "A Comparative Study of Navigation Meshes". In: *Proceedings of the 9th International Conference on Motion in Games*. MIG '16. Burlingame, California: Association for Computing Machinery, 2016, pp. 91–100. ISBN: 9781450345927. DOI: 10.1145/2994258.2994262.

[45] Wouter van Toll, Roy Triesscheijn, Marcelo Kallmann, Ramon Oliva, Nuria Pelechano, Julien Pettré, and Roland Geraerts. "Comparing navigation meshes: Theoretical analysis and practical metrics". In: *Computers & Graphics* 91 (2020), pp. 52–82. ISSN: 0097-8493. DOI: https://doi.org/10.1016/j.cag.2020.06.006.

[46] 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. DOI: 10.1109/IROS.2008.4650637.

[47] 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. DOI: 10.1080/13658810801949850.