Design Document

CS 307 Group 16 | Kyle Day, Jason Seeley, Stefan Gerber, Yuchuan Huang

1. Purpose

The Fathomless Caverns of Peril game strives to create a one of a kind gaming experience without any compromises in performance or user security. There are a plethora of roguelike, procedurally generated dungeon crawl games currently in existence. Our project will combine several uncommon features seldom found within this general genre of game. The main goal of our development is to fix undesired mechanics encountered in other roguelike games while supplying a competitive, score based leaderboard. These newly implemented gameplay features will yield a series of fun, engaging, and challenging single player levels. In our single player game, the player controls a character with 4 way movement across progressively more complex level maps (starting at a simple ground level, going deeper into a cave dungeon, and ending at the elaborate "Underworld"). The player picks up items along the way that help them defeat monsters with the ultimate task of traversing/beating all the maze-like cave levels and killing a final boss.

Additionally the system will satisfy our rigorous requirements for security and latency. We must maintain a seamless transition between each player movement and calculated entity movements along with the generation of new terrain. This is absolutely required in order to contribute to the immersive nature of our unique and interesting world. User data must not be stored unless absolutely critical to the game functionality, such as in the case with usernames and password hashes.

2. Design Outline

(a) Design Decisions:

Our system uses a client-server for its online functionality. The game will run on a lean server to allow for a lowered latency between each user input. All renderings and player movements will be computed on the client while the server simply validates each movement.This will subsequently allow the server the freedom to manage leaderboards, level generation, entity pathfinding, updating of files, and other record keeping tasks.

Client-Server Model Justification:

The server offloads the computationally heavy tasks of procedural generation and game state management, leaving the client to focus on the game's graphical performance.

Client: Responsible for displaying textures, capturing player input (movement, actions, etc.), and displaying game information such as the HUD (health, items, score).

Server: Hosts game logic, handles procedural generation, manages persistent data like player profiles and high scores, and validates player actions. This setup ensures that player data, especially on competitive leaderboards, is secure and not vulnerable to tampering.

Database: Stores player data, high scores, game configurations, and persistent settings such as custom textures or mods.

(b) System Components:

Client: Handles all front-end tasks, such as displaying the game world (dungeon, enemies, player), managing the UI (inventory, stats), and capturing player input.

- Displaying game visuals based on server input (procedural levels, enemies).
- Capturing player actions (movement, combat, item interactions) and sending them to the server.
- Displaying feedback to the player, such as health, items, enemy status, and in-game events.

Server: Acts as the "game master" that controls the game logic and ensures fairness, particularly important for leaderboard tracking and procedural generation.

- Generates each level randomly (within pre-set parameters) while ensuring the design is solvable and fits within the gameplay's difficulty curve.
- Validates player actions (such as attacks, item pickups) and updates the world state accordingly (e.g., enemy deaths, score updates).
- Tracks and updates scores based on in-game achievements, and posts these to a competitive leaderboard.
- Manages save states and game preferences (like toggling sounds and texture qualities).

Database: Persistent storage for all game-related data.

- Saves player information, including stats, saved games, and preferences.
- Stores competitive leaderboard data and player rankings.

(c) Interactions Between System Components:

Client-Server Interaction: When the player moves or attacks, the client captures the input and sends it to the server through an action listener. The server validates the action (e.g., checks if the

attack hits, whether the player is within attack range) and updates the game state. Then the server sends back the updated game state data (e.g., enemy health, player score), which the client uses to render changes on the screen.

For example:

1) The player presses a key to move to the next room.
2) The client sends this movement request to the server.
3) The server checks if the room exists and is accessible.
4) The server responds back with the portion of the map within the player's field of view.
5) The client displays all game objects within the map.

