Simple Doom Game Clone in Three.js

Computer Graphics (E016712A)

GHENT UNIVERSITY

Elias Nijs & René Van Der Schueren

Academic Year 2024/2025

May 2025

Contents

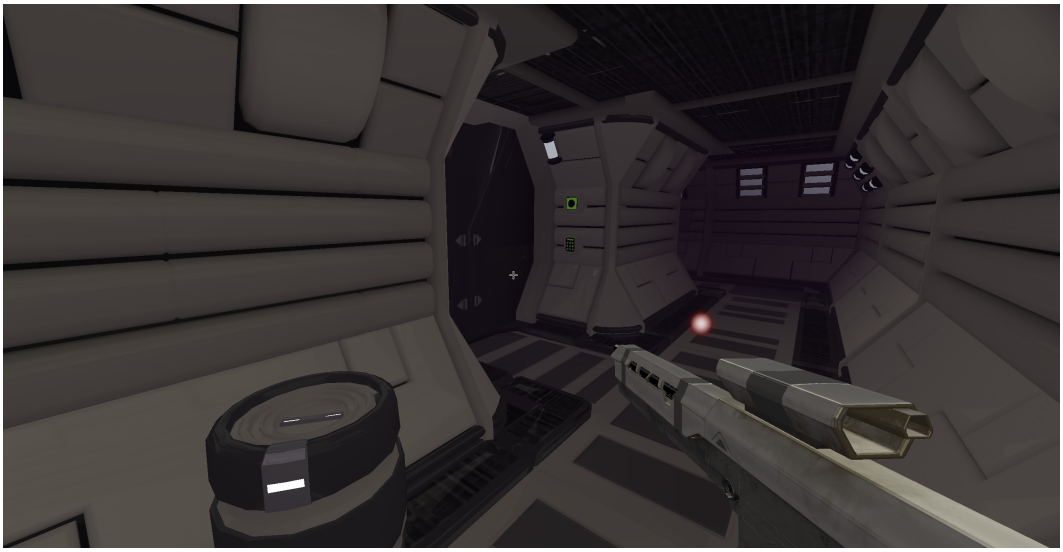

Figure 1: Gameplay screenshot of our Doom-inspired game implemented in Three.js.

1 Introduction

1.1 Tech Choices

Our project leverages Three.js, a lightweight, cross-browser JavaScript library that abstracts WebGL, making 3D graphics programming significantly more accessible. It provides a comprehensive set of features for creating and displaying animated 3D computer graphics in web browsers without requiring users to install additional plugins. Three.js was the best option for its robust ecosystem, extensive documentation, and optimized performance for web-based 3D rendering.

Our development workflow is built around Vite, a modern frontend build tool that provides an extremely fast development server through native ES modules.

We implemented the entire codebase in TypeScript rather than vanilla JavaScript to enhance our development process. TypeScript's static typing helps catch errors during development rather than at runtime, significantly improving code reliability. The explicit type definitions serve as built-in documentation, making it easier for team members to understand each other's code and collaborate effectively.

1.2 Gameplay Overview

Our game is a first-person shooter inspired by the classic Doom, implemented using Three.js. 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 a combat element to the navigation mechanics.

