Comparing Spatial Partitioning Techniques for 3D Game Engines: CPSC 490 Senior Thesis

Tam Vu
tam.vu@yale.edu

Advisor: Michael Shah
michael.shah@yale.edu

A Senior Thesis as a partial fulfillment of requirements for the Bachelor of Science in Computer Science

Department of Computer Science
Yale University
April 21, 2025

Acknowledgements

I want to express my gratitude to Professor Michael Shah for his support and guidance throughout this project, and for reigniting my passion for game development. I want to thank my parents for supporting my education and giving me the opportunity and space to grow not just as a student, but as a person. I want to thank my colleagues for pushing me, inspiring me, and constantly teaching me new things about the world and about myself. I am very grateful for all the resources that have enriched my time here at Yale University.

Contents

1 Introduction **6**

2 Background **9**
2.1 Binary Space Partitioning (BSP) 9
2.1.1 Introduction 9
2.1.2 Mathematical Formulation of BSP Tree Construction 9
2.1.3 Rendering with BSP Trees 10
2.2 Bounding Volume Hierarchy (BVH) 10
2.2.1 Introduction 10
2.2.2 Mathematical Construction of BVH 10
2.2.3 Traversal and Intersection Tests 11
2.3 Octrees 12
2.3.1 Introduction 12
2.3.2 Mathematical Construction of Octrees 12
2.3.3 Traversal and Intersection Tests 13

3 Methodology **14**
3.0.1 Dependencies and Packages 14
3.1 Implementation of Rendering Algorithms 14
3.1.1 Rendering Algorithms 14
3.2 Benchmarking Environment 15
3.3 Performance Evaluation and Metrics 15
3.4 Experimental Setup 16
3.5 Summary 16

4 Results **17**
4.1 Pre-processing Speed Comparisons 17
4.1.1 Explanation 17
4.2 Frames Per Second Comparisons 18
4.2.1 Explanation 18
4.3 Memory Usage Comparison 19
4.3.1 Explanation 20

5 Related Work **21**
5.1 Related Works 21

6 Conclusions **23**

7 Future Work **25**
7.1 Future Work 25

A About The Game **27**
A.1 The Engine 27
A.2 Scene Generation 27
A.3 The Game 27
A.4 Tree Initialization 29
A.4.1 BSP 29
A.4.2 BVH 29
A.4.3 Octrees 29

Comparing Spatial Partitioning Techniques for 3D Game Engines

Tam Vu

Abstract

Efficient spatial partitioning is crucial in 3D game engines for optimizing rendering, collision detection, and scene management. As modern game worlds become more complex, the choice of spatial partitioning technique significantly impacts performance. While various methods such as Binary Space Partitioning (BSP), Bounding Volume Hierarchies (BVH), and Octrees are widely used, their relative strengths and weaknesses remain a topic of ongoing research. This project addresses the following research questions:

- How do different spatial partitioning techniques compare in terms of rendering and collision detection performance?
- What are the trade-offs in memory usage, preprocessing time, and real-time efficiency among these methods?
- Which partitioning technique is best suited for different game scenarios (e.g., dense urban environments vs. open landscapes)?

This project implements and compare multiple spatial partitioning techniques in a controlled 3D game engine environment. By benchmarking their performance in rendering and collision detection, we can determine the most effective methods for different types of game scenes. This paper aims to provide a clear comparison of rendering algorithms for game developers.

This project implements and compare multiple spatial partitioning techniques in a controlled 3D game engine environment. By benchmarking their performance in rendering and collision detection, we can determine the most effective methods for different types of game scenes. This paper aims to provide a clear comparison of rendering algorithms for game developers.

1 Introduction

Graphics serve as the backbone of video games, shaping the visual and immersive experience that defines interactive entertainment. From the earliest pixelated sprites of *Pac-Man* (Figure 1.1) to the modern photorealistic environments of *Red Dead Redemption 2* (Figure 1.2), advancements in graphics technology have driven the evolution of gaming, enabling richer storytelling, more dynamic gameplay, and greater player engagement. The ability to render detailed characters, lifelike lighting, and expansive worlds in real time is a testament to the power of modern rendering techniques and hardware acceleration. Graphics not only enhance aesthetic appeal but also play a critical role in gameplay mechanics—whether through visibility optimizations, physics-based interactions, or immersive visual feedback. As video games continue to push the boundaries of realism and performance, graphics technology remains at the core, making spatial partitioning, rendering algorithms, and efficient memory management essential for delivering seamless and compelling gaming experiences. As video games become an increasing popular past-time, gamers seek for more complex game worlds, vaster landscapes, and more dynamic objects, thus optimizing rendering efficiency becomes paramount. One of the key challenges in 3D rendering is spatial partitioning structuring the scene in a way that allows the engine to efficiently determine which objects are visible and which can be ignored in a given frame. Without effective partitioning, rendering pipelines can become bottlenecked by unnecessary computations, leading to performance degradation.

To address this challenge, various spatial partitioning techniques have been developed, each offering unique advantages and trade-offs. Binary Space Partitioning (BSP) has been historically favored for static environments due to its strict hierarchical segmentation of space. Bounding Volume Hierarchies (BVH), commonly used in ray tracing, provide an efficient means of organizing objects into bounding volumes for fast intersection tests. See Figure 1.3 for a visualization of BVH in use. Meanwhile, Octrees dynamically subdivide space into hierarchical cubes, making them particularly effective for large, sparsely populated environments See Figure 1.4 for a visualization of Octrees in use. Despite their widespread use, the relative strengths and weaknesses of these techniques in real-time rendering and collision detection remain an open question, especially as modern hardware and game engine architectures continue to evolve.

This thesis investigates the comparative effectiveness of BSP, BVH, and Octrees in the context of real-time rendering and collision detection. Existing research primarily focuses on these methods in isolation or within specific applications, such as ray tracing or physics simulations. However, a comprehensive comparison of their performance in a unified game engine setting remains underexplored. This study fills this gap by implementing BSP, BVH, and Octree-based partitioning within a controlled 3D game engine and benchmarking their effectiveness across various scenes and workloads.