Client-Server-Database Interaction:

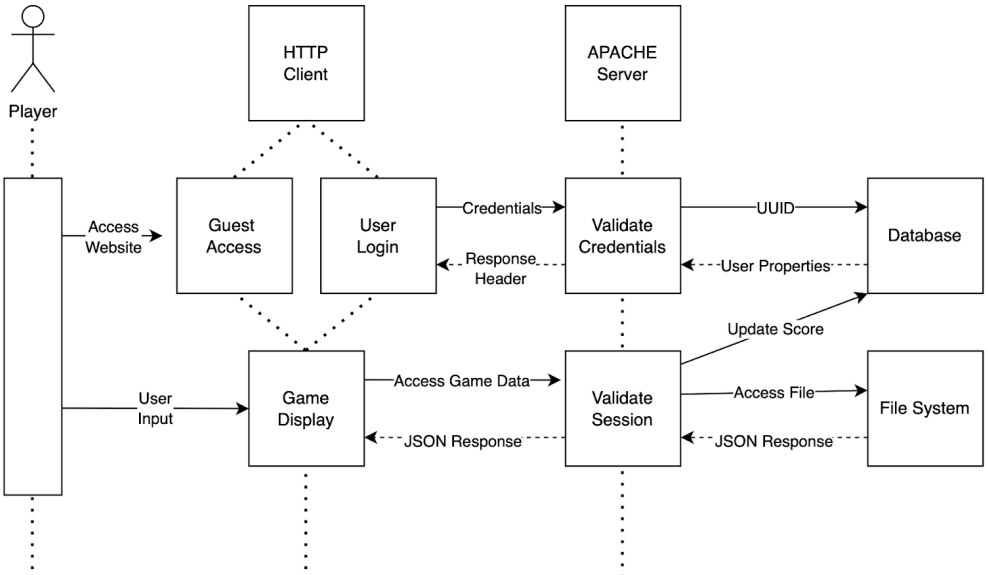
The player can log in to play their saved game state. The user's information is validated on the server against the stored information saved into the database. On a successful login the user's properties are returned and loaded into the client.

Example:

- When a player logs in, the client requests their saved user preferences and statistics from the database. (Volume level, nickname, texture quality, highscore, games played)

Server-Database Interaction: The server retrieves player scores and updates the leaderboard in real-time based on player performance.

(c) High-level structure of the system.



3. Design Issues

Functional Issues:

Entity Pathfinding

How should enemies behave during the game?

Options:

1. Basic AI: Enemies follow a set movement pattern with simple attack behavior.
2. Advanced AI: Enemies have varying degrees of intelligence, such as pathfinding and reactionary behavior based on the player's actions.
3. Dynamic AI: Enemy behavior changes dynamically based on environmental factors or player stats.
4. Random Walks with Weighted Bias: Implement a random walk algorithm but introduce a bias towards specific directions or areas. For example, mobs might be biased to move toward the center of rooms or avoid walls.

Chosen Solution: Advanced AI

Justification: Advanced AI adds strategic depth without overly complicating the system. This includes features like pathfinding and basic tactical behaviors (e.g., mobs flanking the player), making combat more engaging and rewarding

Level generation:

How to make maps randomly generated while maintaining rationality and not violating game conditions

Options:

1. Biome-based Generation: Divide the game world into biomes or regions that each follow different generation rules. Each biome has its own set of rules for what is logically placed, ensuring that it fits the background of the game.
2. Template-based Generation: Predefined several templates of rooms, corridors, and specific terrain features. During map generation, randomly combine these to create the final layout.
3. Completely random generation:
4. Pure Random Generation: Each level is completely randomly generated.
5. Seeded Random Generation: Use a seed for random generation, ensuring the same procedural level is created for a specific playthrough.

Chosen Solution: Biome-based Generation:

Justification: This approach ensures that each area feels unique and logical within its own context, preventing random elements from appearing in places where they don't belong. For example, a cave should have a larger and organically shaped room and have some plant life still growing. while the magma core should contain plenty of pools of lava and hot rocks surrounding them.

Game balance:

How to ensure that the various values in the game are reasonable and balanced

Options:

1. Base level and difficulty: start with a well-balanced base level, where all other content can be measured and scaled.
2. Procedural Challenge Tuning: Use procedural generation not just for map layout, but also to dynamically adjust challenges based on the game progress.

