Design Document: Randomized Thematic Levels

1. Feature Overview

This feature introduces a system where each new dungeon level (Z-index) in the `MAZE` biome is assigned a random "theme" upon its first generation. This theme dictates the level's visual aesthetic (wall, floor, and fog colors) and influences the type of monsters and loot that spawn (e.g., a "RED" theme spawns fire-based monsters and fire-modified items).

2. Core Concepts

1. **Theme Persistence:** The theme for a level (e.g., Level 2) must be random, but it must be *persistent*. If the player leaves Level 2 and returns, it must have the same theme. This will be managed by `WorldManager`.
2. **Data-Driven Generation:** The chosen `RetroTheme.Theme` enum will be passed as a parameter through the chunk generation pipeline (`WorldManager` -> `IChunkGenerator` -> `SpawnManager`/`LootTable`).
3. **Aesthetic vs. Gameplay:** The theme has two effects:
    - **Aesthetic:** The `Maze` object will store its theme, which the `FirstPersonRenderer` will use to pull colors from `RetroTheme`.
    - **Gameplay:** A new mapping class (e.g., `ThemeDataManager`) will translate a `RetroTheme.Theme` into gameplay rules (e.g., `Theme.BLUE` -> `ModifierType.ICE`), which `SpawnManager` and `LootTable` will use.
4. **Trigger:** The theme is assigned when a new Z-level is generated, which is triggered by the player using a "Ladder" object in the `MAZE` biome.

3. Proposed Class Modifications

Here is the high-level plan for the classes we will need to modify.

- **`WorldManager.java` (I have this file):**
    - **New Field:** Add `private Map<Integer, RetroTheme.Theme> levelThemes = new HashMap<>();` This map will store the assigned theme for each level number, ensuring persistence.
    - **New Field:** Add `private RetroTheme.Theme currentLevelTheme = RetroTheme.Theme.DEFAULT_GREEN;` to hold the theme for the *current* level.
    - **New Method:** `public RetroTheme.Theme getThemeForLevel(int level)`: Checks the `levelThemes` map. If a theme exists, it returns it. If not, it generates a *new* random theme, stores it in the map, and returns it.

    - **Modify `setCurrentLevel(int level)`:** This method will now also call `this.currentLevelTheme = getThemeForLevel(level);` to update the currently active theme.
    - **Modify `loadChunk(GridPoint2 chunkId)`:** This method will be modified to pass the `this.currentLevelTheme` to the generator. The line: `Maze newMaze = generator.generateChunk(chunkId, currentLevel, difficulty, gameMode);` ...will become: `Maze newMaze = generator.generateChunk(chunkId, currentLevel, difficulty, gameMode, this.currentLevelTheme);`
    - **Modify `saveChunk` / `loadChunk`:** The `levelThemes` map needs to be serialized and saved as part of the world save data (e.g., in a `world_meta.json`). For now, we'll keep it in memory.
- **`IChunkGenerator.java` (File needed):**
    - **Modify Interface:** The `generateChunk` signature must be updated to include the theme: `Maze generateChunk(GridPoint2 chunkId, int level, Difficulty difficulty, GameMode gameMode, RetroTheme.Theme theme);`
- **`MazeChunkGenerator.java` & `ForestChunkGenerator.java` (Files needed):**
    - **Implement Interface:** Update the `generateChunk` method to match the new signature.
    - **`MazeChunkGenerator`:**
        1. When the `Maze` object is created, it must call `newMaze.setTheme(theme);`.
        2. When spawning monsters or loot, it must pass the `theme` to `SpawnManager` and `LootTable` (e.g., `spawnManager.spawnMonster(..., theme)`).
- **`Maze.java` (File needed):**
    - **New Field:** `private RetroTheme.Theme theme;`
    - **New Methods:** `public void setTheme(RetroTheme.Theme theme)` and `public RetroTheme.Theme getTheme()`.
- **`RetroTheme.java` (File needed):**
    - **New Method:** `public static RetroTheme.Theme getRandomTheme()`: A static method that returns a random theme from the available enums (excluding `DEFAULT_GREEN` perhaps). We'll need `java.util.Random`.
- **`FirstPersonRenderer.java` (or similar rendering class; file needed):**
    - **Modify `render()`:** Before rendering walls/floors, it must get the theme from the current maze: `RetroTheme.Theme theme = player.getMaze().getTheme(); Color wallColor =`