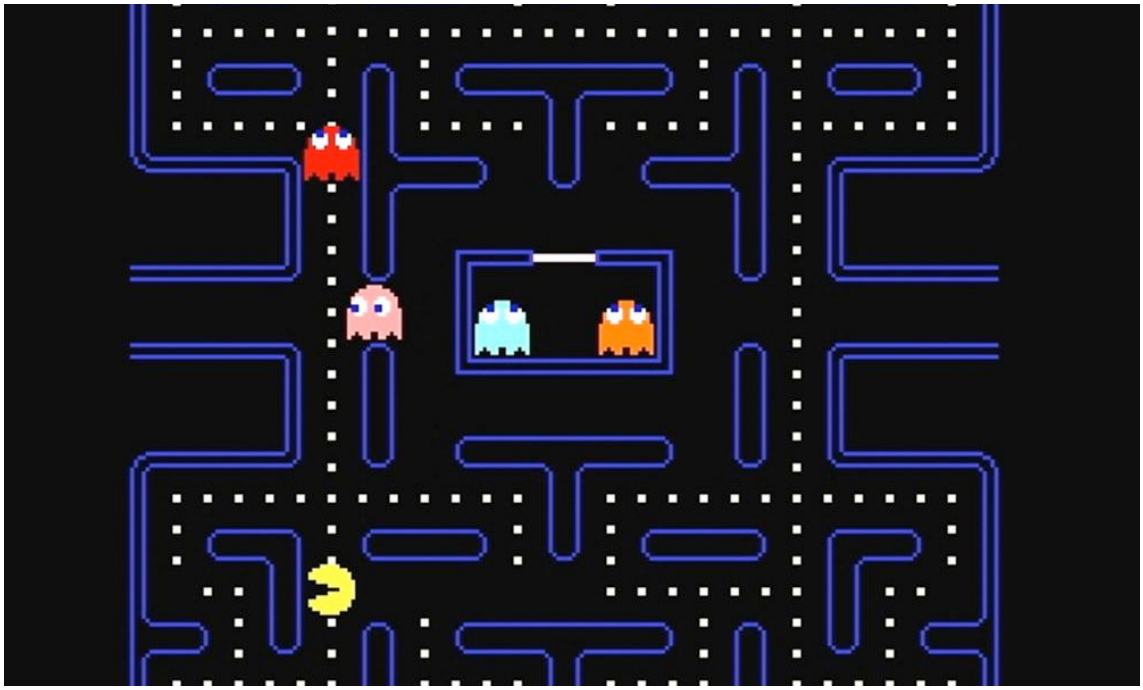

Figure 1.1: Pac-Man

Figure 1.2: Red Dead Redemption 2

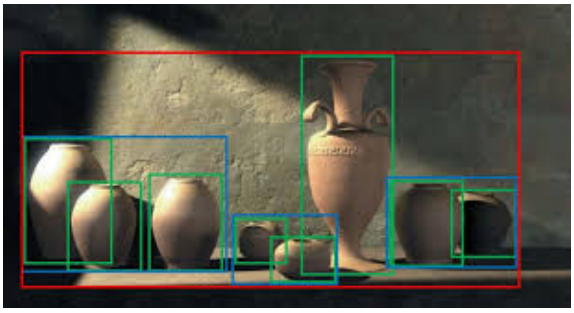

Figure 1.3: BVH used to partition a scene

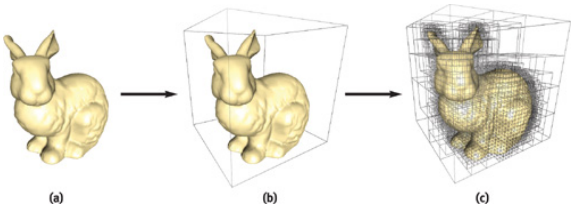

Figure 1.4: Octrees used to partition a bunny

The methodology involves constructing identical game environments and integrating each partitioning technique separately. Performance metrics such as frame rendering time, memory consumption, and collision detection speed are collected and analyzed to provide empirical insights into their efficiency. The findings of this study contribute to a deeper understanding of how spatial partitioning impacts real-time game performance and offer practical guidance to game developers in selecting the most suitable technique based on their specific game design constraints.

By systematically evaluating these algorithms, this research aims to provide a clear and data-driven comparison of spatial partitioning techniques, equipping game developers with the knowledge needed to optimize performance in modern 3D game engines.

2 Background

2.1 Binary Space Partitioning (BSP)

2.1.1 Introduction

Binary Space Partitioning (BSP) is a hierarchical data structure used to recursively subdivide a Euclidean space using hyperplanes. It is particularly effective for visibility determination, occlusion culling, and accelerating ray tracing in 3D graphics. Given a set of polygons that define a scene, BSP constructs a binary tree where each internal node represents a partitioning plane, and the leaf nodes contain convex subregions of space.

Let $S \subset \mathbb{R}^3$ be a set of polygons representing a 3D scene. The BSP tree recursively partitions S using hyperplanes H_i, such that each polygon $p \in S$ is assigned to a unique node of the tree, ensuring that every region in space is convex.

2.1.2 Mathematical Formulation of BSP Tree Construction

Given an initial set of polygons P, a BSP tree is constructed recursively as follows:

1. Select a polygon $p_i \in P$ as a partitioning plane H_i. This plane is defined by the standard plane equation:

$$ax + by + cz + d = 0,$$

where (a, b, c) is the normal vector of the plane, and d is the distance from the origin.

2. Classify each polygon $p_j \in P$ relative to H_i:

- If all vertices of p_j satisfy $ax + by + cz + d > 0$, then p_j is in the front subspace.
- If all vertices satisfy $ax + by + cz + d < 0$, then p_j is in the back subspace.

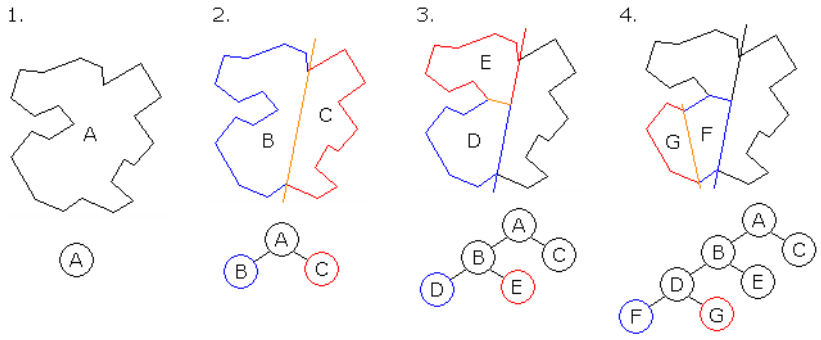

Figure 2.1: Visualization of BSP

- If p_j intersects H_i, it is split into two new polygons p_j^+ and p_j^-, which belong to the front and back subspaces, respectively.

3. The process is repeated recursively for each subspace until termination conditions are met, such as reaching a leaf node with a single polygon or a predefined complexity threshold.

2.1.3 Rendering with BSP Trees

BSP trees are particularly useful for hidden surface removal and visibility ordering. Given a viewpoint $V = (x_v, y_v, z_v)$, the rendering process involves a depth-first traversal of the BSP tree:

1. Let H_i be the splitting plane at the current node. Compute the signed distance of V to H_i:

$$d_v = ax_v + by_v + cz_v + d.$$

If $d_v > 0$, then V is in the front subspace; otherwise, it is in the back subspace.

2. If V is in the front subspace, traverse the back subspace first, then render the polygons at H_i, and finally traverse the front subspace. If V is in the back subspace, the order is reversed. This ensures a back-to-front (Painter's Algorithm) or front-to-back rendering order, which is essential for z-buffering and efficient depth testing.