Chosen Solution: Base level and difficulty

Justification: With a well-balanced base level, we can scale difficulty more effectively. For example, enemies in higher levels could have their stats increase proportionally based on the baseline. Adding new content, such as new monsters, weapons, and abilities, can be systematically scaled relative to the base difficulty.

Health Management System:

How should the player's health be managed during the game?

Options:

1. Health Regenerates Over Time: The player regenerates health slowly over time.
2. No Health Regeneration: The player must rely on items like potions to heal, with no automatic regeneration.
3. Regeneration Based on Actions: Health regenerates when the player completes specific actions, like defeating enemies.

Chosen Solution: No Health Regeneration

Justification: Removing health regeneration adds a level of challenge and forces players to manage resources carefully, aligning with the roguelike genre's focus on strategy and survival. It also encourages players to explore and use consumable items rather than passively waiting to recover health.

Optimized Client Server Interaction:

How can the server and the client effectively and efficiently share information?

Options:

1. Server servers all display instructions. Client only renders what is given to it. The client would perform no other calculations other than what is required to display the game, take user inputs, and communicate with the server.
2. Client performs all tasks related to the game. The only purpose of the server is to maintain the leaderboard and other online functionality.
3. The client and server share the workload.

Chosen Solution:

We have chosen a mixture between options 2 and 3.

Justification:

The online game will use the third option of distributing the workload. This allows for the fundamental calculations to be performed on the user's device and immediately be displayed without having to wait for a server response. The server will only be required for performing computationally intensive tasks like level generation and path finding. Additionally the server will be responsible for validating each of the player's movements and of course maintaining access to the leaderboard. By creating every game element as a tile, the client only has to initially load in each resource for the map and duplicate the asset as instructed to by the server. This reduces the payload that must be sent by the server on each move update into simply a matrix of texture indices and rotations. This is compounded with the fact that the player has a limited field of view so the server only needs to send a subset of the map that the player can see. This was not only chosen for efficiency, but also for functionality as well. A player will be unable to make illegal movements by hacking the client as all actions are validated on the server.

The reason we have chosen a blend of two methods is due to the offline game that will also be supported. This version of the game will clearly be more reminiscent of option 2 with a client only approach, thus completely removing all online functionality. This was chosen as it is the only way for players to be able to safely modify the game on their device. This approach also manages to ensure the safety of the online users and maintains the integrity of the competitive leaderboard.

Non-Functional Issues:

Scalability of Procedural Generation:

How can the procedural generation system be optimized to handle increasingly complex level designs without affecting performance?

Options:

1. Dynamic Loading of Procedural Levels: Load parts of the level dynamically as the player moves through it, reducing memory usage.

