Technical Specification - Dungeons and Crawlers

Project Team:
Caolan William Cochrane - 22490802
Ethan Doyle - 22497082

TABLE OF CONTENTS

1. Introduction

1.1 Overview

Dungeons and Crawlers is a multiplayer, online web-based game. It offers players a procedurally generated voxel world in which they can explore, play with friends, and more. The game incorporates AI-driven storytelling and dynamic NPC interactions. The AI is also implemented in the game's terrain generation - alongside perlin noise generation, the AI can also make biomes of its own, feeding different parameters, biome names and block types + layers into the perlin noise, editing it on its own whim. The NPCs will have the ability to offer quests to the player. The world contains accurate physics, featuring precise collision detection and recorrection through a manual physics engine as well as quality textures that are applied to voxels based on a block ID granted by a mix of the AI and the perlin noise. These textures are applied to different faces of a block through the calculation of the block's normal. The world also features realistic movement as well as a rotatable third-person camera.

For our initial implementation, our focus was on building a strong foundation of AI-driven dialogue and procedural systems that we can expand over time, which we have been able to accomplish.

The game is built with JavaScript and uses Three.js for rendering, graphics and textures, as well as movement and physics. WebSockets is used for multiplayer, and DeepSeek is the AI model used for all-purposes AI generation. We used Express as our web framework and Vite for frontend development. The multiplayer servers are based on a listen server architecture and features client-side prediction.

1.2 Glossary

1. **Replication:**
   Ensures all players see the same world state in multiplayer.
2. **Perlin Noise:**
   Algorithm used for procedural world generation.
3. **Synthetic Database:**
   Synthetic data is artificial data that retains the characteristics of real-world data.
4. **Differential Updates:**
   An update that only has the differences between the previous build version and the new build version.
5. **Context Window:**
   The amount of text (in tokens) that a large language model can remember and use at any time.
6. **DeepSeek**
   An open source LLM model released by DeepSeek, said to rival models like OpenAi's o1 model.

7. **Listen-Server Architecture:**
   A listen server architecture is composed of a user running a game on their machine with this user acting as the server.
8. **Procedural Generation:**
   A method of creating data utilising an algorithm instead of creating data manually.
9. **Entity-Component System:**
   A software architecture pattern used as a representation of game world objects.
10. **NPC:**
    Non-Player Character
11. **LLM:**
    Large language model
12. **ThreeJS:**
    A JavaScript cross-browser library and API used to animate and display 3D computer graphics.
13. **Voxel:**
    A 3D pixel, used to represent a block at a specific XYZ coordinate.
14. **Broadphase:**
    A phase of collision detection which uses an offset to detect possible collisions near the player.
15. **Narrow Phase:**
    The phase of collision after the broadphase where actual collisions are found and then precisely resolves said collisions through calculating the collision point, overlap and collision normal.
16. **Vite:**
    A build tool used for frontend development, enabling fast reloading.
17. **Texture Atlas:**
    An image made up of several different textures.
18. **UV Coordinates:**
    Coordinates used for mapping a texture from a texture atlas.

2. System Architecture

2.1 Reused Components

- **Three.js:**
  - Three.js is a Javascript library. It is used to render 3D graphics in the browser which enables smooth visualization of our worlds.
- **Perlin Noise / Simplex Noise:**
  - Perlin noise is a popular procedural generation algorithm invented by Ken Perlin. It can be used to generate things like textures and terrain procedurally, meaning without them being manually made by an artist or designer. [1]
  - Essentially, perlin and simplex noise are functions that produce noise (random-like values) but the values change smoothly and consistently. Simplex is designed to be more efficient by using polygons and works perfectly well in the 3rd dimension (which is our max scope)
- **Node.js:**
  - Node.js is a cross-platform Javascript runtime environment that is open-source. It allows for server-side execution of JavaScript code.
- **WebSockets:**
  - WebSockets provides a persistent full-duplex communication channel between the client and server - essentially, a two-way communication session between a user's client and a server. Helps to enable the listen-server architecture.
- **Vite:**
  - Vite is a frontend development tool. It helps to reduce wait times for development by introducing hot reloading and usage of ES modules.
- **Express.js:**
  - Express.js is a lightweight Node.js web application framework. It is used to build RESTful APIs.
- **DeepSeek:**
  - DeepSeek is the large language AI model we are using for this project. We tested it against other AI models like LLama and found it to be the best suited for what we want out of the project.