2.2 Bounding Volume Hierarchy (BVH)

2.2.1 Introduction

Bounding Volume Hierarchy (BVH) is a hierarchical spatial partitioning structure used in computer graphics to accelerate rendering, ray tracing, and collision detection. Unlike Binary Space Partitioning (BSP), which partitions space, BVH organizes objects into a tree structure where each node encapsulates a bounding volume containing its child nodes. This hierarchical representation enables efficient traversal and intersection tests by quickly eliminating large portions of the scene that do not contribute to the final image.

Formally, let $S = \{O_1, O_2, ..., O_n\}$ be a set of objects in a 3D scene. The BVH recursively partitions S into disjoint subsets, each enclosed by a bounding volume B_i, such that the root node encompasses the entire scene, and each child node represents a subset of objects enclosed in a tighter bounding volume.

2.2.2 Mathematical Construction of BVH

A BVH tree is constructed recursively as follows:

1. Define a bounding volume function $B(S)$ that computes a bounding volume encompassing a given set of objects S. Common bounding volumes include:

- Axis-Aligned Bounding Boxes (AABB): Defined by min/max corner points $(x_{\min}, y_{\min}, z_{\min})$ and $(x_{\max}, y_{\max}, z_{\max})$.

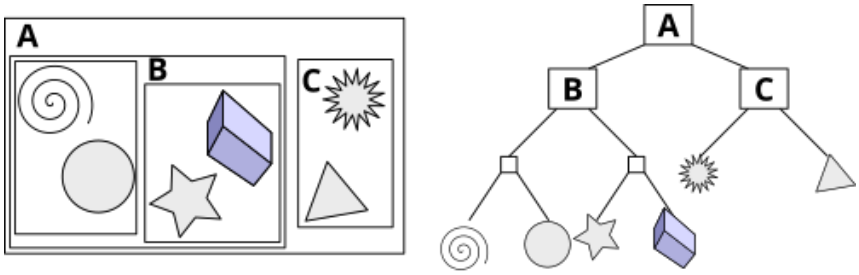

Figure 2.2: Visualization of BVH

- Bounding Spheres: Defined by a center C and radius r such that all objects in S satisfy $\|p - C\| \leq r$.

2. Partition S into two disjoint subsets S_1 and S_2, usually based on heuristics such as:

- Surface Area Heuristic (SAH): Minimizes the expected number of intersection tests.

- Spatial Median Split: Partitions objects by their centroid along the longest axis.

3. Recursively apply steps (1) and (2) to each subset until a termination criterion is met (e.g., leaf nodes contain a single object or a small number of objects).

2.2.3 Traversal and Intersection Tests

BVH trees accelerate ray-object intersection tests by allowing early elimination of large portions of the scene. Given a ray $R(t) = O + tD$ with origin O and direction D, its intersection with a bounding volume B is computed first before checking the enclosed objects.

For an Axis-Aligned Bounding Box (AABB), ray-box intersection is determined using the slab method:

$$t_{\min} = \max\left(\frac{x_{\min} - O_x}{D_x}, \frac{y_{\min} - O_y}{D_y}, \frac{z_{\min} - O_z}{D_z}\right),$$

$$t_{\max} = \min\left(\frac{x_{\max} - O_x}{D_x}, \frac{y_{\max} - O_y}{D_y}, \frac{z_{\max} - O_z}{D_z}\right).$$

If $t_{\min} > t_{\max}$, the ray misses the bounding box, and no further intersection tests are needed.

Traversal follows a depth-first or priority queue approach: 1. If a ray intersects the bounding volume of a node, its children are examined. 2. If the ray misses a bounding volume, the entire subtree is discarded. 3. When a leaf node is reached, actual intersection tests with individual objects are performed.

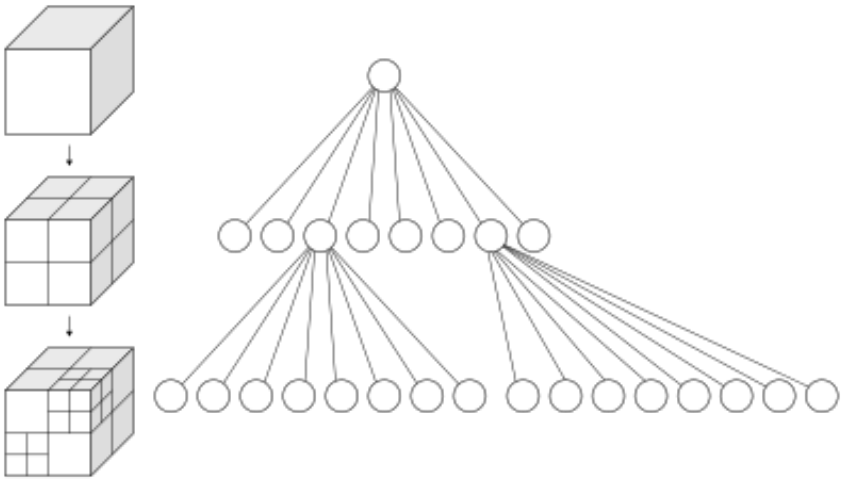

Figure 2.3: Visualization of Octrees

2.3 Octrees

2.3.1 Introduction

Octrees are tree-based spatial partitioning structures that recursively divide 3D space into eight octants. Unlike BVH, which groups objects based on bounding volumes, Octrees partition the scene volume itself. Each node in an Octree represents a cube in 3D space, and has either zero or eight children, each corresponding to a sub-region (octant) of the parent cube. This method is especially effective for scenes where the spatial distribution of geometry is sparse or non-uniform, enabling efficient empty space skipping and localized intersection checks. Let $\mathcal{V} \subset \mathbb{R}^3$ denote the 3D bounding volume of the entire scene. The root node represents $\mathcal{V}$, and each child node recursively subdivides its parent volume into eight axis-aligned sub-cubes.

2.3.2 Mathematical Construction of Octrees

Octree construction follows a recursive spatial subdivision process:

1. Given a cube $C \subset \mathbb{R}^3$ with side length L, define its eight children as sub-cubes:

$$C_{i,j,k} = \left[x_0 + i\frac{L}{2}, x_0 + (i+1)\frac{L}{2}\right] \times \left[y_0 + j\frac{L}{2}, y_0 + (j+1)\frac{L}{2}\right] \times \left[z_0 + k\frac{L}{2}, z_0 + (k+1)\frac{L}{2}\right],$$

where $(i, j, k) \in \{0, 1\}^3$ and (x_0, y_0, z_0) is the minimum corner of C.

2. For each sub-cube $C_{i,j,k}$, collect the set of primitives (e.g., triangles) intersecting it:

$$S_{i,j,k} = \{O \in S \mid O \cap C_{i,j,k} \neq \emptyset\}.$$

3. Recurse on each non-empty $S_{i,j,k}$ until a termination condition is satisfied, such as:

- Maximum tree depth d_{max} is reached.