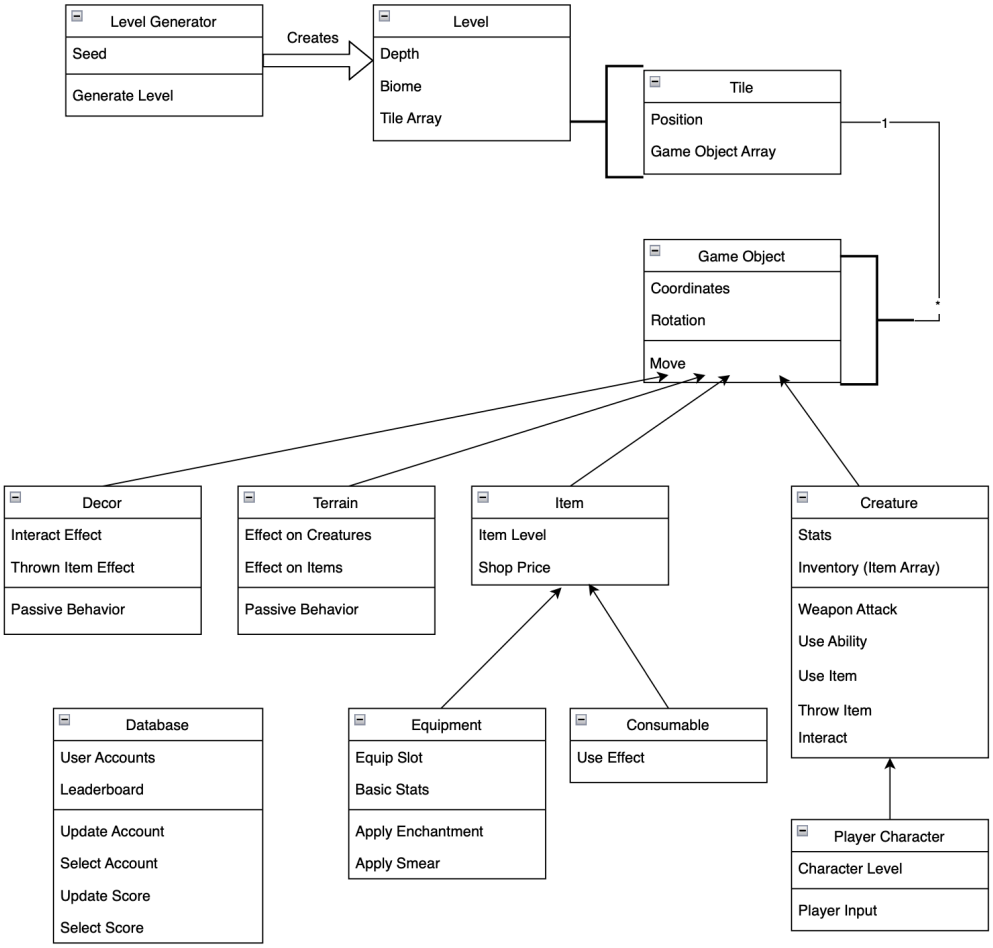
2. Pre-Generated Levels with Random Elements: Generate levels in advance with some randomness, rather than generating everything on the fly.
3. Algorithm Optimization: Optimize the procedural generation algorithm to reduce computational load (e.g., by simplifying pathfinding or decoration placement).

Chosen Solution: Dynamic Loading with Algorithm Optimization

Justification: Dynamic loading allows for larger and more complex levels without overwhelming system resources, while algorithm optimization ensures that the generation process remains fast and efficient. This ensures that the game can scale with increasing level complexity as players progress

