MINI-DOOM

Computer Graphics (E016712A)

GHENT UNIVERSITY

Elias Nijs & René Van Der Schueren

Academic Year 2024/2025

May 2025

Contents

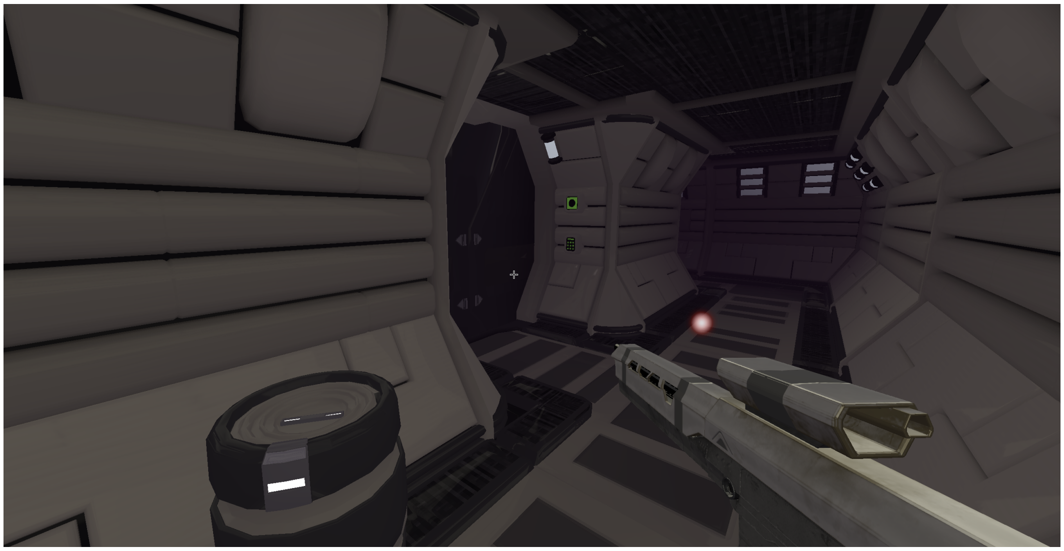

Figure 1: Gameplay screenshot of our Doom-inspired game implemented in Three.js.

1 Introduction

Doom is a game that has earned a lasting legacy. Even more, its enduring popularity and active fanbase have fostered a tradition of recreating the game in various forms ever since the open-sourcing of the original source code. In a similar vein, we develop a simplified iteration, *MINI-DOOM*.

1.1 Tech Choices

Our project uses Three.js, a lightweight JavaScript library that abstracts WebGL for accessible 3D graphics programming in browsers. It was a good choice for its robust ecosystem, documentation, and optimized web rendering performance. Development is powered by Vite for fast server capabilities, and we implemented the codebase in TypeScript to benefit from static typing, which improves code reliability and facilitates better team collaboration.

1.2 Gameplay Overview

Our game is a first-person shooter inspired by the classic Doom. Players navigate through procedurally generated mazes, interact with doors, follow path markers, and encounter various props while trying to reach their destination. Players can also shoot their gun to open doors, adding an interactive element to the navigation mechanics.

