Procedural Terrain Generation

Gyan Prakash

November 14, 2024

1 Project Overview

This project aims to implement a dynamic 3D terrain generation system that utilizes procedural techniques to create terrain chunks in a vast, open world. The system dynamically loads and unloads terrain chunks based on the player's position, ensuring that only the necessary parts of the world are rendered at any given time. The terrain is generated using Perlin noise, creating natural-looking variations in elevation. The chunks are managed in a chunking system, optimizing rendering performance and memory usage by only rendering chunks within a certain distance (render distance) from the player.

1.1 Goal

The primary goal of the project is to create a procedural terrain generation system using the Python programming language and the `pyray` library (Raylib Python bindings). The project is designed to be lightweight and efficient, capable of generating and rendering large terrains in real time, while also minimizing memory usage by unloading terrain chunks that are no longer within the player's view.

2 Key Components

2.1 Terrain Chunk System

The terrain in this project is divided into manageable chunks, which are 3D grid sections of the world that are rendered individually. This chunking

system allows for efficient memory and rendering management, as only the chunks within a defined *render distance* from the player are loaded and drawn at any given time.

Each chunk has a fixed size (`CHUNK_SIZE = 64`), and the height of the terrain within each chunk is defined by the `CHUNK_HEIGHT = 32` constant. Chunks are generated procedurally using Perlin noise, which ensures the terrain has natural-looking variations in height.

When the player moves through the world, chunks are dynamically loaded based on the player's position, and chunks outside of a predefined *render distance* are unloaded to conserve memory and optimize performance.

2.2 Water Generation

To enhance the realism of the terrain, a water layer is added at the base of the terrain. This is done by introducing a flat blue layer below a certain height threshold. The water layer is rendered as a 3D cube with a height of 1 unit and is positioned at appropriate length to generate ideal looking terrain. The water level can be varied to generate features like islands or waterlogged landmasses

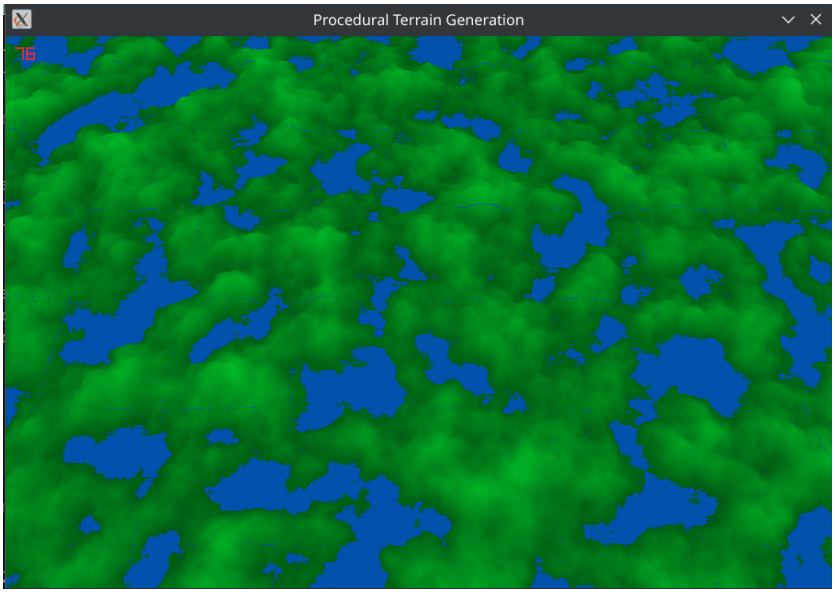

Figure 1: Waterlogged Landmass

The water layer is represented visually by drawing a blue cube at the base of the terrain for each chunk, giving the effect of an ocean or lake depending on the elevation of the surrounding terrain. This helps simulate large bodies of water within the 3D world, which are consistent with the procedural generation of the landmasses.

