Design Document: Biome Fog of War & Seamless Chunk Transitions

Date: 11/03/2025 **Author:** Gemini (Architect) **Status:** DRAFT

1. Overview

This document outlines the design for two major new features intended for non-maze biomes (e.g., `Biome.FOREST`, `Biome.PLAINS`):

1. **Configurable Fog of War (FOW):** A biome-specific fog effect that limits the player's view distance, integrating with the existing raycasting renderer.
2. **Seamless Chunk Transitions (SCT):** An overhaul of the chunk transition logic that removes the current "walled" chunks with gates and replaces them with an "open world" feel. This is achieved by allowing transitions on any passable border tile, supported by a new **Proactive Chunk Loading (PCL)** system to ensure the next chunk is loaded before it becomes visible to the player.

These features will significantly enhance player immersion in "wilderness" biomes, shifting them from a room-based dungeon crawl to a seamless exploration experience.

2. Goals & Scope

Goals

- Implement a configurable Fog of War (distance, color) that is managed on a per-biome basis.
- Remove the immersion-breaking "gates" and boundary walls from non-maze biomes.
- Implement a seamless transition system where players can walk from one chunk to another without a loading screen or "gate" interaction.
- Proactively load adjacent chunks just before they would be revealed by the FOW, hiding the loading process from the player.

Scope

- **In-Scope:**
    - Modifications to `Biome.java` to store new properties.
    - Modifications to `ForestChunkGenerator.java` to remove gate generation.
    - Modifications to `GameScreen.java` (or player controller) to handle PCL triggers and new transition logic.
    - Modifications to `WorldManager.java` to support new transition handling.

    - Significant refactoring of `FirstPersonRenderer.java` to support FOW and multi-chunk rendering.
- **Out-of-Scope:**
    - Applying this system to `Biome.MAZE`. The maze will retain its classic gate-and-wall structure.
    - Dynamic fog (e.g., weather effects, time-of-day).
    - Generation of scenery that straddles chunk boundaries (a known limitation of the current tile-based generator).

---

3. Feature 1: Biome-Specific Fog of War (FOW)

This feature will apply a distance-based fog effect to the 3D rendering, with parameters controlled by the biome the player is currently in.

3.1. Data Model (`Biome.java`)

The `Biome` enum will be updated to store properties for these new systems.

Java
```java
// core/src/main/java/com/bpm/minotaur/generation/Biome.java
package com.bpm.minotaur.generation;

import com.badlogic.gdx.graphics.Color;

public enum Biome {
    // Special static biomes
    MAZE(false, false, 100, null), // isSeamless, hasFog, fogDistance, fogColor

    // Wilderness biomes
    FOREST(true, true, 12, new Color(0.1f, 0.2f, 0.1f, 1.0f)),
    DESERT(true, true, 20, new Color(0.7f, 0.6f, 0.4f, 1.0f)),
    PLAINS(true, true, 25, new Color(0.3f, 0.5f, 0.2f, 1.0f)),
    MOUNTAINS(false, false, 100, null), // Not seamless, impassable
    LAKELANDS(true, true, 15, new Color(0.4f, 0.4f, 0.7f, 1.0f)),
    OCEAN(false, false, 100, null);  // Not seamless, impassable

    // --- New Properties ---
    private final boolean isSeamless;
    private final boolean hasFogOfWar;
    private final int fogDistance;
    private final Color fogColor;
    Biome(boolean isSeamless, boolean hasFog, int fogDistance, Color fogColor) {
        this.isSeamless = isSeamless;
        this.hasFogOfWar = hasFog;
        this.fogDistance = fogDistance;
        this.fogColor = fogColor;
    }

    public boolean isSeamless() { return isSeamless; }
    public boolean hasFogOfWar() { return hasFogOfWar; }
    public int getFogDistance() { return fogDistance; }
    public Color getFogColor() { return fogColor; }
}
```

3.2. Rendering Logic (`FirstPersonRenderer.java`)

The renderer, which performs the raycasting, is the ideal place to apply fog.

1. **Shader Modification:** The wall/scenery fragment shader (`.frag`) will be modified to accept new uniforms:
   - `uniform bool u_fogEnabled;`
   - `uniform float u_fogDistance;`
   - `uniform vec3 u_fogColor;`
   - `uniform float u_rayDistance;` (This will be passed from the vertex shader, which gets it from the CPU).

Shader Logic: In the fragment's `main()` method, the final `gl_FragColor` will be mixed with the `u_fogColor` based on the ray's distance.

OpenGL Shading Language
```
// Example GLSL snippet
if (u_fogEnabled) {
    float fogFactor = smoothstep(u_fogDistance - 2.0, u_fogDistance, u_rayDistance);
    gl_FragColor.rgb = mix(gl_FragColor.rgb, u_fogColor, fogFactor);
}
```

2.
3. **Renderer Modification (`FirstPersonRenderer.java`):**
   - The renderer will fetch the current biome's properties from `WorldManager.getBiomeManager().getBiome(currentPlayerChunkId)`.
   - Before rendering the 3D scene, it will set the shader uniforms (`u_fogEnabled`, `u_fogDistance`, `u_fogColor`) based on the biome's properties.