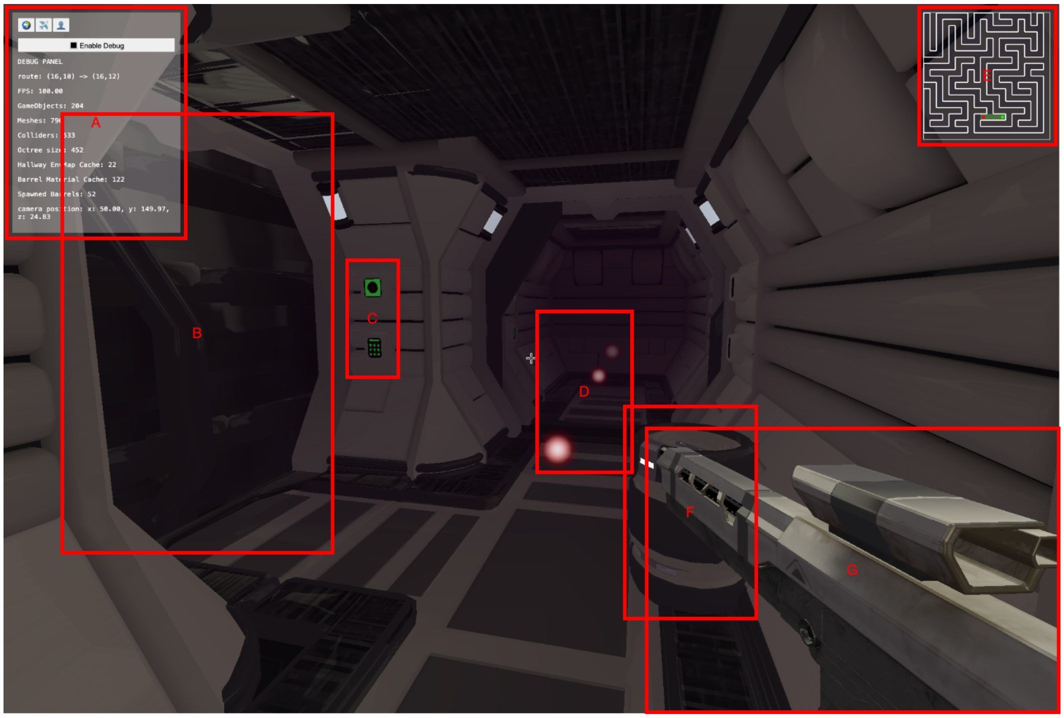

Figure 2: Gameplay screenshot showing key elements: (A) Debug menu for adjusting game parameters, (B) Closed door with its control panel (C), (D) Path markers guiding the player through an open door, (E) Mini-map displaying player position and path to destination, (F) Barrel spawned as a random prop, and (G) The player's weapon.

Figure 2 illustrates the main gameplay elements:

- **A:** Debug menu that allows changing camera settings, enabling debug mode, and displays game engine statistics.
- **B:** A door that was previously opened, allowing passage through the maze.
- **C:** Door control panel that can be activated by shooting it with the player's weapon.
- **D:** Path markers showing the optimal route to the destination, guiding the player through an open door.
- **E:** Mini-map displaying the player's current position in red, and the path to the destination, marked in green.
- **F:** Barrel that was procedurally spawned as a random prop in the environment.
- **G:** The player's gun.

2 Game Engine

Before diving into specific features, we'll provide an overview of our custom game engine architecture that powers the entire application.

2.1 Core Architecture

Our game engine follows a component-based design pattern, with a central state management system that coordinates all game elements. The engine is structured around these key components:

- `GameObject`: An abstract base class that serves as the foundation for all entities in the game world. Each game object automatically registers itself with the state management system upon creation and implements lifecycle methods such as `animate` (called every frame), `cleanup`, and `destroy`.
- `State`: The central management class that maintains references to all active game objects, handles scene configuration, manages the camera system, and coordinates the physics simulation. It provides methods for registering and unregistering game objects, finding objects by type, toggling debug visualization, ...
- `Physics`: A collision detection and resolution system that uses axis-aligned bounding boxes (AABBs) to represent object boundaries. The physics system calculates collision corrections and applies them to maintain proper object separation. For efficient collision detection, we implemented an octree-based spatial partitioning system (explained in detail in the Space Partitioning section), which significantly reduces the number of collision checks by organizing objects hierarchically based on their spatial location.

2.2 Game Loop

The engine implements a standard game loop that:

1. Calls the `animate` method on the state, which in turn calls all it on all registered game objects.
2. Updates the physics system and rebuilds the dynamic octree.
3. Renders the scene.

This architecture provides a solid foundation for implementing the game-specific features described in the following sections.

3 Maze Generation

First, we will start by explaining our maze generation algorithm, which forms the foundation of our game's level design.

3.1 Implementation

The maze generation system is implemented as a modular component in the utils directory. It provides function for creating the abstract maze structure but not for converting it to the physical 3D environment. This separation of concerns allows the maze logic to be tested independently from the game rendering system while maintaining a clean integration between the two.

3.1.1 Data Structures

The maze is represented by a grid of cells, where each cell contains information about its walls and visited state:

- `Cell`: A type representing a single cell in the maze with properties for walls (north, east, south, west) and a visited flag.
- `Grid`: A type containing an array of cells and dimensions (number of rows and columns).
- `Pos`: A type alias for a position in the grid, represented as [row, column].

3.1.2 Generation Algorithm

The maze generation follows these steps:

1. Initialize a grid where all cells have all four walls intact and are marked as unvisited.
2. Start at a cell (typically [0,0]) and mark it as visited.
3. Push the starting cell onto a stack to track the path.
4. While the stack is not empty:
   (a) Get the current cell from the top of the stack.
   (b) If the current cell has any unvisited neighbors:
      i. Choose one randomly.
      ii. Remove the wall between the current cell and the chosen neighbor.
      iii. Mark the neighbor as visited.
      iv. Push the neighbor onto the stack.
   (c) If there are no unvisited neighbors, pop the current cell from the stack (backtrack).

This algorithm ensures that every cell in the maze is reachable from any other cell, creating a perfect maze with exactly one path between any two points.