- The number of primitives in a node is below a threshold t_{min}.
- The spatial extent of the cube becomes smaller than a minimum resolution.

2.3.3 Traversal and Intersection Tests

Octrees facilitate efficient ray traversal and frustum culling by allowing early elimination of entire spatial regions. Given a ray $R(t) = O + tD$, the algorithm proceeds as follows:
1. Test the ray against the bounding cube of the root node using slab-based AABB intersection.
2. If intersected, recursively test the ray against child nodes in the order determined by entry t-values, prioritizing near-to-far traversal for early hit detection.
3. At leaf nodes, perform intersection tests with the contained primitives.

This spatial partitioning can reduce ray-primitive intersection complexity from $O(n)$ to approximately $O(\log n)$ in balanced trees. Additionally, empty regions of space can be skipped entirely, making octrees well-suited for scenes with large unoccupied volumes (e.g., voxelized terrain or sparse meshes).

3 Methodology

This study aims to compare the performance of different 3D rendering algorithms by implementing them in a controlled first-person game environment and analyzing their efficiency under various conditions. The methodology consists of three main components: (1) implementation of multiple rendering algorithms, (2) controlled benchmarking using a first-person game, and (3) performance evaluation through empirical measurements.

This project is implemented in Python, leveraging OpenGL for rendering and using Pygame for window management and user input. Python was chosen for its flexibility, ease of prototyping, and the availability of libraries for graphics rendering and performance analysis.

3.0.1 Dependencies and Packages

The implementation relies on several key Python libraries:

- **Pygame (pygame)**: Used for window management, event handling, and user input.
- **PyOpenGL (PyOpenGL)**: Provides bindings to OpenGL for rendering 3D graphics.
- **NumPy (numpy)**: Used for efficient numerical operations, such as vector math and matrix transformations.
- **JSON (json)**: Handles level loading and saving in the level editor.

3.1 Implementation of Rendering Algorithms

To facilitate direct comparisons, the study involves implementing multiple rendering algorithms in a consistent and modular framework. The rendering methods are incorporated into a custom-built first-person game environment, where the player can navigate and interact with a 3D scene.

3.1.1 Rendering Algorithms

The algorithms implemented for comparison include:

- **Binary Space Partitioning (BSP)**: A tree-based rendering approach that recursively subdivides the scene into front and back partitions. BSP is known for efficient occlusion culling but may introduce high preprocessing costs.

- **Bounding Volume Hierarchy (BVH)**: A hierarchical structure that groups objects into bounding volumes, allowing for efficient ray traversal and intersection tests. BVH is commonly used in ray tracing and dynamic scenes.
- **Octrees (Planned Future Implementation)**: A spatial partitioning structure that recursively divides the 3D space into eight octants, optimizing visibility and collision detection.

Each algorithm is implemented within a unified rendering pipeline, allowing for a direct comparison of their performance in the same environment. The game can be executed with different algorithms by specifying the desired method via a command-line argument, e.g., `python main.py bsp` or `python main.py bvh`.

3.2 Benchmarking Environment

The first-person game serves as a controlled benchmarking environment where the player's movement, scene complexity, and rendering conditions remain consistent across different tests. The game provides an interactive 3D world with the following key features:

- **User-controlled Navigation**: Players can move freely within the scene, generating different viewpoints and rendering demands.
- **Procedurally Defined Levels**: Scenes are created using a level editor that allows for dynamic wall placement, ensuring that complexity can be varied systematically.
- **Uncapped Frame Rate**: The game runs without a frame rate cap to accurately measure the performance of each rendering algorithm. Frame rate is logged over time to observe fluctuations and performance trends.
- **Overlay Performance Metrics**: Real-time FPS is displayed in the window title, allowing for immediate feedback on rendering performance.

The scene complexity is controlled by the number of walls in that scene. We will test on scenes with $10, 50, 100, 500, 1,000, 5,000, 10,000$ and $20,000$ walls.

By keeping all variables constant except for the rendering algorithm, the study ensures a fair comparison of each method's efficiency.

3.3 Performance Evaluation and Metrics

To assess the efficiency of each rendering algorithm, multiple performance metrics are collected and analyzed:

- **Frame Rate (FPS)**: The number of frames rendered per second, serving as a direct indicator of performance. Higher FPS suggests a more efficient rendering algorithm.

- **Scene Complexity Scaling**: The impact of increasing scene complexity (e.g., number of walls, object count) on pre-processing time.
- **Memory Usage**: The amount of memory consumed by the rendering data structures (BSP tree, BVH nodes, etc.).

Data collection is performed by running automated test sequences within the game, measuring FPS over time, and logging performance statistics for each rendering algorithm.

3.4 Experimental Setup

To ensure repeatability and reliability, all experiments are conducted on the same hardware setup. The test system specifications include:

- **Processor**: Apple M1 Pro (8 core)
- **Graphics Card**: Integrated Apple M1 Pro GPU (14 core)
- **Memory**: 16GB
- **Operating System**: macOS
- **Rendering API**: OpenGL (via PyOpenGL)

Each rendering algorithm is tested under identical conditions, with results averaged over multiple runs to account for variability.

3.5 Summary

This methodology ensures a rigorous and unbiased evaluation of different rendering techniques. By leveraging a controlled first-person game environment, this study provides a real-world benchmark for rendering algorithms, offering insights into their efficiency in interactive 3D applications. The next section presents the experimental results and analysis.

4 Results

4.1 Pre-processing Speed Comparisons

Pre-processing refers to reading level data (e.g., walls, textures) from files and converting it into a format suitable for rendering or collision detection. For our tests, we calculate the pre-processing time as the time it takes our renderer to finish building the respective data structures on initialization. We initialize the game 100 times for each set of number of walls we are testing for, across the three rendering algorithms. The results are given visualization in Figure 4.1. We see that the pre-processing times grow exponentially as the number of walls to render increases.

We can see that the BSP performs the worst as we scale up the environment. All three algorithms have similar preprocessing times for when the number walls are 100 or less. At 500 walls, we begin to see some difference. BSP takes on average 0.21815 seconds preprocess, while BVH takes .20787 seconds and Octree takes 0.18045 seconds. When we are rendering with 1,000 walls, BSP takes 0.47403 seconds, BVH takes 0.37698 seconds and Octree takes 0.29637 seconds. At 50,000 walls, we see a significant difference in pre-processing times. BSP takes 60.33071 seconds, BVH takes 27.09668 seconds and Octrees took 11.50221 seconds.

4.1.1 Explanation

BSP trees recursively split the space into two halves at each node, which can lead to a highly unbalanced tree if the walls are not evenly distributed. As the number of walls increases, the depth of the tree grows significantly, resulting in more recursive calls during traversal and increasing the computational overhead. BSP trees do not inherently group nearby walls together. It is likely that walls that are spatially close may be stored in different parts of the tree, requiring more traversal steps.