    ○ When drawing a wall slice or scenery, it must pass the `ray.distance` to the shader.

---

4. Feature 2: Seamless Chunk Transitions (SCT)

This feature removes the artificial barriers between chunks, allowing the player to walk freely from one to the next.

4.1. Generator Modification (`ForestChunkGenerator.java`)

The `generateChunk` method currently hard-codes a call to `spawnTransitionGates`.

- **Action:** In `ForestChunkGenerator.java`, `generateChunk()` method (around line 257):
    - **Remove:** The line `spawnTransitionGates(maze, this.finalLayout, chunkId);` must be deleted.
- **Result:** The generator will no longer create gates, and since its tile data does not include '#' boundary walls, the chunk edges will be open, defined only by the trees, rocks, or floor tiles at the edge of the layout.

4.2. Transition Logic (`GameScreen.java`)

The player movement logic must be updated to detect and handle border crossings. This logic should run in the `GameScreen.update()` method *before* the player's move is finalized.

Java
```
// Psuedo-code for GameScreen.update()

// Get player's intended move (e.g., player wants to move NORTH)
GridPoint2 playerPos = player.getPosition();
Direction moveDirection = player.getMoveDirection();
Biome currentBiome =
worldManager.getBiomeManager().getBiome(worldManager.getCurrentPlayerChunkId());

if (currentBiome.isSeamless() && player.isAtBorder(moveDirection)) {

    // 1. Calculate Target
    GridPoint2 targetChunkId = worldManager.getAdjacentChunkId(moveDirection);
    GridPoint2 targetPlayerPos = player.getTargetPositionInNewChunk(); // e.g., (x, 0) if moving NORTH

    // 2. Check Passability (relies on PCL already having loaded the chunk)
    Maze targetChunk = worldManager.getChunk(targetChunkId);

    if (targetChunk != null && targetChunk.isPassable(targetPlayerPos.x, targetPlayerPos.y)) {

        // 3. Execute Transition
        // This is a new method to handle swapping the player's context.
        worldManager.transitionPlayerToChunk(player, targetChunkId, targetPlayerPos);

        // Prevent normal movement logic from running this frame
        return;
    } else {
        // Don't transition; block movement (ran into a tree on the border)
        player.blockMove();
    }
}

// ... normal game update logic ...
```

4.3. New Method (`WorldManager.java`)

We will add a new method to `WorldManager` to handle the player's transition.

Java

```
// In core/src/main/java/com/bpm/minotaur/managers/WorldManager.java
public void transitionPlayerToChunk(Player player, GridPoint2 newChunkId, GridPoint2
newPlayerPos) {
    Maze newMaze = loadedChunks.get(newChunkId);

    // This assumes the newMaze is already loaded by the PCL system
    if (newMaze == null) {
        Gdx.app.error("WorldManager", "CRITICAL: Player tried to transition to a chunk that was
not loaded: " + newChunkId);
        // As a fallback, load it now (will cause a lag spike)
        newMaze = this.loadChunk(newChunkId);
        if (newMaze == null) return; // Transition failed
    }

    // Update WorldManager's state
    this.currentPlayerChunkId = newChunkId;

    // Update Player's state
    player.setMaze(newMaze); // Update player's internal maze reference
    player.setPosition(newPlayerPos); // Set new grid position
    // The GameScreen will also need to update its 'currentMaze' reference
    // This can be done via a return value or a listener.
}
```

---

5. Feature 3: Proactive Chunk Loading (PCL)

This is the most critical component for ensuring the "seamless" part of the transition. The system must load adjacent chunks *before* they enter the player's FOW-limited view.

5.1. The Trigger

The trigger for loading the next chunk must be based on the player's distance to the border relative to the fog distance.

- **Goal:** Load the chunk when the player is 1 tile *before* the border becomes visible.
- **Player's View:** A player can see `fogDistance` tiles ahead.
- **Border Visibility:** The border becomes visible when the player is `fogDistance` tiles away.
- **Trigger Distance:** We must trigger the load when the player is at `fogDistance + 1` tiles from the border.
    - *Example: Fog = 4 tiles. Border at y=0 is visible at y=4. We must trigger the load at y=5.*

5.2. PCL Logic (`GameScreen.java`)

This logic will run in the `GameScreen.update()` method every frame.