3.1.3 Upscaling

The implementation includes an optional upscaling feature that doubles the effective resolution of the maze by inserting buffer cells between the original cells. This creates a more visually appealing maze with wider corridors while maintaining the logical structure of the original maze. Additionally, the upscaling process leaves room between parallel hallways, providing space for game elements such as doors to animate into when opened, enhancing the interactive experience without causing clipping or collision issues. Figure 3 illustrates the difference between a maze without spacing and one with spacing, clearly showing how the upscaling creates buffer zones between parallel corridors.

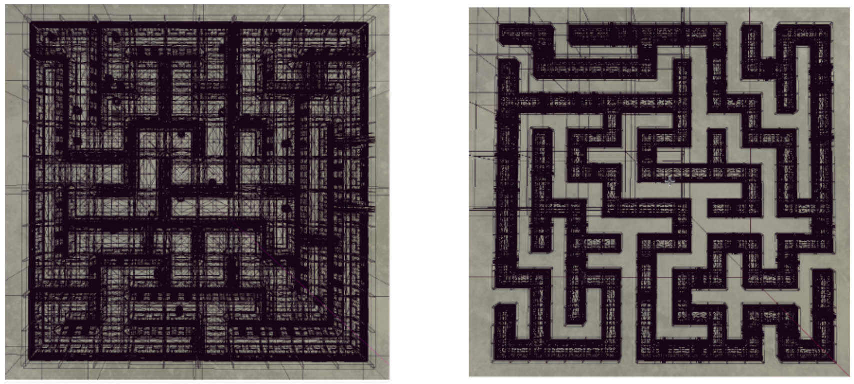

Figure 3: Comparison of a 10x10 maze: left without spacing, right with spacing, viewed from the top in debug mode. Notice the extra spacing making sure that parallel hallways do not touch.

3.1.4 Pathfinding

The implementation features an efficient A* pathfinding algorithm that calculates optimal routes between any two points in the maze. This algorithm employs a Manhattan distance heuristic—particularly suitable for grid-based movement—and accounts for walls when evaluating potential paths. The pathfinding system provides visual guidance for players through path markers that highlight the shortest route to objectives. The algorithm maintains separate data structures for tracking both the cost of the path so far (g-score) and the estimated total cost to the destination (f-score), ensuring optimal path discovery even in complex maze configurations.

4 3D World Organization

Our 3D world uses a modular approach with standardized hallway units as building blocks, optimizing both performance and visual quality.

The world is built on a grid of uniform hallway segments, each occupying one grid cell with consistent dimensions. Hallways have identical outer dimensions with internal variations (straight, corner, junction) and are positioned using precise grid coordinates. While hallway elements are hierarchically organized in the Three.js scene, our GameObject system maintains a flat structure for efficient game logic.

Our system loads all hallway variants from a single GLB file and clones meshes rather than creating new geometry for each instance. Materials and environment maps are cached to prevent redundant creation. This approach significantly reduces memory usage and draw calls, enabling efficient rendering of large mazes.

The player exists as a separate entity from the environment structure, comprising a first-person camera at eye level, a weapon model in the lower view portion, collision geometry for physical interaction, and independent movement and interaction logic. The player navigates through the hallway grid using keyboard and mouse controls, with collision detection preventing movement through walls and other obstacles. The camera's position and orientation update in real-time based on player input, creating a smooth first-person experience.

Our world features several dynamic elements: sliding doors that update collision geometry in real-time; interactive door control panels; sprite-based path markers; procedurally distributed props; and temporary bullet impact markers that fade over time to maintain performance while providing visual feedback (see Figure 4). All these elements—doors, control panels, props, and other interactive objects—are implemented as children of their respective hallway parts in the scene hierarchy, maintaining a clean organizational structure while preserving the flat GameObject system for game logic.

Figure 4: Bullet impact markers on walls that gradually fade over time and are eventually deleted to conserve resources, providing visual feedback of weapon impacts while maintaining performance.

5 User Controls and Movement

Our Doom clone implements a classic first-person shooter control scheme. Player movement is controlled using the WASD keys, with W and S for forward and backward movement, and A and D for strafing left and right. Mouse input controls the camera rotation, allowing players to look around the environment with configurable sensitivity parameters.

The movement system applies the player's directional input to a normalized vector, which is then rotated according to the player's current view direction. This creates the expected behavior where pressing W always moves the player forward relative to their view. Movement velocity is controlled by adjustable speed parameters and is framerate-independent thanks to delta time scaling.Collision detection prevents the player from moving through walls, this is explained later.

The game also features immersive visual feedback during movement, with a weapon bob animation that simulates the natural motion of carrying a weapon while walking. This is achieved by applying sinusoidal oscillations to the weapon position based on the player's movement state. Additionally, the weapon model automatically lowers when the player approaches a wall, providing a realistic response to close-proximity obstacles and preventing clipping issues.