4. Design Details

Class Level Design



(b) Classes and interactions

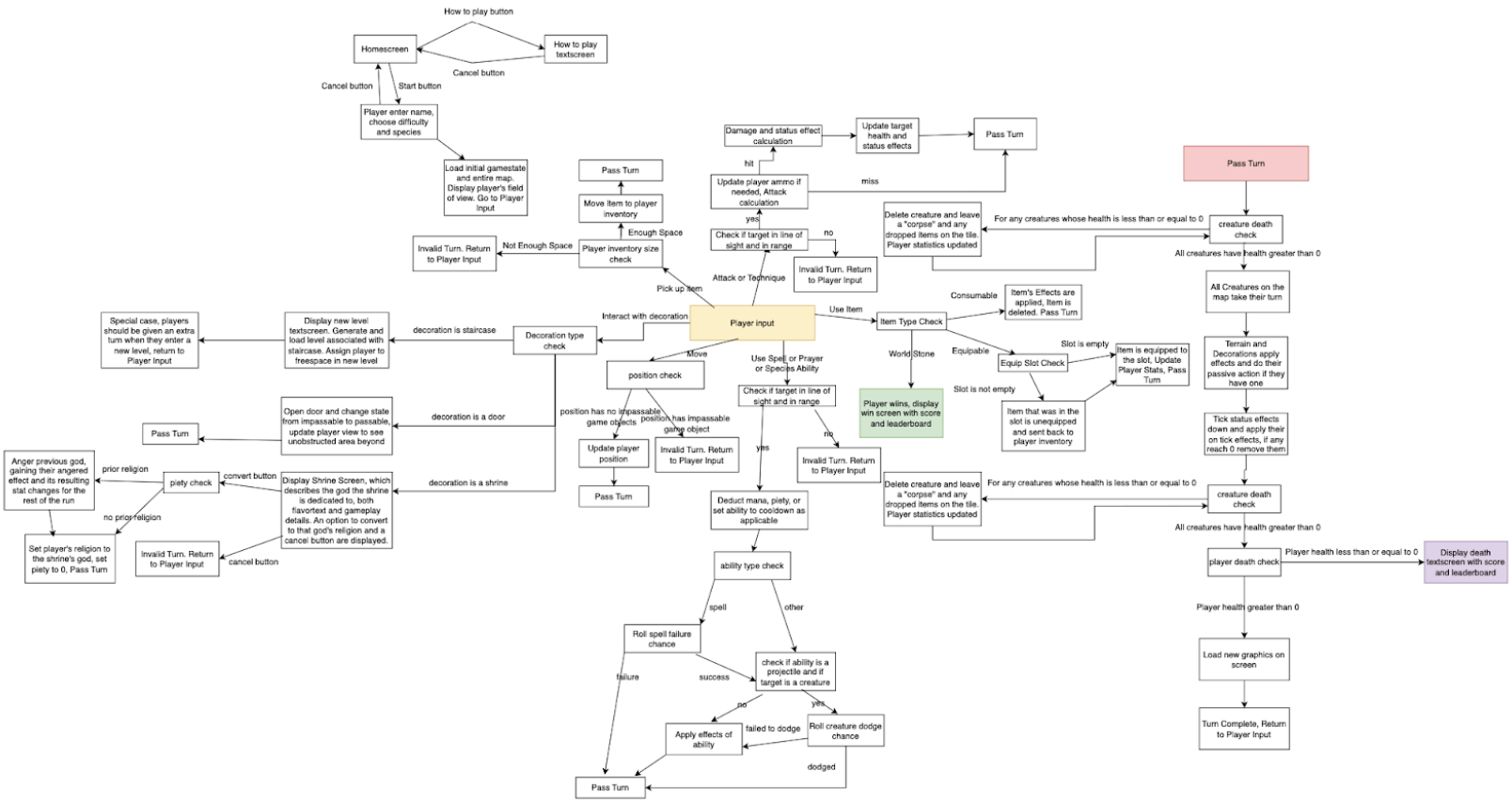
An object oriented approach to level generation allows for new items, tiles, entities and behaviors to be added just by creating a new object.

There is a level generation class that creates a level or map based on unique seed criteria. The level class holds all of the elements for the current map. It is defined as a tile matrix where each tile's index acts as its coordinate within the level. A tile is an object that is able to hold many different game objects (decor, terrain, item, or creature). Each of these specific game object types inherit from the main game object which has a unique coordinate identifier and rotation. There is a unique creation type called the player. There are also unique types of items which are either consumable or equipment items.

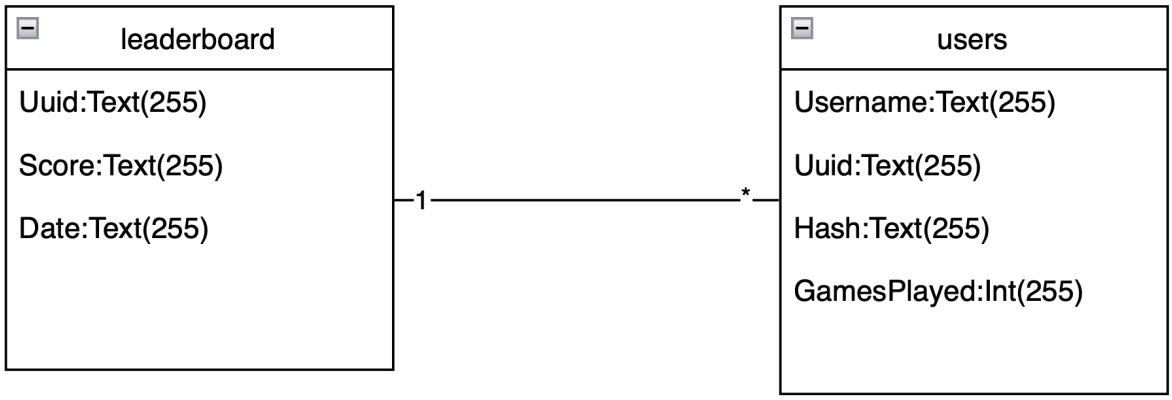
Finally there will be a database class that is responsible for directly accessing the database. Its methods will allow for the manipulation of the data tables.