Java
```
// Psuedo-code for GameScreen.update()

// ... after player movement ...
private void checkForProactiveChunkLoading() {
    Biome biome =
worldManager.getBiomeManager().getBiome(worldManager.getCurrentPlayerChunkId());

    // Only run this for seamless biomes
    if (!biome.isSeamless()) {
        return;
    }

    int triggerDistance;
    if (biome.hasFogOfWar()) {
        triggerDistance = biome.getFogDistance() + 1;
    } else {
        triggerDistance = 5; // Default trigger distance for seamless biomes without fog
    }

    GridPoint2 playerPos = player.getPosition();
    GridPoint2 currentChunkId = worldManager.getCurrentPlayerChunkId();

    // Check North
    if (playerPos.y >= Maze.MAZE_HEIGHT - 1 - triggerDistance) {
        worldManager.requestLoadChunk(new GridPoint2(currentChunkId.x, currentChunkId.y +
1));
    }
    // Check South
    if (playerPos.y <= triggerDistance) {
        worldManager.requestLoadChunk(new GridPoint2(currentChunkId.x, currentChunkId.y -
1));
    }
    // Check East
    if (playerPos.x >= Maze.MAZE_WIDTH - 1 - triggerDistance) {
        worldManager.requestLoadChunk(new GridPoint2(currentChunkId.x + 1,
currentChunkId.y));
    }
    // Check West
    if (playerPos.x <= triggerDistance) {
        worldManager.requestLoadChunk(new GridPoint2(currentChunkId.x - 1,
currentChunkId.y));
    }
}
```

5.3. Loading Method (`WorldManager.java`)

We create a new wrapper method `requestLoadChunk` to avoid re-loading chunks that are already loaded or in the process of being loaded.

Java
```
// In core/src/main/java/com/bpm/minotaur/managers/WorldManager.java
public void requestLoadChunk(GridPoint2 chunkId) {
    // Check cache first. If it's already loaded, do nothing.
    if (loadedChunks.containsKey(chunkId)) {
        return;
    }
    // Check for impassable biomes
    Biome biome = biomeManager.getBiome(chunkId);
    if (biome == Biome.OCEAN || biome == Biome.MOUNTAINS) {
        return;
    }

    // If we add async loading, this is where we'd check if it's "in-progress".
    // For now, we just call loadChunk directly.

    Gdx.app.log("WorldManager", "Proactively loading chunk: " + chunkId);
    loadChunk(chunkId); // This method already handles loading from file or generating.
}
```

---

6. Summary of Required Changes & Risks

6.1. Class-by-Class Changes

- `generation/Biome.java`:
  - **Add** fields: `isSeamless`, `hasFogOfWar`, `fogDistance`, `fogColor`.
  - **Update** enum constructor to accept and store these new values.
  - **Add** getters for all new fields.
- `generation/ForestChunkGenerator.java`:
  - **Remove** the call to `spawnTransitionGates(...)` from the `generateChunk(...)` method.
- `managers/WorldManager.java`:
  - **Add** new method: `transitionPlayerToChunk(Player player, GridPoint2 newChunkId, GridPoint2 newPlayerPos)`.
  - **Add** new method: `requestLoadChunk(GridPoint2 chunkId)` (which intelligently calls `loadChunk`).
  - **Add** new method (helper): `getAdjacentChunkId(Direction direction)`.
- `screens/GameScreen.java`:
  - **Add** call to `checkForProactiveChunkLoading()` in the `update()` loop.
  - **Implement** `checkForProactiveChunkLoading()` method (see 5.2).
  - **Modify** player movement logic to check for border transitions in seamless biomes (see 4.2).
  - **Update** `GameScreen`'s internal `currentMaze` reference after `worldManager.transitionPlayerToChunk` is called.
- `rendering/FirstPersonRenderer.java`:
  - **MAJOR REFACTOR REQUIRED.**
  - **Modify** fragment shader to accept and process fog uniforms.

    - **Modify** class to hold a reference to `WorldManager` instead of a single `Maze`.
    - **Modify** `drawMaze3D()` and all raycasting helper methods (`castRay`, etc.) to:
        - Operate in "world coordinates" (or translate between chunk-local and world).
        - Fetch the correct chunk data (`worldManager.getChunk(...)`) based on the ray's current position.
        - Fetch current biome properties from `WorldManager` and set fog shader uniforms once per frame.

6.2. Risks & Open Questions

1. **High Risk: Renderer Refactor.** The change to `FirstPersonRenderer` is non-trivial. It must be refactored to be "world-aware" rather than "chunk-aware." This is the most complex part of the implementation and must be tested thoroughly, especially at chunk boundaries.
2. **Performance:** `loadChunk()` involves I/O and procedural generation, which can cause a lag spike. The PCL trigger helps, but if generation is slow, the player may reach the border before the load is complete. *Future Consideration: Asynchronous chunk loading on a separate thread.*
3. **Memory Management:** The PCL system only *adds* chunks to `loadedChunks`. It never removes them. This will lead to unbounded memory growth. A new **Chunk Unloading** system will be required as a fast-follow. (e.g., "Unload any chunk more than 2 chunk-lengths away from the player").
4. **Visual Seam:** The tile-based generation will still create visible seams in scenery (e.g., a line of trees at `x=0` on one chunk and a line of trees at `x=31` on the previous). While the *transition* is seamless, the *visual* connection may not be. This is an accepted limitation of the current generator.