6 Rendering Components

6.1 Lighting

The game uses a global ambient light for base illumination. Hallway light fixtures use diffuse materials to simulate lights, providing something that looks like a light, even in the reflections, without requiring expensive dynamic lighting calculations. Figure 5 shows an example of these light fixtures in the game.

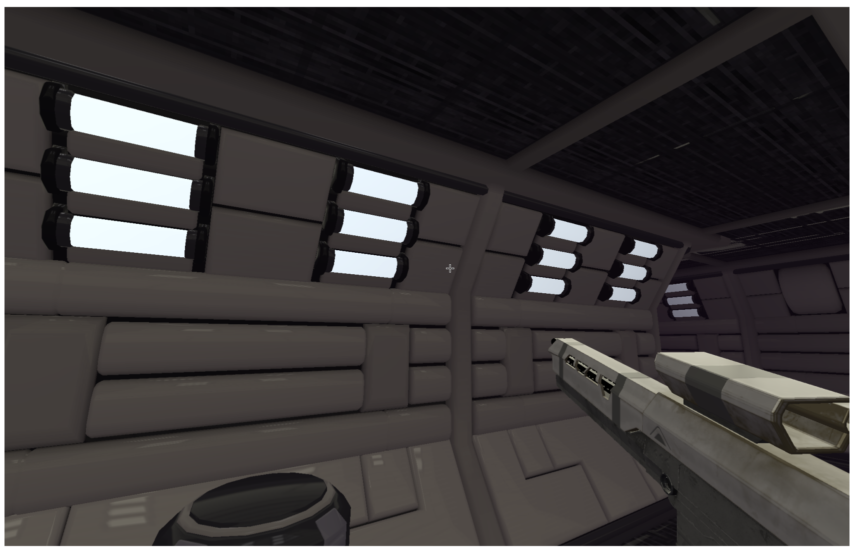

Figure 5: Light fixtures in hallways providing illumination.

6.2 Fog

A simple exponential fog is used to create an atmosphere in the game. Figure 6 shows an example of fog in the game.

6.3 Hallway Materials and Reflections

Each hallway segment uses a standardized material system composed of multiple texture maps: diffuse maps for base color, normal maps for surface detail, metalness maps to define reflective characteristics, roughness values to control reflection blurriness, and environment maps to simulate reflections from the surrounding scene.

At launch, we generate a unique environment map for every hallway type and orientation using a cube camera. These maps provide dynamic yet efficient reflections by precomputing

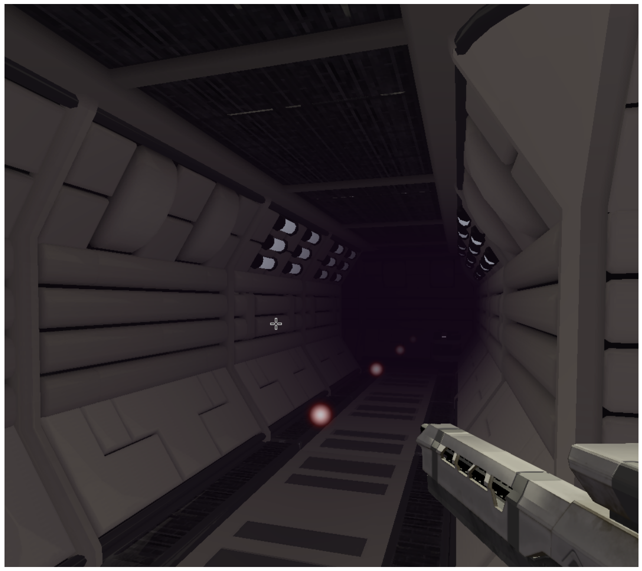

Figure 6: Example of fog in the game. The end of this hallway can hardly be seen.

the surrounding lighting environment. A material cache ensures that hallway segments with the same type and orientation reuse the same material and environment map. This significantly reduces memory overhead and improves rendering performance.

6.4 Prop Materials and Inheritance

Props embedded within hallways use a simplified material configuration. They rely on basic diffuse texture maps for color, global metalness and roughness values shared across all props, and the inherited environment map from their parent hallway segment.

This approach eliminates the need for generating unique environment maps for each prop. Instead, props reflect their environment based on the surrounding hallway's precomputed reflections. As with hallways, a dedicated material cache prevents duplication and reduces draw calls, contributing to performance gains.

6.5 Reflection Rendering Strategy

Reflections in the scene are implemented via the pre-generated environment maps. These maps accurately reflect light sources—such as hallway fixtures—on nearby surfaces. Figure 7 demonstrates how these reflections appear on both walls and props, visible as horizontal highlights from light sources.

By relying on cached, precomputed reflections, we avoid the high computational cost of real-time reflection techniques. Reflection sharpness and intensity are controlled through each material's metalness and roughness settings. This component-based, cache-efficient strategy ensures consistent visual fidelity and optimized performance, even on lower-end systems.

