Computer Graphics (E016712A)
D. BABIN and A. PIZURICA
GHENT UNIVERSITY
2024/2025 AY
Responsible assistant: Jens De Winne

Project: Quake/Doom environment in Three.js

1 Introduction

Three.js is a JavaScript library and API using WebGL that can be used to create and render high-quality color images composed of 3D geometric objects. The goal of this project is to get introduced to the basic functionalities in Three.js and render your own 3D Quake/Doom like environment in a web browser (cfr. figure 1).

2 Assignment

In this project, a Quake/Doom-like environment is created, where a user can move through a maze. The basic features include random maze generation, shortest path finding, drawing geometric primitives, animation, mouse and keyboard handling, usage of light and material properties and texture mapping. In addition, we will investigate how 3D space partitioning can be used to speed up geometric operations involved in moving through the scene and, optionally, shooting on objects.

The project consists of three main tasks. First, a random maze structure has to be generated. Based on this, a 3D world is created in Three.js. In order to move efficiently through the maze, an algorithm has to be implemented that finds the shortest path. Second, space partitioning by means of an octree has to be investigated. More specifically, collision detection should be implemented so the user can move in the 3D scene. Optionally, ray intersection can be implemented to include shooting on objects. Finally, a report has to be written, covering both the thorough but concise explanation *and* the evaluation of the implemented algorithms.

Figure 1: Doom 3 environment.

2.1 Creating the maze and 3D world

The first task is to create a random maze floor plan. Based on this, a 3D maze is generated by using geometric primitives and populating it with corridors. Objects and light sources should be present to distinguish sections of the maze. The choice of objects may be freely chosen but they should have different colors and material properties. Texture has to be applied to the floors, walls and ceilings of the world. To make the world more realistic, bump mapping can be applied. Creativity is encouraged by integrating different objects, e.g. teleports, spawn points, etc. At least one object should have a reflection on the floor. It should also be possible to resize the window. Allow the user to move through the maze using the typical keys: 'a' turn left, 'd' turn right, 'w' move forward, 's' move backward, 'q' move strafe left, 'e' move strafe right. The user should be able to use the mouse in order to look around. Moving mouse forward or backward turns the view up or down, while moving the mouse left or right turns the view to left or right.

Secondly, a shortest path finding algorithm has to be implemented so the user can move through the maze efficiently. It should be possible to interact with a map of the maze and choose a destination. Based on the location of the user inside the maze, a shortest path is determined and dynamically updated. The shortest path should be displayed either on the 2D map or in the 3D world.

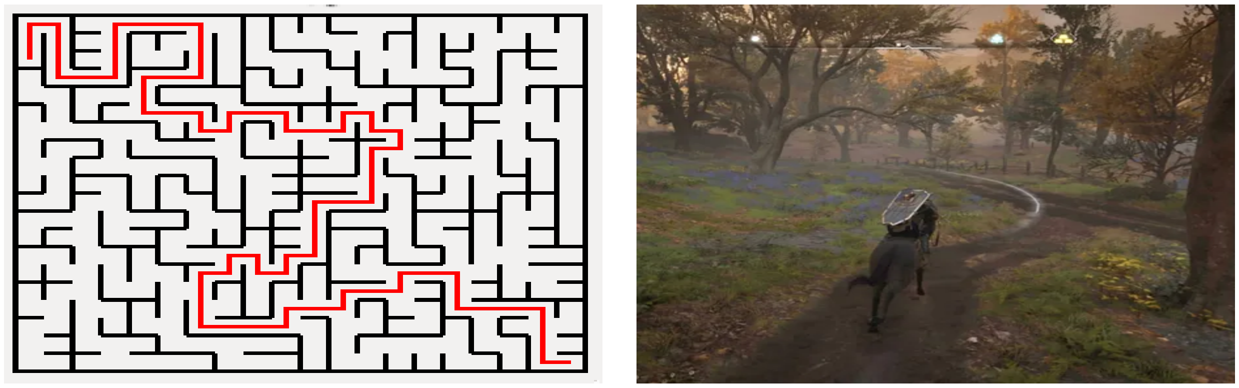
Figure 2: Shortest path visualization can be done either on a 2D map of the maze (left) or directly in the 3D world (right).

