Graphics Visualization - Final Project

Michael Brusegard

April 2025

1 Project Topic

This project implements a block-based terrain renderer in Rust using OpenGL 4.1, inspired by Minecraft. The renderer features dynamic terrain generation, a day/night cycle with dynamic lighting, and various graphics optimization techniques. While the initial proposal included volumetric lighting (god rays), I ended up pivoting focusing on shadows and lighting based on the response to the topic submission, which was more than challenging enough!

The final implementation includes:

- Procedural terrain generation using noise
- Dynamic chunked terrain loading with multi-threaded generation
- Shadow mapping with PCF (Percentage Closer Filtering)
- LOD system for distance-based mesh optimization
- Day/night cycle with a moving sun and moon
- Dynamic sky coloring with stars that appear at night
- Specular highlights and directional lighting

2 How the Implementation Achieves its Goal

The renderer integrates several systems with computer graphics techniques. How they work is outlined below.

2.1 Shadow Mapping System

Dynamic shadows use a standard two-pass shadow mapping technique with enhancements for stability and quality. See Figure 1 and Figure 2 for examples.

1. **Light Space Definition:** A directional light source's view is defined. An orthographic projection is used, aligned to world-space texel increments derived from the shadow map resolution for stability. The combined matrix is the `lightSpaceMatrix`.
2. **Depth Pass:** Opaque scene geometry within the shadow distance is rendered from the light's perspective. A minimal vertex shader and empty fragment shader are used; hardware rasterizes depth into the shadow map texture. Front-face culling helps self-shadowing.
3. **Scene Pass:** The scene is rendered normally from the camera. Fragment world positions are transformed by `lightSpaceMatrix` into light space for shadow map sampling.

4. **Shadow Calculation & PCF:** Fragment depth in light space (`currentDepth`) is compared against the shadow map's depth (`closestDepth`). A small bias helps prevents "shadow acne". Percentage-Closer Filtering (PCF) uses a 3x3 kernel of samples around the projected coordinates. The averaged depth comparison results produce a smoothed shadow value creating softer edges.

2.2 Level of Detail (LOD) System

The LOD system adjusts chunk mesh complexity based on distance to maintain performance. See Figure 5.

1. **Distance Calculation:** Squared Euclidean distance in the XZ plane between the camera's and each loaded chunk's coordinates is calculated.
2. **LOD Level Selection:** Based on distance thresholds, chunks are assigned one of four LOD levels: `LOD1` (full detail, 1x1 blocks), `LOD2` (2x2x2 simplification), `LOD4` (4x4x4), or `LOD8` (8x8x8).
3. **Mesh Generation Trigger:** A chunk mesh regenerates if: it's `ChunkDirty`, its required LOD level changes, or the required LOD level for any immediate neighbor changes (crucial for correct culling at boundaries).
4. **Downsampling Algorithm:** For each LOD level, the `MeshGenerator` first creates lower-resolution data. For each larger voxel, it determines a representative block type by examining original blocks, counting exposed blocks, and selecting the most frequent. If none are exposed, it uses internal blocks. This preserves visible surface appearance.
5. **Mesh Construction:** Mesh generation operates on this (potentially downsampled) data, building faces only between adjacent voxels according to culling rules. Vertices are scaled by the LOD factor to repeat textures.

2.3 Terrain Generation and Management

Handling a large world requires generating and loading parts of it efficiently.

1. **Procedural Generation:** Terrain is created using noise functions (like Simplex and Perlin) to determine height, biome, and block types.
2. **Dynamic Loading:** The system tracks camera position and identifies nearby "chunks" to load.
3. **Background Tasks:** Heavy work like generating or loading chunks happens on separate background threads to keep the game smooth.
4. **Loading and Creation:** Workers check for cached chunk data; if not found, they generate it using noise. Results are sent back to the main process.

5. **Visual Representation (Meshing):** Chunk data received on the main thread triggers mesh generation requests sent to background workers. Workers create the visual mesh (with LOD) and return it for rendering.

2.4 Lighting and Atmospheric System

Lighting creates dynamic environmental effects based on a simulated time of day.