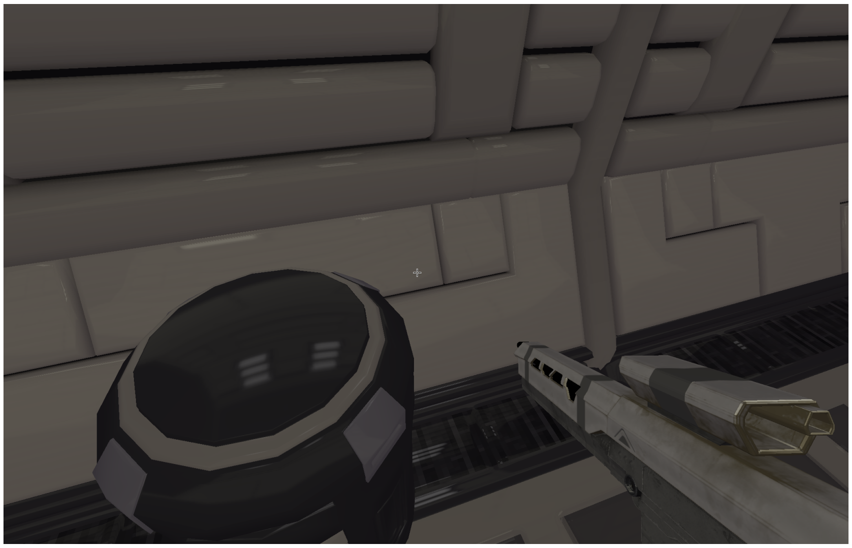

Figure 7: Reflections of the lights shown in Figure 5 on walls and props using environment maps. The horizontal white lines indicate reflected light.

6.6 Frustrum Culling

We use Three.js's default frustum culling to improve performance by automatically skipping objects outside the camera's view. This built-in feature checks each object's bounding volume against the camera's frustum and omits rendering if it's not visible, reducing unnecessary GPU processing.

7 Collision Detection Using Octree

For efficient collision detection, we employ an octree-based spatial partitioning system. This significantly reduces the computational complexity of detecting object interactions.

7.1 Octree Implementation

Our implementation uses a standard octree data structure with some optimizations for performance and memory efficiency.

The octree stores elements in a backing list and the nodes of the octree store indices into this list. This allows multiple octree nodes to reference the same element without duplication. This is required as large objects can span multiple partitions.

We use bit patterns to index the octree. This leverages the 1-1 mapping between 3 bits and 8 octants (each bit corresponds to an axis).

To prevent excessive subdivision in dense areas, we add a depth limit for the octree. Excessive subdivision leads to diminishing returns in performance and increased memory usage.

We implement removal from the octtree by marking elements as dead in the backing list rather than physically removing them from the nodes. This approach avoids costly array operations and simplifies the tree maintenance logic. Dead elements are discarded in the next rebuild.

Other ideas to optimize the octtree include:

1. Trying to exploit caching behavior more aggresively
2. Writing an optimized verion in a language such as C
3. At the moment there is a lot of type checking because we are using typescript. In a more efficiency-focussed iteration, we would remove these as certain configurations are impossible by construction and thus don't need to be checked.

7.1.1 Computational Complexity

An octree has the following scaling behavior:

- **Construction**: $O(n \log n)$, adding an element is $O(\log n)$ and we add n elements
- **Query**: $O(\log n + k)$, finding the element is $O(\log n)$ and delivers a subset of size k which needs to be checked linearly.

The practical benefits of our octree implementation are quantified in the benchmarks presented in Section 8.

7.2 Engine Integration

Our engine maintains two separate octrees: one for static objects (walls, floors, stationary props) and another for dynamic objects (player, moving doors). The static tree only builds on level-load while the dynamic tree updates every frame. This reduces the cost of rebuilding.

Game objects must manage their own colliders. They must register and unregister them with the state management system. This registration process includes a flag to specify whether the collider should be added to the static or dynamic tree. They are also responsible for resolving their own collisions.

For debugging purposes, we implemented a visualization system that renders the octree structure with color coding based on depth. This is visualised in Figure 8.