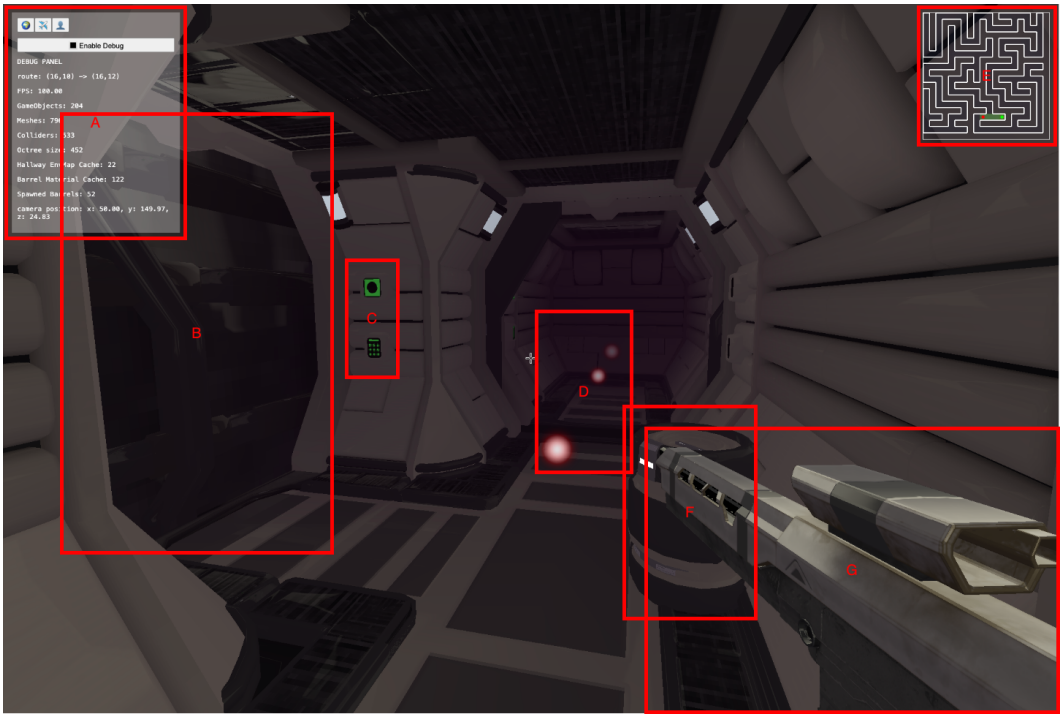

Figure 2: Gameplay screenshot showing key elements: (A) Debug menu for adjusting game parameters, (B) Closed door with its control panel (C), (D) Path markers guiding the player through an open door, (E) Mini-map displaying player position and path to destination, (F) Barrel spawned as a random prop,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

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 and the path to the destination.
- **F:** Barrel that was procedurally spawned as a random prop in the environment.
- **G:** The player's weapon in gun mode.

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 Rendering Pipeline

The rendering system leverages Three.js capabilities while adding custom functionality:

- Scene with visual features such as ambient lighting, exponential fog for atmosphere, and HDR environment mapping for realistic reflections.
- A dual-camera system that allows switching between a first-person player view and a freely moveable camera.
- Debug visualization tools that can render collision boundaries and octree structure when enabled.

2.3 Game Loop

The engine implements a standard game loop that:

1. Updates the physics system and rebuilds the dynamic octree.
2. Calls the `animate` method on all registered game objects.
3. Handles user input and camera updates.
4. 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.

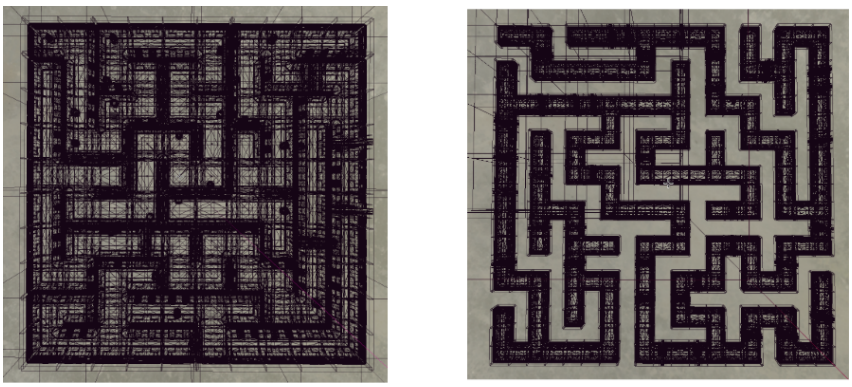

Figure 3: Comparison of a 10x10 maze: left without spacing, right with spacing, viewed from the top in debug mode. Notice the extra spacing making sure that parallel hallways do not touch.

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 serves dual purposes: it generates navigation routes for potential enemy AI and 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.

3.1.5 Random Cell Selection

A utility function is provided to select random cells with specific properties (e.g., cells with at least one open side). This function is specifically used to choose the start and destination points within the maze, ensuring that these critical locations are appropriately positioned in accessible areas of the maze.

3.2 Benchmark

4 3D World Organization

TODO

5 User Controls and Movement

TODO

6 Lighting, Materials and Texture Mapping

TODO

7 Collision Detection Using Octree

TODO

7.1 Benchmarking

7.2 Comparison with other space partitioning algorithms