2.2 Integrating spatial partitioning

Space partitioning is the process of dividing the (3D) space in non-overlapping regions. It is mostly organized in the form of a hierarchical tree because of its $O(\log n)$ search complexity and therefore it is often used in the field of computer graphics to speed up certain geometry queries. There are several tree-based datastructures, among them the k-d tree, the octree and the binary space partitioning (BSP) tree. Each of them has its own features and limitations and they should be used in those specific use cases where they fit best. However, in this project we will only investigate the octree, an axis aligned tree constructed by cutting the space using regularly spaced planes into cubes. It is the 3D equivalent of a quad-tree. The construction process starts with a bounding box or cube of the entire scene that is recursively subdivided into 8 equal sub-cubes (also called voxels). A visualization of this process is depicted in figure 3. It is not needed to create an implemention of the octree. Instead, an existing implementation (such as in section 3.3) can be used.

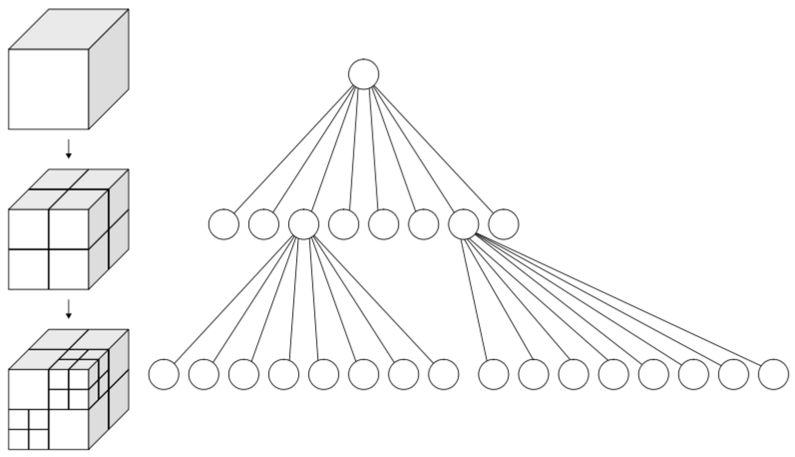

Figure 3: The concept of the octree data structure. The root voxel represents a bounding box of the entire 3D model or 3D scene. Subsequently, at each lower level, the *occupied* voxels are subdivided in eight smaller sub cubes of the same size. The same procedure is repeated until the desired leaf size is obtained.

2.2.1 Collision detection

One of the main usages of space partitioning is collision detection. Collision detection can be used to prohibit the player from running through walls and other objects. Try to implement collision detection using the octree space partitioning.

2.2.2 Ray intersection [Optional]

Another usage of space partitioning is ray tracing. The next task is to allow the player to shoot on objects and walls. To this end, you will have to implement a ray tracing algorithm using the implemented octree. In other words, you will have to compute the octree voxels intersected by a straight line. Try to visualize where the bullet hit the wall or object by leaving a pothole on that wall or object.

2.3 Writing the report

A report has to be written explaining the choices you made for each of the algorithms implemented. An answer on at least each of the following questions should be given.

- Explain how you generated the random maze
- Explain the shortest path finding algorithm
- Explain how you organized the 3D world in terms of geometric primitives.
- Explain moving in 3D. Related to that, describe how frustum culling or hidden surface determination is done.
- What kind of light sources did you use? Explain how they work.
- Which texture mapping strategie(s) did you use? Explain.

- How long does it take to build the octree? Do you have any ideas on further speeding up this process?
- Explain how you implemented collision detection based on the octree.
- Explain how you implemented ray tracing using the octree. What is the time complexity? What is the worst case scenario? [Optional]
- As mentioned in section 2.2 there are other space partitioning algorithms such as the kd-tree and BSP-tree. What are the benefits of using an octree? In which situations would you definitely use it? What are on the other hand the limitations of the octree compared to these other data structures? In which situations you wouldn't use it?

3 General remarks

3.1 Tips and tricks

- Describe the implemented algorithms and their evaluation thoroughly, but keep the report concise. Pseudo code is allowed and in certain situations advisable but keep it as short as possible. Do not list complete classes.
- Understanding the algorithms is important. You are allowed to use existing implementations that you find on the internet as long as you know exactly how they work.
- Do not hesitate to contact your supervisor if something is unclear or if you have further questions.
- Remember that Google is your best friend.

3.2 Software

- Three.js is available on `https://threejs.org/`

3.3 Documentation

- Three.js octree and collision detection: `https://threejs.org/examples/games_fps.html`
- Three.js tutorial: `https://discoverthreejs.com/book/`
- Three.js examples: `https://threejs.org/examples/`
- Three.js manual: `https://threejs.org/manual/`