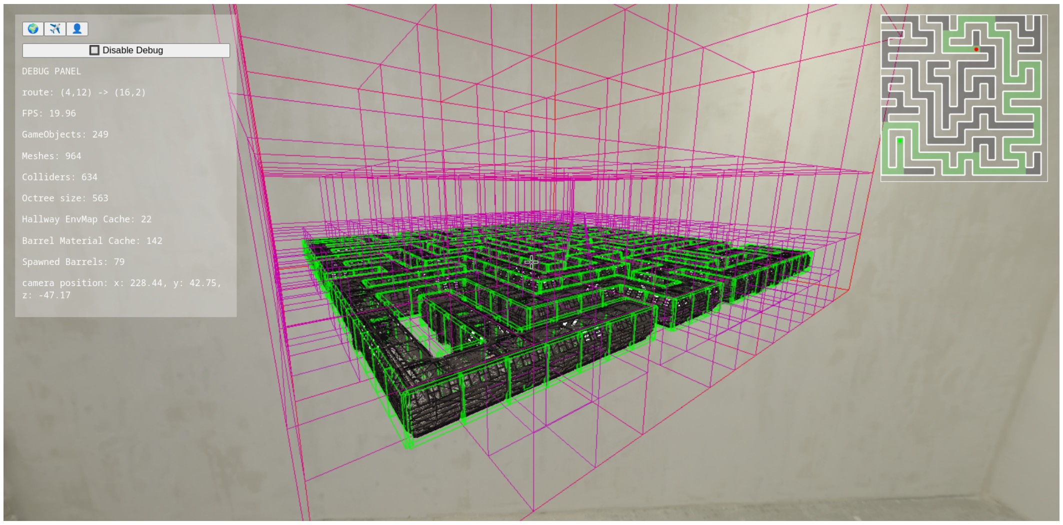

Figure 8: Screenshot of our debugging view showing the octtree.

7.3 Collision Resolution

Since our game world is organized on a grid, we use axis-aligned bounding boxes (AABBs) as colliders. In order to resolve possible collisions for a given object, we first query both the static and dynamic octrees to retrieve all colliders that share the same spatial partition . We then determine the amount of overlap between the colliding objects. The system then implements minimum penetration axis selection by comparing overlaps in the X, Y, and Z directions and only applies that correction (separating axis theorem). Multiple simultaneous collisions are handled via a normalised sum. This could probably be better dealt with by using a component-wise max. However, we decided to keep the current implementation as it was already working as intended.

7.4 Octtree, KD-Tree or BSP-Tree?

Octrees excel in 3D applications with uniform space subdivision like our grid based game, offering efficient spatial queries for regularly distributed data. Their main limitation is their rigid structure, i.e. always splitting into eight equal octants. This is unlike kd-trees which adapt splitting planes to data distribution or BSP-trees which allow arbitrary splitting. Avoid octrees for dimensions beyond 3D, highly skewed data distributions, or complex geometries where kd-trees (for irregular data) or BSP-trees (for arbitrary splitting planes) would perform better.

8 Benchmarking

To assess the performance and efficiency of our implementation, we conducted a series of controlled tests focusing on key areas that affect gameplay and user experience. These benchmarks provide quantitative data on how our design choices impact performance as scale increases, helping to identify optimization opportunities and hardware requirements.

8.1 Maze Size Performance Analysis

To evaluate the scalability of our implementation, we conducted performance tests with mazes of increasing size. All tests were performed on a MacBook Air (2022) with an Apple M2 chip, 8-core CPU, 8-core GPU, and 24GB of unified memory. The performance was measured in frames per second (FPS) while maintaining all graphical settings and game features at consistent levels.

Table 1 presents the relationship between maze dimensions, scene complexity (measured by the number of game objects and meshes), and rendering performance. The maze dimensions represent the logical grid size before upscaling, while the total tiles indicate the actual number of navigable cells in the generated maze.

Maze Size	Total Tiles	Game Objects	Mesh Count	FPS
10×10	100	204	697	100
20×20	400	904	2,844	100
40×40	1,600	3,204	11,595	50
80×80	6,400	12,804	46,477	10
160×160	25,600	51,204	187,113	2.5

Table 1: Performance metrics for different maze sizes showing the relationship between maze dimensions, scene complexity, and frame rate.