Octrees divide the space into eight equally sized regions (octants) at each level. This ensures that walls are grouped spatially, reducing the depth of the tree and improving traversal efficiency. Nearby walls are more likely to be in the same or adjacent octants, minimizing the number of nodes that need to be visited. Octrees scale better with increasing numbers of walls because their hierarchical structure ensures that the workload is distributed evenly across the tree. This leads to faster preprocessing compared to BSP.

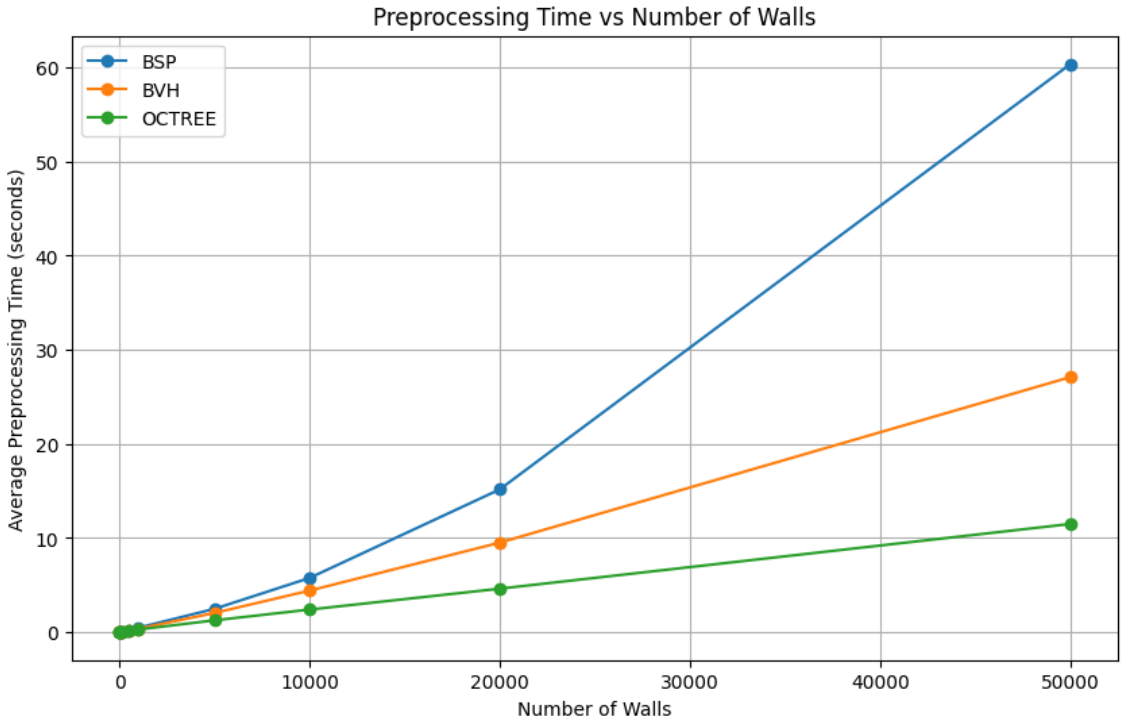

Figure 4.1: Preprocessing Time vs. Number of Walls

4.2 Frames Per Second Comparisons

Before we examine the frames per second (FPS) performance of each algorithm, we must understand how FPS is calculated. During runtime, we calculate the current FPS based on the time elapsed between frames. The PyGame library then uses an internal function to calculate the time between frames in milliseconds (dt), and then the FPS is calculated as

$$1000/dt$$

For each rendering algorithm, we run the game and play for 30 seconds. We log the FPS for each frame, and then calculate the average FPS across all frames. Due to hardware limitations, testing with environment with thousands of walls is not possible. Thus, we instead test on a range of 0 to 200 walls. A plot of the averaged frames per second for each count of walls is shown in Figure 4.2.

4.2.1 Explanation

As the number of walls in the scene increases, the performance of each rendering algorithm — measured in frames per second (FPS) — degrades at different rates due to the structural properties of the underlying spatial partitioning technique. The BSP algorithm exhibits the steepest drop in FPS, with performance declining in a nonlinear, exponential-like fashion. This is primarily because BSP recursively splits space along the planes defined by wall geometry. As more walls are added, the tree becomes deeper, and many

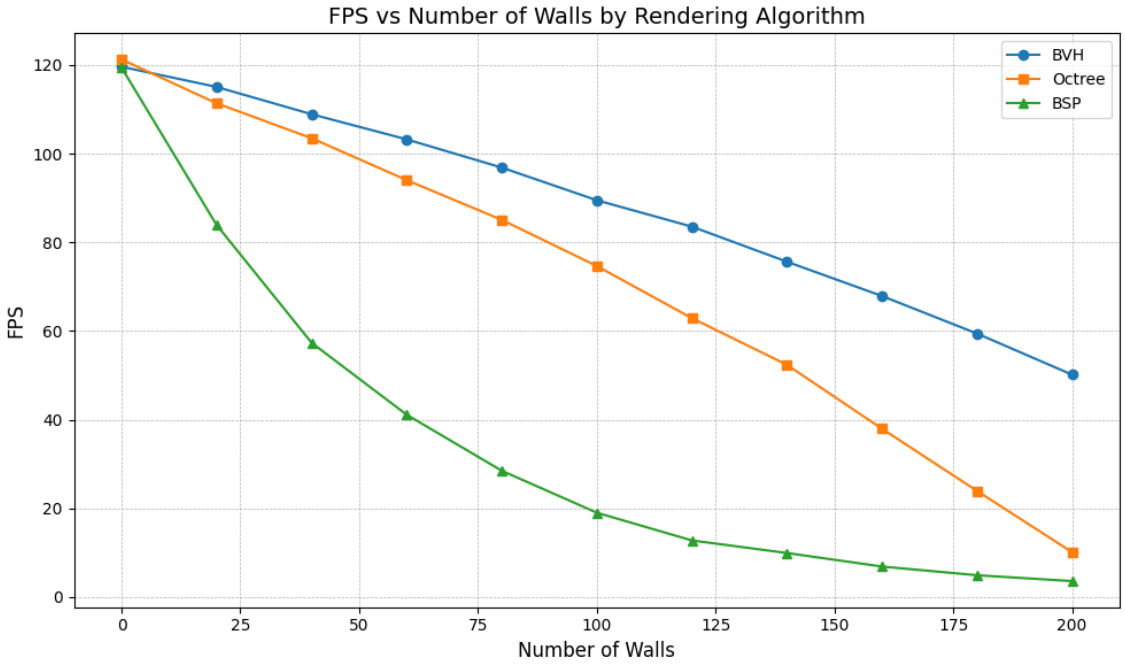

Figure 4.2: Frames per Second vs. Number of Walls

walls are split multiple times, leading to a rapid growth in the number of fragments and traversal steps. Consequently, rendering becomes increasingly inefficient as the number of walls increases.