2.2 Architectural Overview

Our system architecture is modular and contains many key components, mainly consisting of both frontend and backend. It is divided into the following modules.

2.2.1 Frontend - Game Client, 3D Rendering

- Runs in the browser.
- We use Vite for frontend development and Express.js as our Node.js web application framework.

- Our frontend development is stored in public/js and we have static texture files stored in public/textures.
- The game client & frontend largely handle 3D rendering, user input and real-time multiplayer connection using WebSockets.
- Our 3D rendering and texturing is largely handled using Three.js, which is our JavaScript 3D library.
- The frontend communicates with the backend to render & generate terrain.
- Physics is also handled through the frontend. Player positioning is sent to the backend for validation and client-side prediction.
- The user connects to their game client with a username and opens up a WebSockets communication. When they connect to their client, the 3D rendering, the terrain generation and position and physics updates begin running.

2.2.2 Backend - Game Server, Perlin Noise Generation

- The backend is in charge of setting up the game server for multiplayer synchronization. The game server handles the positioning of other players, and gives player-specific & chunk-specific information to the frontend so it can render and update these changes accordingly on the game's client.
- The server follows a listen server architecture format and features client-side prediction.
- It uses WebSockets for server-client information.
- Procedural generation is also done by the backend and is sent to the frontend so it can render chunks. This is done through perlin noise. In order to render chunks, we generate an array of valid block IDs through noise alongside a valid chunk location. This information is then sent to the frontend which iterates through the array.
- The noise can have different parameters set, like amplitude, which generates higher terrain.
- Chunks are a 3d array, 16 * 16 * 16 in size.

2.2.3 LLM (Large Language Model)

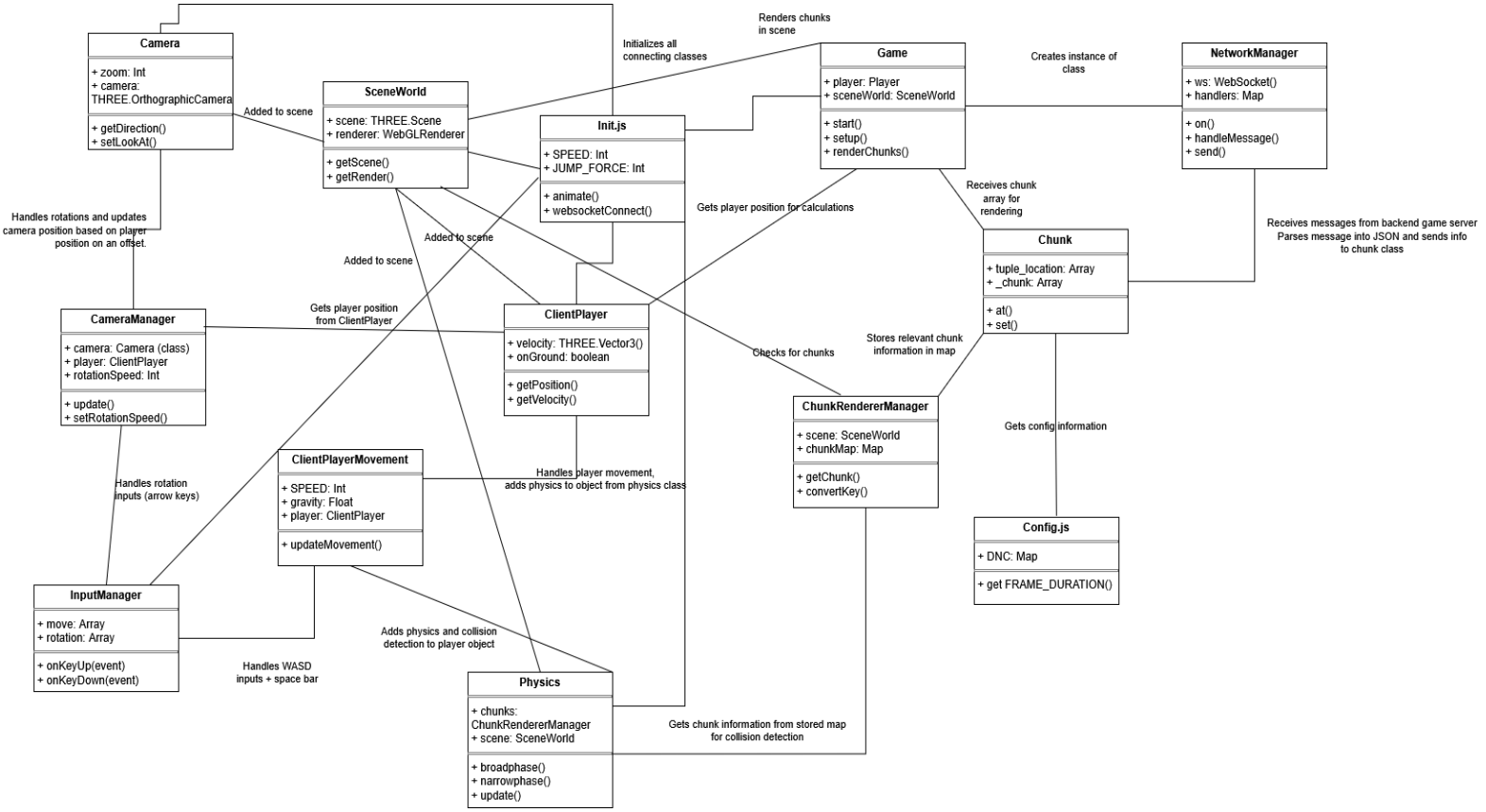
- Our large language model, DeepSeek, powers and generates dynamic NPC dialogue, quests and story content.
- It communicates with the game server to receive world context.
- In addition to dialogue, it also fuels our biome system, sending javascript objects to the perlin noise generator for it to generate, with parameters, names, layers, textures to use, etc.