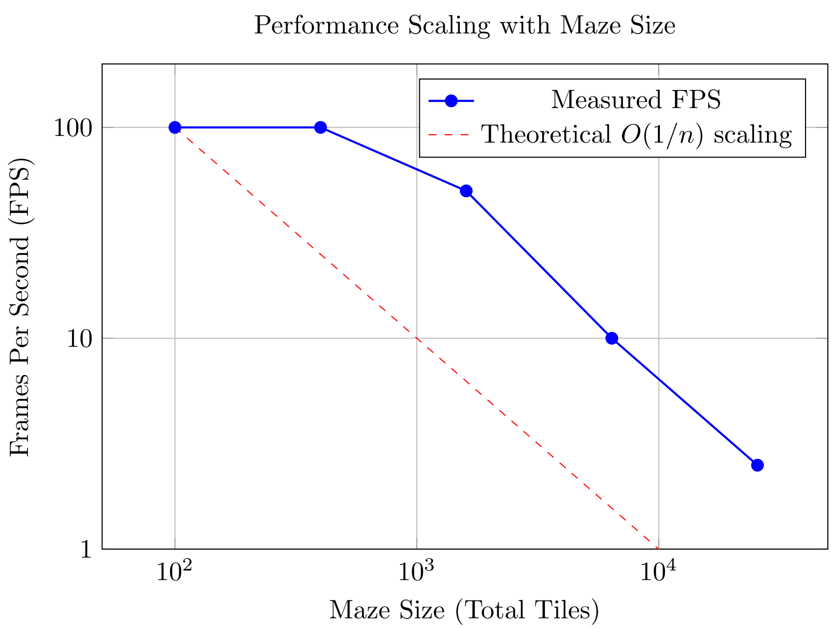

Figure 9: Log-log plot showing the relationship between maze size (total tiles) and performance (FPS). The dashed red line represents theoretical inverse scaling, demonstrating how our implementation closely follows expected performance characteristics as scene complexity increases.

As evident from the data, our implementation maintains an optimal frame rate of 100 FPS for mazes up to 20×20 (400 tiles), demonstrating that the game is well-optimized for small to medium-sized environments. Note that the 100 FPS ceiling is due to the testing monitor's maximum refresh rate; the actual performance for smaller mazes may exceed this value. Performance begins to degrade at 40×40 (1,600 tiles), dropping to 50 FPS, which is still playable but shows the increasing computational demands.

At the extreme end, the 160×160 maze (containing 25,600 tiles and over 187,000 meshes) pushes the hardware to its limits, resulting in only 5 FPS. While not playable, this test demonstrates the upper bounds of our implementation on consumer hardware and provides valuable insights for future optimization efforts.

8.2 Octree Performance Analysis

To evaluate the efficiency of our octree, we conduct a series of benchmarks.

8.2.1 Octree Query Performance

We first focus on query-time and compare it with linear search. For this test, we distribute elements with varyingsize bouding boxes (1-5 units) uniformly in the world (1000 units per side). For the queries, we generate 100 random query regions with varying sizes (20-50 units).

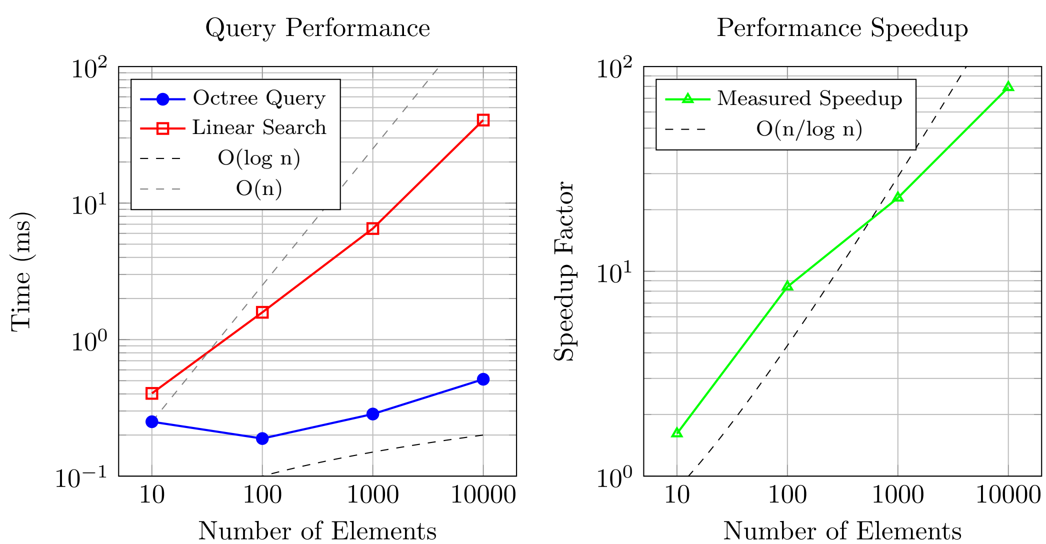

Figure 10: Octree performance comparison. Left: Log-log plot of query time for octree vs. linear search, showing how octree performance follows logarithmic scaling while linear search scales linearly. Right: Speedup factor of octree-based queries compared to linear search, demonstrating efficiency gains that grow with scene complexity.

As shown in Figure 10, octree query time scales approximately logarithmically with the number of elements, while linear search exhibits the expected linear scaling.

These performance characteristics enable our game to maintain interactive frame rates even in complex environments with numerous dynamic objects, confirming that the octree implementation successfully addresses the computational challenges of collision detection at scale.

8.2.2 Octree Insertion Performance

We also benchmark the insertion performance. For this test, we vary the amount of elements inserted (10 to 100,000), the capacity (4,8, or 16) and max-depth (4,8, or 12), and we give the elements a size bewteen 1 and 5 units and distribute them uniformly.