In contrast, the Octree algorithm degrades more moderately, showing a curved, mildly nonlinear FPS decline. Octrees divide space uniformly into cubic regions, which is efficient in open or sparsely populated environments. However, in scenes where walls are densely clustered, the tree can over-subdivide space, leading to unnecessary traversal of empty or low-density regions. While it scales better than BSP, Octree performance still suffers when the spatial distribution of objects is uneven.

The BVH algorithm demonstrates the most stable performance across increasing wall counts. Its FPS decreases slowly in a smooth, nonlinear manner — typically resembling a shallow quadratic trend. BVH structures adaptively group nearby objects into tightly fitting bounding volumes, allowing large portions of the scene to be skipped during rendering via efficient hierarchical culling. Even as scene complexity grows, BVHs maintain relatively shallow tree depths and effective pruning, making them highly scalable and suitable for complex environments.

4.3 Memory Usage Comparison

To track the current memory usage during run-time, we used the `tracemalloc` package in Python. The package hooks into the Python memory allocator to trace all memory allocations made via Python objects. Like we did with pre-processing time, we ran the game 100 times across all the testing environments and rendering algorithm, taking the

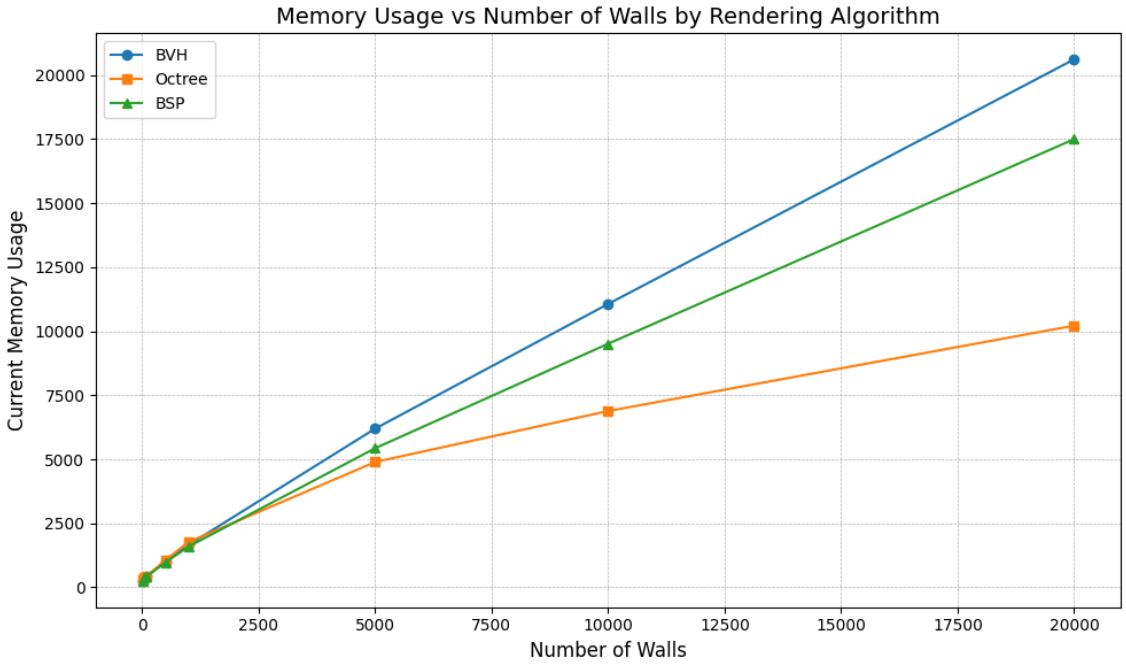

Figure 4.3: Memory Usage (KB) vs. Number of Walls

average memory usage across the 100 trials. See Figure 4.3 for a visual of how the memory usage scales as the number of walls increase.

As shown in the figure, current memory usage scales the fastest under the BVH algorithm, following a linear trend. BSP scales at a similar rate, following a sublinear trend. Octree scales the slowest, following a logarithmic trend.

4.3.1 Explanation

This pattern aligns well with theoretical expectations. BVH structures typically require more memory because each node stores a bounding volume and may include overlapping objects, especially in scenes where objects are not tightly clustered. This duplication leads to a linear growth in memory usage as the number of walls increases. In contrast, BSP trees are more memory-efficient, as they recursively divide space using planes and avoid duplicating objects unless they intersect a split plane. While still scaling linearly in practice, BSP trees generally result in fewer redundant nodes compared to BVHs. Octrees, however, subdivide space into eight child regions per node and often avoid allocating memory for empty or sparse regions. This leads to significantly slower growth in memory usage, especially in scenes where walls are small and well-distributed spatially. As a result, Octrees exhibited near-logarithmic memory scaling in my experiments, making them the most memory-efficient among the three under the tested conditions.

5 Related Work

5.1 Related Works

Spatial data structures such as Binary Space Partitioning (BSP), Bounding Volume Hierarchies (BVH), and Octrees have long been fundamental in accelerating 3D rendering, collision detection, and ray tracing.

The concept of Binary Space Partitioning was first introduced by Fuchs et al. [1] as a method for visible surface determination in 3D environments. Their work demonstrated the use of BSP trees for efficient rendering of scenes in computer graphics, particularly in indoor environments with static geometry. BSP became especially popular in early 3D games such as *DOOM* and *Quake*, where deterministic front-to-back rendering offered predictable performance [2].

Bounding Volume Hierarchies (BVH), introduced by Kay and Kajiya [3], are an alternative acceleration structure where objects are grouped hierarchically into bounding volumes (typically axis-aligned bounding boxes). BVH has become a standard approach in ray tracing engines due to its efficient construction and traversal times, as well as its adaptability to dynamic scenes [4]. Further optimizations such as Surface Area Heuristic (SAH) [5] have made BVH particularly effective for minimizing traversal costs in large scenes.

Octrees, first introduced by Meagher [6], are hierarchical spatial partitioning structures that recursively divide 3D space into eight octants. Their memory usage scales efficiently in sparse environments and they support fast spatial queries and culling. Octrees have been used in voxel rendering [7], point cloud processing [8], and physics engines where adaptive spatial resolution is beneficial. Compared to BVH, Octrees partition space rather than objects, which can reduce memory overhead in scenarios with high spatial sparsity.

Recent work by Pantaleoni [9] and others has explored hybrid approaches that combine voxel-based Octree data with ray tracing or rasterization backends for real-time rendering on modern GPUs. Additionally, real-time game engines such as Unreal Engine and Unity incorporate variants of these spatial structures for both static and dynamic objects, often leveraging GPU-accelerated traversal and culling [10].