2.2.4 Database

- Stores plater data, world states and NPC interactions
- Interfaces with the server and procedural world generation system

3. High-Level Design

3.1 Object Model



This object model describes the frontend classes/components of our system and how terrain generation is rendered and received from the backend. We decided to showcase this instead of the backend as this is what will mostly be seen during the project demo.

3.2 Component Breakdown

Init.js:

- Main file. Initializes most of the other classes. Contains the animate() function which begins the 3D rendering. Also contains the game class, which starts the game loop which is connected to the backend, and also handles the chunk rendering.

Config.js:

- Stores some important information like chunk size and render distance.

SceneWorld:

- Sets up the Three.js scene, renderer and lights for the game. Passed to many classes.

Game:

- Game loop class. Is connected to the backend through its network manager, which

sends messages to the game server. The network manager in the game class will create a chunk instance upon receiving a chunk message from the game server. This chunk instance is used to render terrain in the renderChunks() function, which uses instancedMeshes and a shaderMaterial to apply textures onto the blocks using our texture atlas in the public/textures folder. Sends position updates to the game server for players.

Camera:

- An isometric camera that is set at an offset at (20, 20, 20) that follows the player. We can get the direction the camera is facing to use so we know which direction to move the player object in.

CameraManager:

- Handles movement from inputManager for camera - rotates camera when arrow keys are pressed. Also updates the camera's position to the player's each frame.

ClientPlayer:

- The client-side player class. Creates a player object through Three.js and stores its position, velocity and grounded state.

ClientPlayerMovement:

- Similarly to CameraManager, handles movement for the client-side player class through updating the velocity vector using info from inputManager. It also applies gravity to the player object. Updated every frame.

InputManager:

- Contains event listeners and two maps that store movement information, one for WASD + space bar and another for arrow keys. Passes these maps to the ClientPlayerMovement and CameraManager classes respectively.