(c) Diagrams

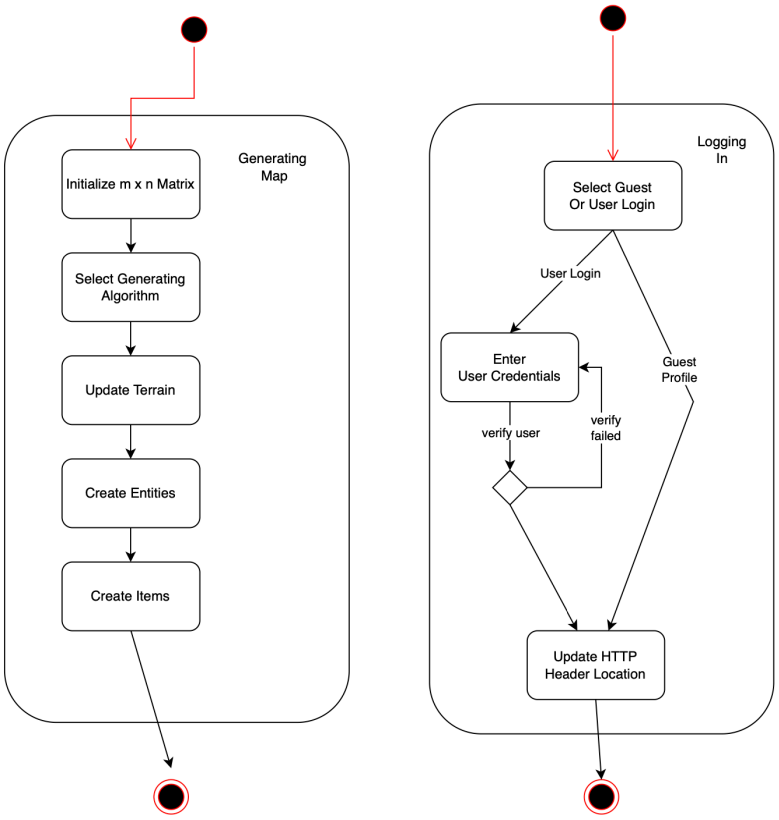
Sequence diagram



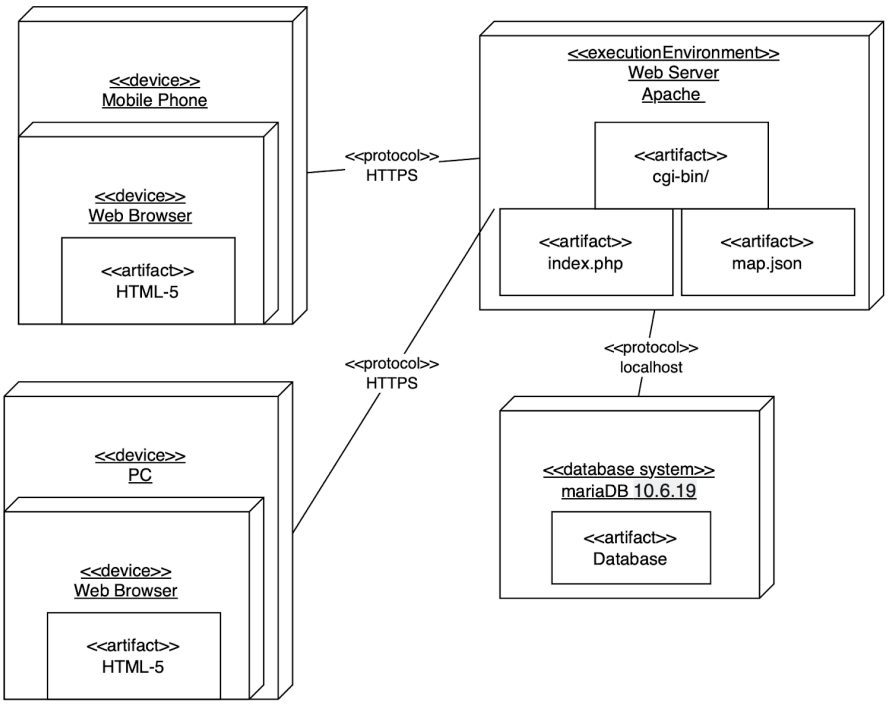
ERD Diagram



State Machine Diagrams

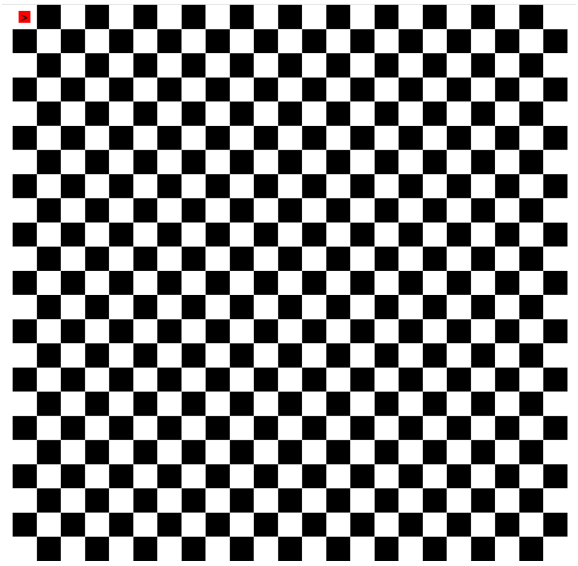


Deployment Diagram

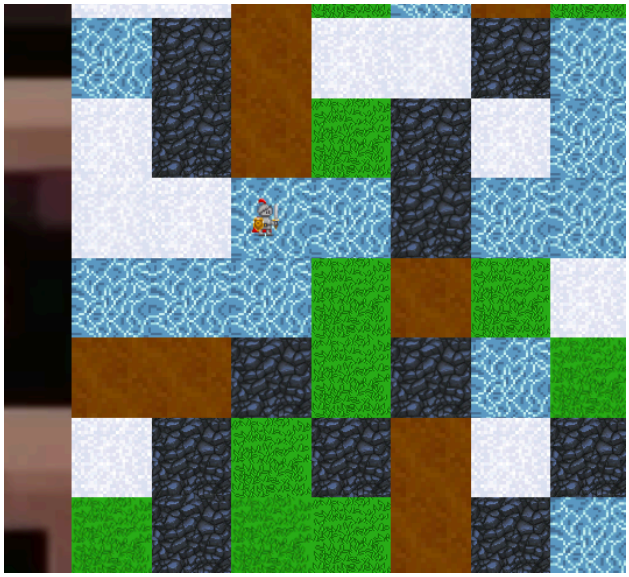