1. **Time Simulation:** A global variable representing the time of day progresses each frame, controlled by a configured speed.
2. **Light Calculation:**
   - **Direction:** The primary light direction is calculated based on the time of day, simulating the sun's and moon's movement.
   - **Color & Intensity:** Sky color and overall light level are determined by interpolating between key colors and intensity values at different times of the day (like midnight, noon, sunrise, sunset).
   - **Ambient Light:** The final ambient light color is derived from the calculated sky color and overall light level, ensuring a minimum level of ambient contribution.
   - **Direct Light:** The Phong reflection model is applied in the fragment shader. Diffuse lighting is based on the surface normal and light direction (Lambertian). Specular lighting considers the reflection of the light and the view direction, controlled by a shininess uniform value. The final color combines the texture color with ambient and shadowed direct lighting:
     `textureColor * (ambient + shadow * (diffuse + specular))`.
     See Figure 3 and Figure 4.
3. **Celestial Bodies:** The sun and moon are rendered as textured quads using the main shader, with a flag indicating they are celestial objects. They are positioned far away along the light/moon direction and always face the camera (billboarded). Depth testing is disabled so they appear behind scene geometry.
4. **Starfield:** Stars are rendered with a separate shader program during nighttime. They are drawn as points (`gl_PointSize`) positioned based on random directions on a sphere. A twinkling effect is created in the fragment shader using time and a unique value per star. Their visibility fades smoothly based on the `nightFactor`. See Figure 6.

3 Problems

I encountered several problems during the project.

1. **Mesh Update Performance:** Updating chunk meshes on the GPU individually is inefficient, especially with many simultaneous updates. The large number of draw calls makes the terrain mesh update very slowly. A more efficient approach would involve batching updates by combining data from multiple chunks into larger buffers or using techniques like indirect drawing to reduce draw calls. But this was very complicated so I was not able to implement it.

2. **Level of Detail (LOD) Seam Artifacts:** Implementing multiple LODs caused visual artifacts at boundaries between different detail levels. Lower LOD vertices don't align perfectly with higher-detail neighbors at the chunk borders, creating visible gaps or seams. Addressing this requires "stitching" edge vertices or generating transition meshes. I tried fixing the issue but I ended up just rendering unnecessary faces between chunk borders, so in the end I had to move on.

3. **Shadow Mapping Artifacts:** Implementing shadow mapping introduced persistent artifacts:

    - **Acne and Peter Panning:** Eliminating shadow acne (surfaces incorrectly shadowing themselves) and peter panning (shadows disconnected from casters) across all lighting angles proved difficult. A compromise bias value (`0.009`) was chosen to prevent flickering, but doesn't perfectly solve both.
    - **Front-Face Culling Gaps:** Enabling front-face culling to mitigate peter panning caused large, incorrect gaps within the shadows of complex shapes, as only back faces were used for depth.
    - **Depth Inversion Workaround:** Sampled shadow map depth values appeared inverted; they increased with proximity instead of distance. This flipped the depth comparison logic. A temporary fix involved swapping the near and far plane values in the light's orthographic projection matrix. This re-flipped the depth gradient in the projection step, making standard comparison work. While this corrected the visuals, it suggests an underlying mismatch in depth conventions or the shadow map's depth format, as this workaround shouldn't typically be necessary.

4. **Multi-threading Complexity:** Implementing efficient background processing introduced the inherent difficulties of multi-threaded programming. Managing shared resources, ensuring thread safety, and handling workers was very hard, consuming development time not directly related to graphics programming itself.

5. **Project Topic:** After doing this project I learned that my choice of project topic was flawed from the start. Choosing a block based renderer was more difficult than expected due to all of the necessary CPU programming for chunks, threads, generation and the vast amounts of culling. In

the future I would instead choose a topic that requires less CPU programming and is mostly shaders. On the bright side I learned a lot :).

4 Advantages, limitations and effective use case

Through implementing this project, I gained practical insight into several graphics techniques. Shadow mapping provides dynamic shadows but is sensitive to bias and prone to artifacts; PCF helps smooth edges. LOD is essential for performance in large worlds but can cause visual popping and requires careful handling of seams. Procedural noise generation is excellent for creating vast, varied terrain efficiently but needs multi-threading for real-time loading. Multi-threading itself is vital for offloading CPU-bound tasks but adds significant complexity in managing shared data and thread safety. The Phong reflection model is a simple, fast lighting method suitable for basic real-time rendering, though it's a simplified approximation of real-world lighting. Effectively using these techniques involves balancing their performance benefits against their visual limitations and implementation complexity. This also includes handling specific material properties like transparency, as shown in Figure 7, or water.

The biggest limitation to most of the techniques is the complexity and the time it takes to implement them and understand them. Also fixing all the artifacts that occur when implementing them is another limitation.