In evaluating these data structures, researchers have explored trade-offs among memory consumption, preprocessing time, and real-time rendering performance. Wald et al. [11] conducted extensive benchmarks showing that BVH generally outperforms BSP in dynamic scenes due to faster construction times, while Octrees tend to outperform both in memory efficiency when dealing with large, sparsely populated 3D worlds.

This project draws from these foundational and modern works to compare BSP, BVH, and Octree implementations in a game rendering context, analyzing not only frame rate

performance but also preprocessing and memory scaling as the number of renderable walls increases.

6 Conclusions

This project set out to rigorously compare three widely used spatial partitioning techniques—Binary Space Partitioning (BSP), Bounding Volume Hierarchies (BVH), and Octrees—in the context of a custom-built 3D game engine. Our primary goal was to investigate how these structures scale along three key dimensions of performance: preprocessing time, real-time frame rate (FPS), and memory usage, as the number of renderable walls increases. While each of these algorithms has been studied in isolation, few works offer a direct, side-by-side evaluation within a controlled and unified rendering environment. This project aimed to fill that gap.

Our results revealed a consistent and meaningful set of trade-offs. In terms of frame rate performance, BVH outperformed both BSP and Octree across nearly all test cases. Its hierarchical grouping of objects and efficient traversal algorithms (such as those optimized with the Surface Area Heuristic) make BVH especially well-suited for dynamic scenes with varying object distributions. BSP followed closely behind, with relatively stable FPS that benefits from deterministic front-to-back rendering, but it struggled as the number of walls increased due to deeper tree traversal. Octrees yielded the lowest frame rates overall, particularly in densely packed scenes, where their spatial resolution and subdivision logic introduced higher traversal overhead.

When evaluating preprocessing time, BVH again demonstrated superior efficiency. Its top-down construction approach, which recursively splits object groups using simple bounding volumes, scaled well with scene complexity and remained fast even at high object counts. Octrees occupied a middle ground: their spatial subdivision allowed for fast insertion when walls were distributed evenly but incurred greater overhead in unbalanced configurations. BSP, however, had the slowest preprocessing times, owing to the recursive selection of splitting planes and the need to partition geometry across those planes—an operation that becomes increasingly expensive as scene density grows.

Perhaps the most surprising and significant finding came from our analysis of memory usage. As anticipated, Octrees displayed the most efficient memory scaling. Due to their recursive subdivision of space rather than objects, and the fact that empty regions remain uninstantiated, Octrees achieved sublinear growth in memory usage with respect to wall count in many scenarios. BVH and BSP, in contrast, scaled linearly or worse, with BVH maintaining large bounding volumes and BSP storing redundant geometry along splitting planes. This highlights Octrees as an excellent candidate for memory-constrained environments, particularly in large, sparse 3D worlds or voxel-based applications.

Taken together, these results reinforce a fundamental truth in graphics systems design: there is no universally optimal structure. Each spatial partitioning scheme comes with its own strengths and limitations. BVH excels in runtime performance and preprocessing speed, making it ideal for general-purpose rendering and real-time engines. BSP offers

simplicity and predictability, particularly valuable for static geometry or games requiring deterministic sorting. Octrees scale best in memory, providing a compelling solution for environments with massive spatial extent or sparse object distributions.

By implementing all three structures in the same engine and subjecting them to the same experimental conditions, this project offers a rare apples-to-apples comparison that helps clarify when and why each algorithm should be used. These insights can guide engine developers, graphics researchers, and systems designers in selecting the spatial data structure best aligned with the constraints and priorities of their application.

7 Future Work

7.1 Future Work

While this project provides a robust comparative analysis of BSP, BVH, and Octree spatial structures, several avenues remain open for future exploration. First, our current implementations assume static scenes and fixed wall geometries. Extending the comparison to dynamic environments—where objects move, appear, or disappear—would provide valuable insight into the adaptability of each structure. In particular, BVH and Octree are known to support dynamic updates more efficiently than BSP, and benchmarking this behavior could clarify their respective trade-offs in games with frequent object motion or procedural generation.

Second, all measurements were taken in a CPU-bound rendering context. Incorporating GPU-accelerated traversal or hybrid CPU-GPU pipelines could yield different performance characteristics, especially for BVH, which is widely used in real-time ray tracing engines on modern hardware. Exploring how each structure performs under hardware-accelerated conditions would make this work more applicable to high-performance graphics applications.

Third, memory usage was profiled from a high-level perspective using runtime metrics. A deeper analysis of memory allocation patterns—such as cache coherence, fragmentation, and memory locality—could reveal additional optimization opportunities, particularly for Octrees, whose spatial layout could benefit from cache-aware node ordering.

Finally, future work could explore hybrid or adaptive spatial structures. For instance, some game engines use Octrees for broad-phase culling and then fall back on BVH for fine-grained rendering. Investigating such hybrid combinations could offer best-of-both-worlds performance and inspire more flexible engine architectures.

By addressing these extensions, future research can build on the foundation laid here to provide a more complete understanding of spatial partitioning in both traditional rasterization and modern rendering pipelines.

Bibliography

[1] Henry Fuchs, Zvi M. Kedem, and Brian F. Naylor. “On Visible Surface Generation by A Priori Tree Structures”. In: *Proceedings of the 7th annual conference on Computer graphics and interactive techniques (SIGGRAPH)*. ACM, 1980, pp. 124–133.

[2] Michael Abrash. *Michael Abrash's Graphics Programming Black Book*. Coriolis Group Books, 1997.

[3] Timothy L. Kay and James T. Kajiya. “Ray tracing complex scenes”. In: *Proceedings of the 13th annual conference on Computer graphics and interactive techniques (SIGGRAPH)*. ACM, 1986, pp. 269–278.

[4] Ingo Wald, Solomon Boulos, and Peter Shirley. “Ray tracing animated scenes using coherent grid traversal”. In: *ACM Transactions on Graphics (TOG)* 26.1 (2007), pp. 1–10.

[5] Jeffrey Goldsmith and John Salmon. “Automatic creation of object hierarchies for ray tracing”. In: *IEEE Computer Graphics and Applications* 7.5 (1987), pp. 14–20.

[6] Donald Meagher. “Geometric modeling using octree encoding”. In: *Computer Graphics and Image Processing*. Vol. 19. 2. Elsevier, 1982, pp. 129–147.

[7] Samuli Laine and Tero Karras. “Efficient sparse voxel octrees”. In: *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D)*. ACM, 2010, pp. 55–63.

[8] Jan Elseberg, Dorit Borrmann, and Andreas Nüchter. “Comparison of nearest-neighbor-search strategies and implementations for efficient shape registration”. In: *Journal of Software Engineering for Robotics* 4.1 (2013), pp. 2–12.

[9] Jacopo Pantaleoni. “VoxelPipe: A programmable pipeline for 3D voxelization”. In: *Proceedings of the Conference on High Performance Graphics*. ACM, 2011, pp. 99–106.