UI Mockup

A character should be able traverse a grid with orientation

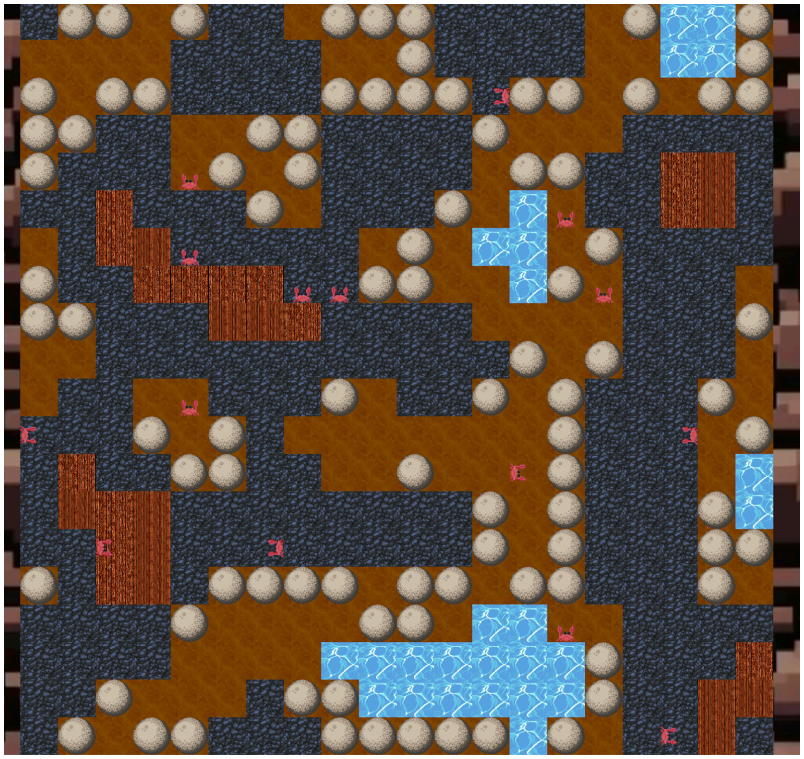


The grid should allow for different textures

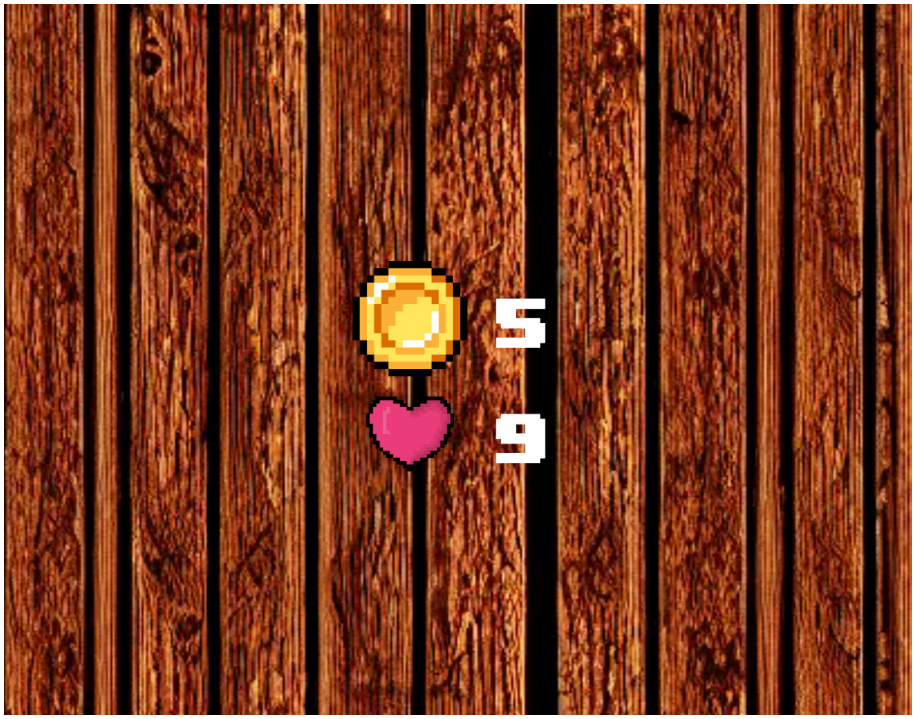


The textures should be arranged in procedurally generated patterns to produce a fully explorable world with items, obstacles, and entities.

Example of procedurally generated map



An inventory with current items and health.



Helpful tips may appear to guide the player