Physics:

- Manual physics engine for collision detection + recorrection. Uses information from the chunkRendererManager. 3 phases: broadphase, narrow phase and resolving collisions. Broadphase checks for potential collisions using a slight offset from the player object's position. Narrow phase calculates actual collisions using the contact point, overlap and collision normal. Using the information from the narrow phase, collisions are resolved by multiplying the collision overlap by the collision normal and adding it to the player position. Also negates the player's velocity along the axis that it collided with. Updates every frame.

Chunk:

- Stores two arrays: one, the local chunk coordinates, two, a 16*16*16 array of block IDs detailing where blocks are in a chunk.

ChunkRendererManager:

- When an instancedMesh is created for a chunk, it stores the chunk's location as a key, then the chunk itself and its mesh as values in a map.

NetworkManager:

- Handles and parses information from and to the backend, namely the Server.js class. For the game instance, awaits a chunk creation from the backend and sends it to the game class so it knows to create a chunk instance and render it.

4. Problems and Resolution

4.1 Physics Overhead

Originally, we had used Cannon-es.js alongside Three.js as our physics engine. It handled collisions, player movement, gravity and so on automatically through its engine. Cannon-es.js works by adding a “body” to a Three.js render. This body has forces acted upon itself by other forces, either collisions from other objects or forces that are implemented into the “world”, like gravity. However, having to add physics bodies to entire chunks added a huge performance overhead into our game - chunks were rendered extremely slowly and the player’s movement would come to a halt.

To solve this problem, we opted to use instanced meshes for our Three.js renders for chunks. Essentially, an instanced mesh reduces the number of draw calls as it renders a large number of objects that use the same geometry and materials - perfect for voxel chunks. However, we still found there to be a performance issue with Cannon-es. We eventually decided to drop the engine from our game and implement our own manual physics engine that would offer its own movement system and calculate its own broadphase, narrow phase and contact resolution.

4.2 LLM Performance

Originally, we opted to use the AI model LLama3.2-1B, an open source LLM model released by Meta. However, we found its performance lacking, especially for what we wanted out of the AI model. We eventually decided to switch to DeepSeek down the line.

4.3 Texture Mapping

Since we had decided to use instanced meshes to reduce performance overhead, we struggled to find out how to map textures onto the objects, since instanced meshes only use one material. We found that we could make this material a ShaderMaterial, which accepted UV coordinates that mapped onto a texture atlas and gave blocks the texture(s) that were mapped out from the UV coordinates.

We also had to find out how to get the textures mapped onto different faces of the blocks - we found that we could do this through the block’s normal from its vertices. Additionally, there was an error with textures being set upside down. Most frameworks and programs, when mapping to a texture atlas, go from 0 being the top to 1 being the bottom, but in Three.js, it is flipped. 1 refers to the top whereas 0 is the bottom, so the index (0,1) means top left instead of bottom left. After finding this out, we corrected our texture mapping and everything worked perfectly.

5. Installation Guide

5.1 System Requirements

Software Requirements:

- Node.js
- PostgreSQL
- Three.js
- WebSocket server
- Local LLM (if wanted)

5.2 Installation Steps

First, clone the repository from the GitLab:

git clone git@gitlab.computing.dcu.ie:cochrac2/2025-csc1049-ccochrane-dnc.git (ssh)
git clone https://gitlab.computing.dcu.ie/cochrac2/2025-csc1049-ccochrane-dnc.git (https)

(SSH recommended).

We're using Vite as our frontend development tool for fast builds and live reloading. It's currently set up for local development but can be swapped out if needed.

If you're new to Node.js, take a look at the package.json file. There's a "scripts" section that contains:

```
"scripts": {
  "front-dev": "vite"
}
```

This allows you to quickly start Vite by running:

npm run front-dev

Don't forget to install all dependencies by running:

npm install

This will install everything listed in package.json that Vite and Express need.
You can start the backend using npm run start. It will output the port it's listening on, go to localhost:your_port as a url in a web browser to visit the page.

The main entry point for the application is src/main.js. This is where we set up the Express server and can build out API routes as needed.

3.2 Directory Overview

public/ – Holds all static files served to the frontend, like HTML, images, and stylesheets.
public/js/ – Contains frontend JavaScript files for the UI.
src/ – Holds the main application code and API routes for the backend.