5 Resources

Figuring things out for this project involved a mix of approaches. I started with using LearnOpenGL for core concepts. When I hit specific problems or needed to learn a new technique, I research online, digging through articles, forums, and code examples. Large Language Models (LLMs) were also a big help for getting explanations and figuring out tricky parts. But honestly, a huge chunk of my learning was just jumping into the code and messing around. Tying different ways to do things, and seeing what happened when things broke. It helped a lot that I am also taking the software architecture course which was a great resource. From there I employed an ECS (Entity Component Architecture) which made the code a lot cleaner. I also used rust libraries like `noise` for the simplex and perlin noise and `hecs` for the ECS so I did not have to implement that myself. For OpenGL I used `gl-rs` for rust bindings to OpenGL.

For more screenshots from development, you can check out the repository: https://github.com/michaelbrusegard/Meinkraft

6 Appendix

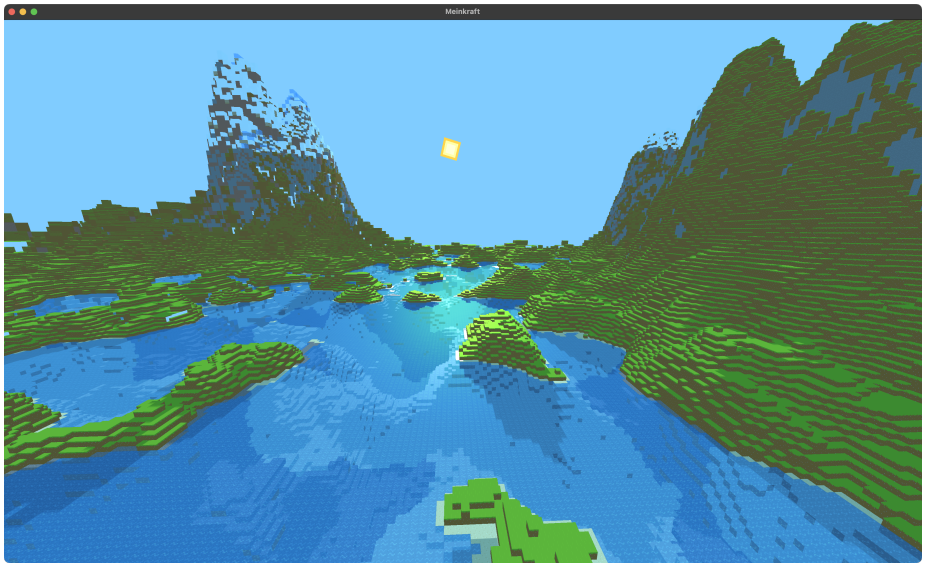