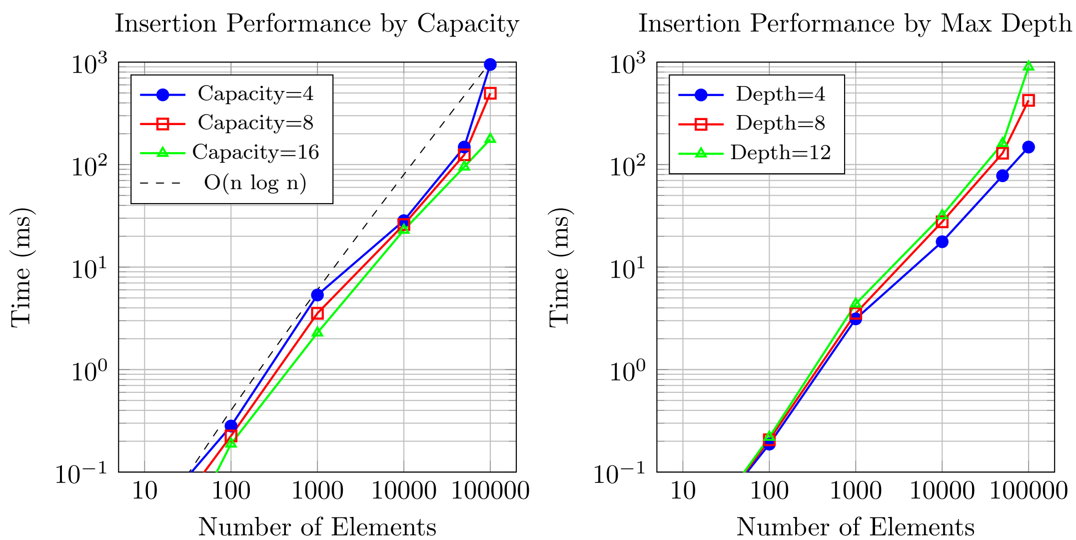

Figure 11: Octree insertion performance analysis. Left: Log-log plot showing how node capacity affects insertion time (with fixed max depth=8, world size=1000). Right: Log-log plot showing how maximum tree depth affects insertion time (with fixed capacity=8, world size=1000).

Our benchmarks reveal the following insights:

- **Scaling Behavior**: Insertion time scales as O(n log n) with the number of elements.
- **Capacity Impact**: Higher capacity values (16) improved insertion performance up to 5.3× compared to lower values (4) at 100,000 elements. Higher capacities reduce tree subdivisions and overhead.
- **Max Depth Effect**: Shallow trees (max_depth=4) outperformed deeper trees (max_depth=12) by up to 6× at large element counts. Excessive subdivision degrades performance without proportional benefits.

9 Conclusion

In this project, we successfully developed a Doom-inspired game using Three.js that implements a wide range of computer graphics techniques. Our maze generation system creates complex, playable environments with optimal pathfinding capabilities, while our octree-based spatial partitioning system efficiently handles collision detection even in large scenes. Performance benchmarks demonstrate that our implementation achieves interactive frame rates for reasonably sized mazes, with the octree providing up to 79× speedup compared to naive collision detection approaches.

The modular design of our game engine allows for easy extension and modification, with a clear separation between game logic and rendering components. By implementing advanced features such as dynamic lighting, material properties, and physics-based interactions, we've created an immersive first-person experience that captures the essence of classic FPS games while showcasing modern web-based rendering capabilities.

Throughout development, we faced and overcame several challenges, particularly in optimizing performance for larger environments and ensuring consistent physics behavior. Future work could focus on further optimizing the rendering pipeline, implementing more advanced lighting techniques, and expanding gameplay elements with additional enemy types and weapons.

10 Art Credits

While the focus of this project was on the technical implementation of computer graphics concepts, we utilized several third-party 3D models and textures to enhance the visual quality of our game. All assets were used under appropriate licenses, and modifications were made to better suit our game's aesthetic and technical requirements.

- **Hallway Modules**: The modular hallway system was based on assets from "Spaceship Modules" by ThisIsBranden, available at `https://thisisbranden.itch.io/spaceship-modules`.
- **Environmental Props**: Various props such as barrels and crates were sourced from the "Sci-Fi Assets Pack" available on TurboSquid at `https://www.turbosquid.com/3d-models/sci-fi-assets-model-1876664`.
- **Weapon Model**: The player's gun was adapted from the "Sci-Fi DMR" model available on TurboSquid at `https://www.turbosquid.com/3d-models/3d-scifi-dmr-model-1983451`.

Small modifications were made to these models using Blender. Texture adjustments were performed using Krita. This was done to maintain a consistent visual style throughout the game environment.

11 Acknowledgment

During this project we made use of AI tools to help us write the report and more specifically to help us formulate our report in proper academic English. We also utilized AI assistance during code development to speed up certain aspects of development.