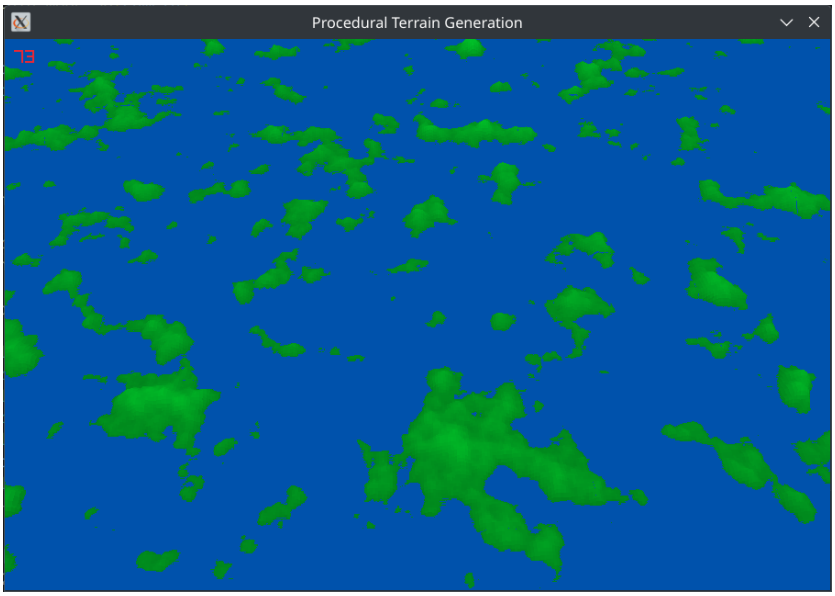

Figure 2: Islands

2.3 Camera Controls

The project includes a first-person camera controlled by the player. This allows the user to move through the 3D world and explore the procedurally generated terrain. The camera controls include:

- **WASD keys**: For movement in the X and Z axes (forward, backward, left, right).
- **SHIFT key**: For descending vertically (moving down).
- **SPACE key**: For ascending vertically (moving up).
- **Mouse movement**: For looking around in the 3D space.

- **ESC key**: To toggle the cursor visibility.
- **F11 key**: To toggle fullscreen mode.

The camera operates in a first-person perspective, giving the user full control over their movement and viewing angle within the 3D environment.

2.4 Render Distance

To optimize performance and prevent rendering unnecessary parts of the world, the system only renders chunks within a certain *render distance* from the player. The render distance is defined by the `render_distance` variable, which is set to `8` in the provided code.

This means that the system will load and render all chunks within a square region centered on the player's position, with a width and height equal to $2 \times$ `render_distance`. Chunks that fall outside of this range are unloaded to free up memory and processing power.

3 Functions and Logic

3.1 locIsEq

The `locIsEq` function checks if two `Vector3` locations are equal. It compares the `x`, `y`, and `z` components of two vectors to determine whether the locations are the same. This function is used to prevent loading duplicate chunks by checking if a chunk with a given location already exists in the list of loaded chunks.

```
def locIsEq(loc1: rl.Vector3, loc2: rl.Vector3) -> bool:
    return (loc1.x == loc2.x and loc1.y == loc2.y and loc1.z ==
    ↪  loc2.z)
```

3.2 isChunkIn

The `isChunkIn` function determines whether a chunk should remain active by checking if its location is within the player's render distance. If the chunk is outside of this range, it is unloaded to free up resources.

```
def isChunkIn(chunk: cg.Chunk, pos: rl.Vector3,
↪  render_distance: int):
    if(chunk.location.x >= pos.x - cg.CHUNK_SIZE *
    ↪  render_distance and chunk.location.x <= pos.x +
    ↪  cg.CHUNK_SIZE * render_distance):
        if(chunk.location.z >= pos.z - cg.CHUNK_SIZE *
        ↪  render_distance and chunk.location.z <= pos.z +
        ↪  cg.CHUNK_SIZE * render_distance):
            return True
    cg.unloadChunk(chunk)
    return False
```

3.3 generateChunk

The `generateChunk` function is responsible for generating a new terrain chunk at a specific position. It uses Perlin noise to generate the heightmap for the terrain and combines two layers of noise using the `addImages` function. It then generates a mesh from the heightmap and loads a texture for the chunk.

```
def generateChunk(position: rl.Vector3) -> Chunk:
    offset = getChunkPos(position)
    perlin_image_1 = rl.gen_image_perlin_noise(
        int(CHUNK_SIZE), int(CHUNK_SIZE), int(offset.x),
        ↪  int(offset.z), 5)
    perlin_image_2 = rl.gen_image_perlin_noise(
        int(CHUNK_SIZE), int(CHUNK_SIZE), int(offset.x),
        ↪  int(offset.z), 3)
    perlin_image_final = ai.addImages(perlin_image_1,
    ↪  perlin_image_2)

    mesh = rl.gen_mesh_heightmap(perlin_image_final,
    ↪  rl.Vector3(
        CHUNK_SIZE, CHUNK_HEIGHT, CHUNK_SIZE))
    model = rl.load_model_from_mesh(mesh)
    texture = rl.load_texture_from_image(perlin_image_final)
```

```
    ↪  model.materials[0].maps[rl.MaterialMapIndex.MATERIAL_MAP_ALBEDO].texture
    ↪  = texture

    rl.unload_image(perlin_image_1)
    rl.unload_image(perlin_image_2)

    return Chunk(model, texture, offset)
```

The scale field provided when generating the perlin noise can be changed to generate relatively flat or mountainous terrain.