[10] GameDev Research Collective. *Real-Time Rendering Techniques in Modern Game Engines*. https://research.gamedev.net/real-time-rendering-techniques-survey. 2021.

[11] Ingo Wald et al. “State of the art in ray tracing animated scenes”. In: *Computer Graphics Forum* 28.6 (2009), pp. 1691–1722.

A About The Game

A.1 The Engine

The game engine is composed of three main components: the renderer, the game logic handler, and NPC interaction handler. When the game is running, the engine initializes the NPC objects and the renderer. The renderer begins by building the data structure that holds the information of the walls (this structure depends on which algorithm we follow). While the game is running, the game logic handler handles user interaction (keyboard presses, mouse clicks) and updates our player's position and camera angle accordingly. Then the new frame is rendered given the player's position and the walls, following the traversal method of the current rendering algorithm. This process repeats until the game ends.

A.2 Scene Generation

Players can create custom "levels" using the level editor. The level editor is a 30x30 grid that allows the player to draw in walls along the edges. The grid will get converted into a JSON file that contains the information of the walls position and texture. This JSON file is read when the game is run, and the wall information is then fed into the renderer as described above.

For benchmarking, a script that randomly generates a scene with a specified number of walls was used to create environments with 10, 50, 100, 500 walls, etc...

A.3 The Game

The game is a first person 2.5D game in which the player travels around the world and collects dog treats. Players gain points by feeding dogs around the world these dog treats. The enemy, a cat, will chase you and attack you if you get too close. You can stun them by clicking on them when they are nearby.

There is a minimap on the top left of the screen, which shows your position and what walls are being rendered.

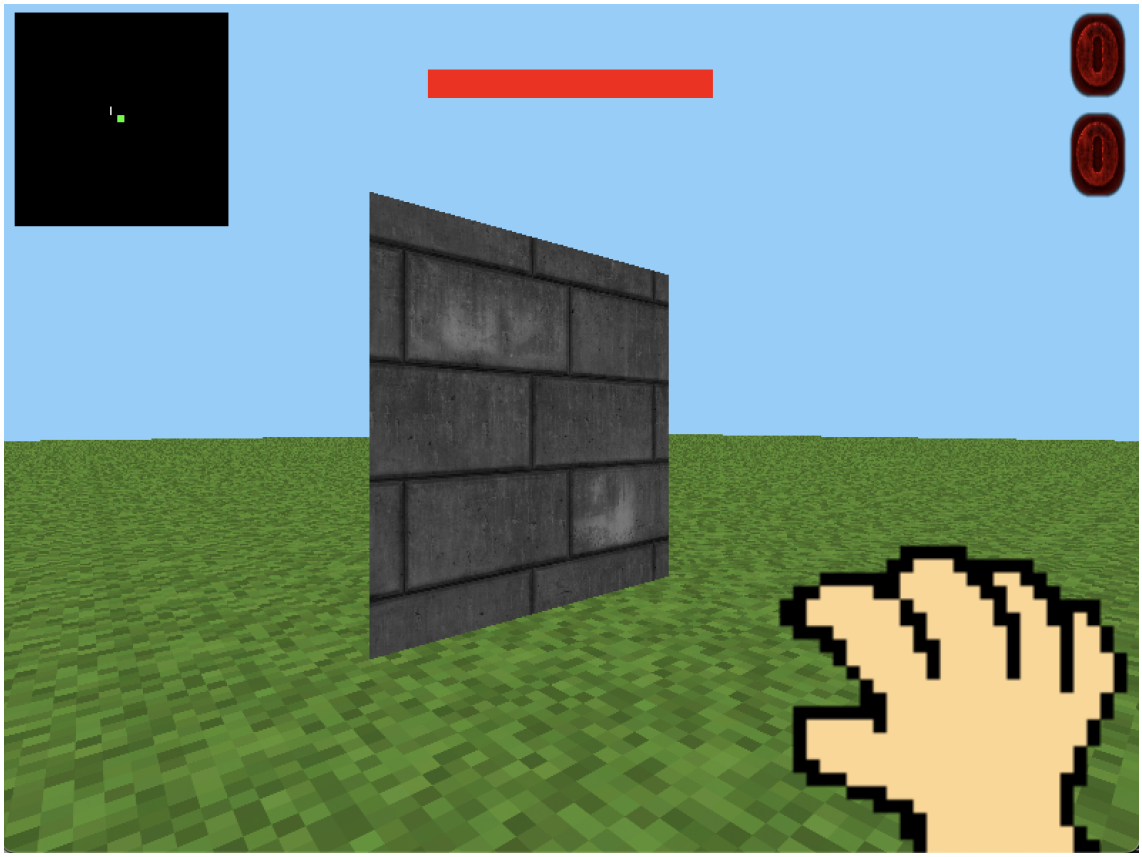

Figure A.1: Wall

A.4 Tree Initialization

A.4.1 BSP

The BSPNode class represents a node in a Binary Space Partitioning (BSP) tree. Each node contains a wall (a dividing plane or polygon), an aabb (Axis-Aligned Bounding Box) that encloses the walls, and two child nodes, front and back, which represent the partitions of space in front of and behind the dividing wall.

A.4.2 BVH

The BVHNode class represents a node in a Bounding Volume Hierarchy (BVH) tree. Each node contains a list of walls (if it is a leaf node), an aabb (Axis-Aligned Bounding Box) that encloses the walls, and two child nodes, left and right, which represent the partitions of walls in the hierarchy. Internal nodes do not store walls directly but instead divide the space into two subregions.

The build_tree method constructs the BVH tree recursively. It begins by computing the AABB for the given walls. If the number of walls is small (e.g., 2 or fewer), a leaf node is created with the walls and their AABB. Otherwise, the method determines the longest axis of the AABB (X, Y, or Z) and sorts the walls by their centroids along that axis. The walls are then split into two groups, left_walls and right_walls, at the midpoint. The method recursively builds the left and right child nodes for these partitions. This process continues until all walls are processed, resulting in a tree structure that organizes walls hierarchically for efficient spatial queries and rendering.

A.4.3 Octrees

The OctreeNode class represents a node in an octree. Each node contains a list of walls (if it is a leaf node), an aabb (Axis-Aligned Bounding Box) that encloses the walls, and eight children nodes, corresponding to the eight octants of the 3D space. Internal nodes do not store walls directly but instead divide the space into smaller regions.

The build_tree method constructs the octree recursively. It starts by computing the AABB for the given walls. If the number of walls is small (e.g., 2 or fewer) or the AABB size is below a threshold, a leaf node is created with the walls and their AABB. Otherwise, the method calculates the center of the AABB and divides the walls into eight octants based on their centroids. Each wall is assigned to an octant based on its position relative to the center. If splitting is meaningful (i.e., at least one octant has fewer walls), the method recursively builds the child nodes for each octant. This process continues until all walls are processed, resulting in a hierarchical structure that efficiently organizes walls in 3D space for spatial queries and rendering.