`theme.getWallColor(); // (Assuming method exists) Color floorColor = theme.getFloorColor();`
    - It will use these colors instead of hard-coded ones.
- **`SpawnManager.java` & `LootTable.java` (Files needed):**
    - **Modify Methods:** All public methods for spawning entities or creating loot will be updated to accept a `RetroTheme.Theme theme` parameter.
    - These classes will use a `switch` statement or a new `ThemeDataManager` to apply theme-specific modifiers (e.g., if `theme == Theme.RED`, add `ModifierType.FIRE`).
- **`GameScreen.java` (File needed):**
    - **Modify `update()` / Input Handling:** We will find the "use" action logic.
    - **New Logic:** When the player "uses" a ladder (which we'll define as a `Scenery` object), `GameScreen` will orchestrate the level descent. This involves:
        1. `int newLevel = worldManager.getCurrentLevel() + 1;`
        2. `worldManager.setCurrentLevel(newLevel);` (This will set the new theme)
        3. `worldManager.clearLoadedChunks();`
        4. `GridPoint2 startPos = worldManager.getInitialPlayerStartPos();`
        5. `Maze newMaze = worldManager.loadChunk(startPos);`
        6. `worldManager.transitionPlayerToChunk(player, startPos, newMaze.getPlayerStart());`
        7. `player.setMaze(newMaze);` (or this might be handled by `transitionPlayerToChunk`)

Project Plan & Implementation

This plan is broken into testable milestones. After each milestone, we will compile and run the game to ensure stability.

Request for Files

To begin, I will need to see the latest versions of the following files from your repository. Please provide their full content:

1. **`core/src/main/java/com/bpm/minotaur/rendering/RetroTheme.java`** (To add theme randomization)
2. **`core/src/main/java/com/bpm/minotaur/gamedata/Maze.java`** (To store the theme)

3. **`core/src/main/java/com/bpm/minotaur/generation/IChunkGenerator.java`** (To update the interface)
4. **`core/src/main/java/com/bpm/minotaur/generation/MazeChunkGenerator.java`** (To implement the new interface)
5. **`core/src/main/java/com/bpm/minotaur/generation/ForestChunkGenerator.java`** (To ensure it still compiles)
6. **`core/src/main/java/com/bpm/minotaur/screens/GameScreen.java`** (To find the "use" action and implement the trigger)

I will stop here. Once you provide these files, I will generate the explicit code changes for our first milestone.

Milestone 1: The Data Pipeline (Plumbing)

Goal: Pass the theme data from `WorldManager` through the generator into the `Maze` object. No visual or gameplay changes yet.

1. **[ME]** I will provide the code changes for:
    - `RetroTheme.java` (add `getRandomTheme()`)
    - `Maze.java` (add `theme` field and setters/getters)
    - `IChunkGenerator.java` (update `generateChunk` signature)
    - `MazeChunkGenerator.java` & `ForestChunkGenerator.java` (implement new signature and call `maze.setTheme(theme)`)
    - `WorldManager.java` (I will provide the full, updated file with the new `levelThemes` map and modified `loadChunk`/`setCurrentLevel` methods).
2. **[YOU]** You will replace the content of these files in your IDE.

Milestone 1 Test: Compile and run the game. Play as normal. The game should not crash. No new functionality will be visible, but the backend data will now be flowing.

Milestone 2: The Visual Change (Aesthetics)

Goal: Make the newly generated levels *look* different based on their theme.

1. **[YOU]** I will ask you for `core/src/main/java/com/bpm/minotaur/rendering/FirstPersonRenderer.java` (or its equivalent, like `WorldRenderer`).
2. **[ME]** I will provide the changes for the renderer to read `player.getMaze().getTheme()` and use its colors.
3. **[YOU]** You will implement the changes.

Milestone 2 Test: Compile and run. When you trigger a level change (we'll manually add a debug key for this at first, bound to 'L'), the new level should render with a new, random color scheme.

Milestone 3: The Gameplay Change (Spawns & Loot)

Goal: Make the monsters and loot match the level's theme.

1. **[YOU]** I will ask for:
    - `core/src/main/java/com/bpm/minotaur/managers/SpawnManager.java`
    - `core/src/main/java/com/bpm/minotaur/gamedata/LootTable.java`
    - `core/src/main/java/com/bpm/minotaur/gamedata/ItemModifier.java` (or `ModifierType.java`, to see what modifiers exist)
2. **[ME]** I will provide the changes to update the spawn/loot methods to accept the `theme`. I will also create a new helper class, `ThemeDataManager.java`, to hold the `Theme -> ModifierType` mapping.
3. **[ME]** I will update `MazeChunkGenerator.java` again to pass the theme to the `SpawnManager` and `LootTable` calls.
4. **[YOU]** You will implement these changes.

Milestone 3 Test: Compile and run. Go to a level. If it's RED, you should encounter fire-based enemies. If it's BLUE, ice-based, etc.

Milestone 4: The Trigger (Ladders)

Goal: Connect the "use ladder" action to the level descent logic.

1. **[ME]** Using the `GameScreen.java` you provided earlier, I will pinpoint the "use" logic (e.g., checking `Input.Keys.E`).
2. **[ME]** I will provide the full logic for `GameScreen` to:
    - Check if the player is on a `LADDER` tile.
    - If so, execute the level-change logic (clear chunks, set new level, load new chunk, transition player).
3. **[YOU]** You will implement this logic in `GameScreen.java`.

Milestone 4 Test: Compile and run. Find a ladder in the maze, use it, and you should descend to a new, randomly-themed level with matching aesthetics and enemies.