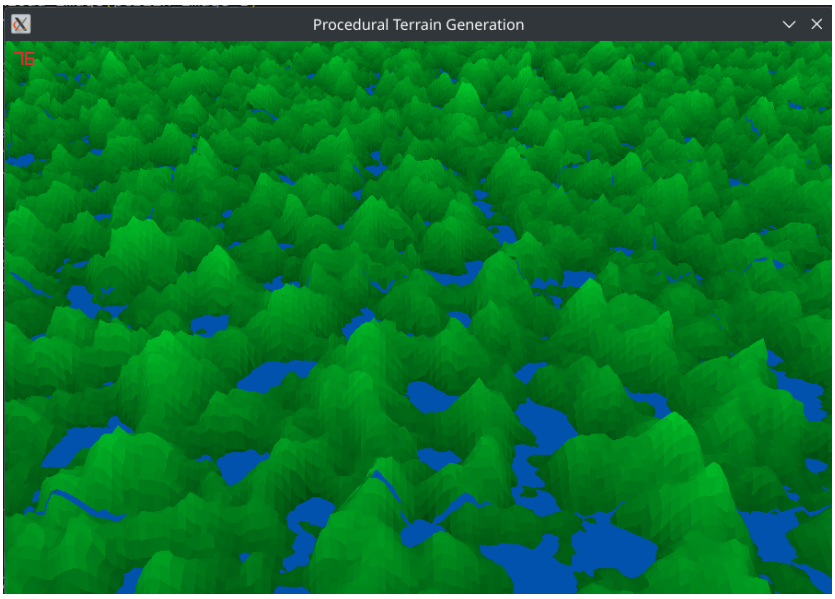

Figure 3: Mountains

3.4 renderChunk

The `renderChunk` function is responsible for drawing each chunk in the 3D environment. It draws the terrain model and a visual representation of the chunk's boundaries.

```
def renderChunk(chunk: Chunk) -> None:
    rl.draw_model(chunk.model, chunk.location, 1, rl.GREEN)
```

```
rl.draw_cube(rl.Vector3(chunk.location.x + CHUNK_SIZE // 2,
↪  16,
    chunk.location.z + CHUNK_SIZE // 2), CHUNK_SIZE, 1,
    ↪  CHUNK_SIZE, rl.BLUE)
```

4 Optimization Techniques

4.1 Chunk Unloading and Loading

One of the main challenges in this project is managing the performance of the system as it dynamically loads and unloads chunks. To mitigate performance issues, chunks that are no longer within the player's render distance are unloaded using the `isChunkIn` function, which checks the distance between the chunk and the player.

By only keeping the necessary chunks in memory and unloading those outside of the render distance, the system ensures that resources are used efficiently. This approach minimizes the number of active chunks in the scene and helps prevent unnecessary computation for chunks that are not visible to the player.

4.2 Perlin Noise Optimization

The use of Perlin noise to generate terrain ensures that the terrain looks natural, but it can also be computationally expensive. In this project, two layers of Perlin noise are generated with different frequencies and combined to create a more complex heightmap. The use of multiple layers of Perlin noise helps create more varied terrain features, but the implementation could be further optimized by caching noise results or using more efficient noise generation techniques.

5 Conclusion

This project demonstrates a dynamic and efficient approach to procedural terrain generation in 3D. The use of chunking allows for easy memory management, and the render distance technique ensures that only the necessary parts of the world are rendered, optimizing performance. The Perlin noise-

based terrain generation creates natural-looking landscapes, and the first-person camera allows for immersive exploration of the generated world.

The system could be further improved by optimizing the Perlin noise generation and adding more sophisticated terrain features such as vegetation, water bodies, or dynamic weather. Additionally, future versions of the project could include more advanced rendering techniques, such as Level of Detail (LOD) to further enhance performance and visual quality.

Overall, this project successfully demonstrates the core concepts of procedural terrain generation, chunk management, and efficient rendering, making it a solid foundation for creating large, dynamic 3D worlds.