Figure 1: Shadows

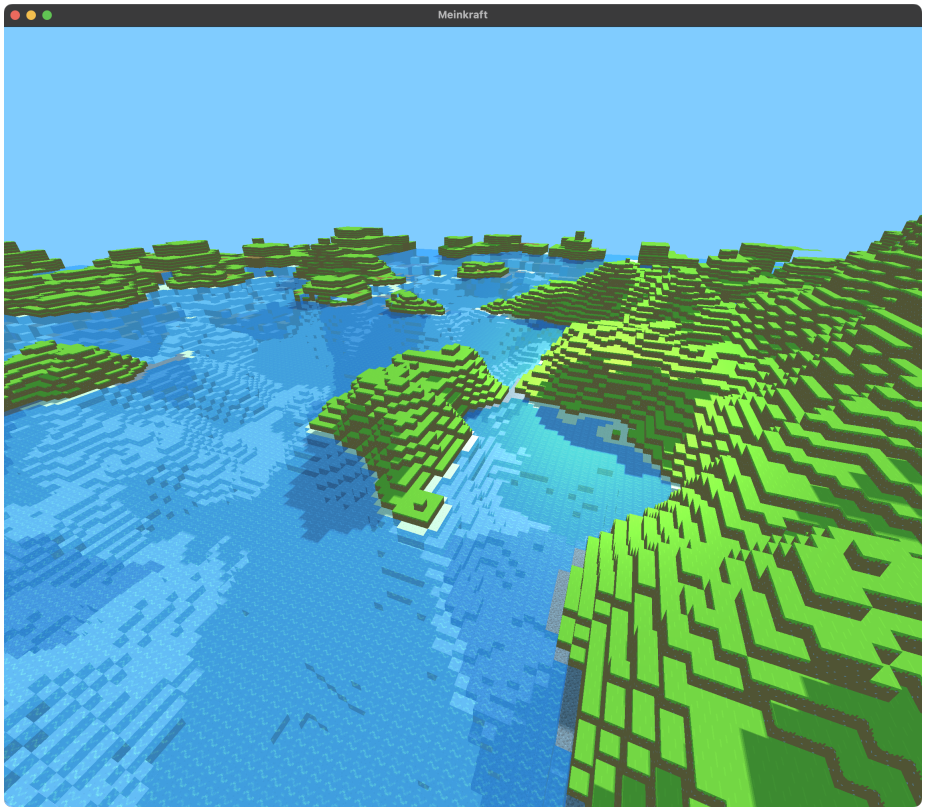

Figure 2: Shadows 2

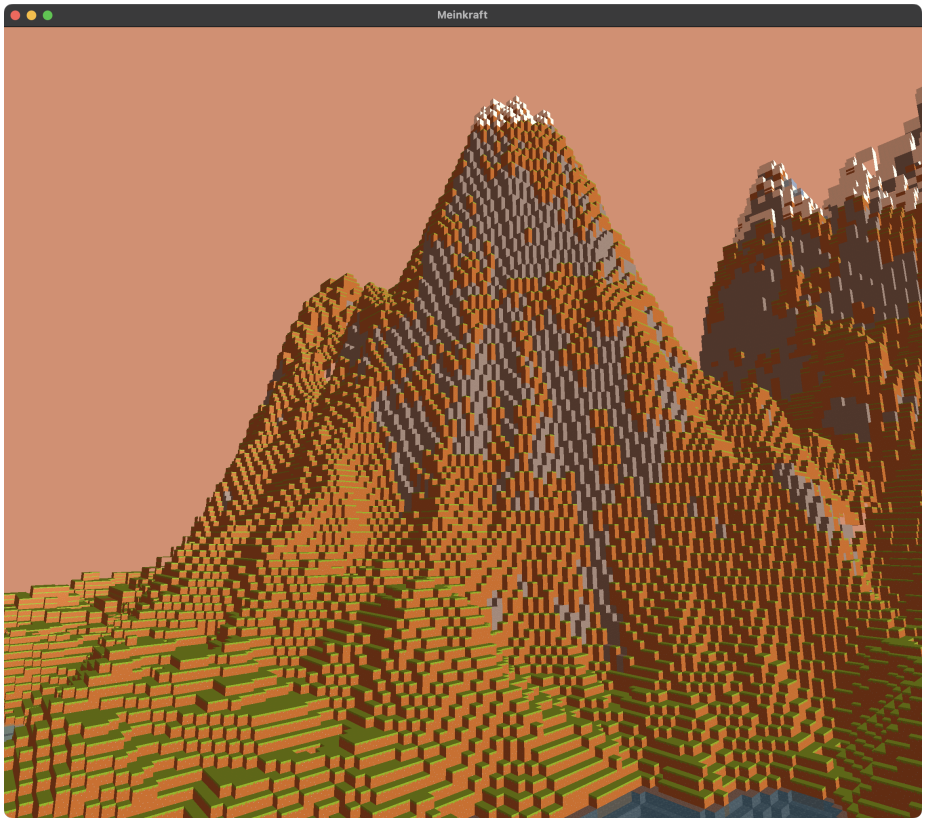

Figure 3: Ambient and direction lighting

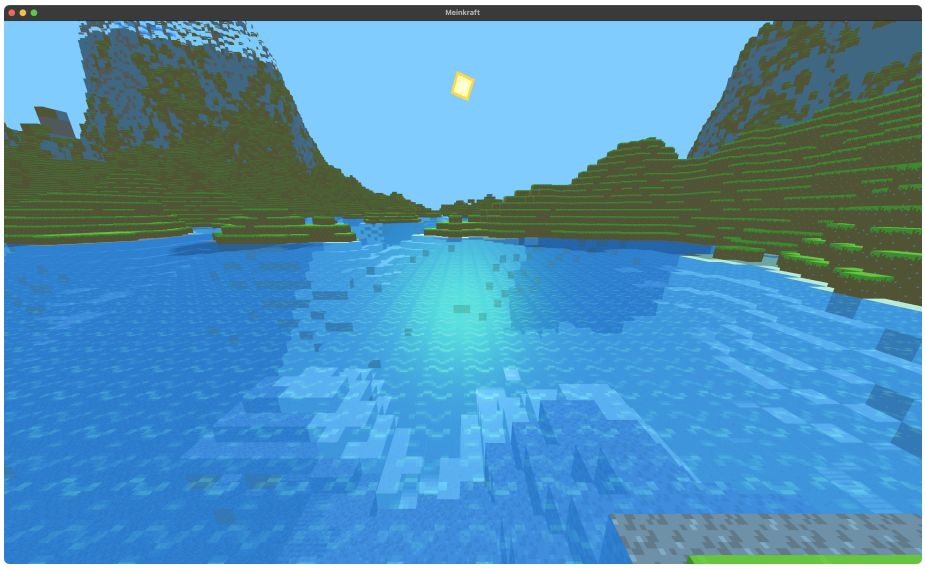

Figure 4: Specular light in the water

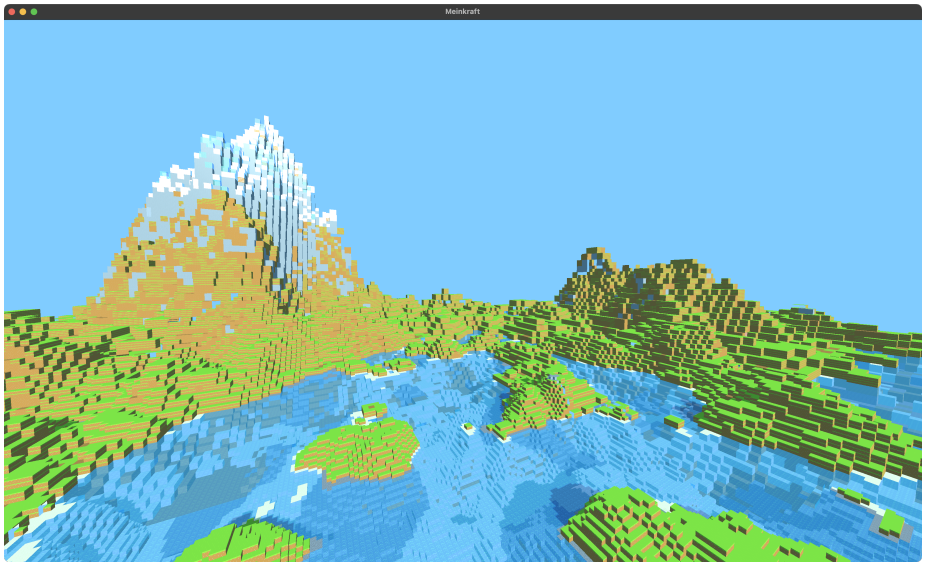

Figure 5: Level of Detail (here the LOD levels are set a lot closer to be very noticeable, the blocks get very big further away)

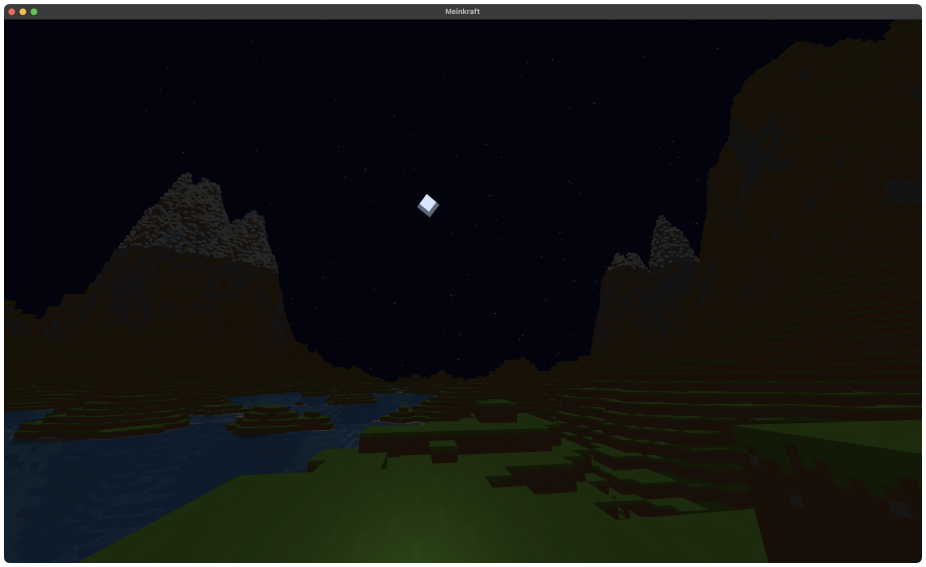

Figure 6: Starfield (you may need to zoom in on the picture to see the stars)

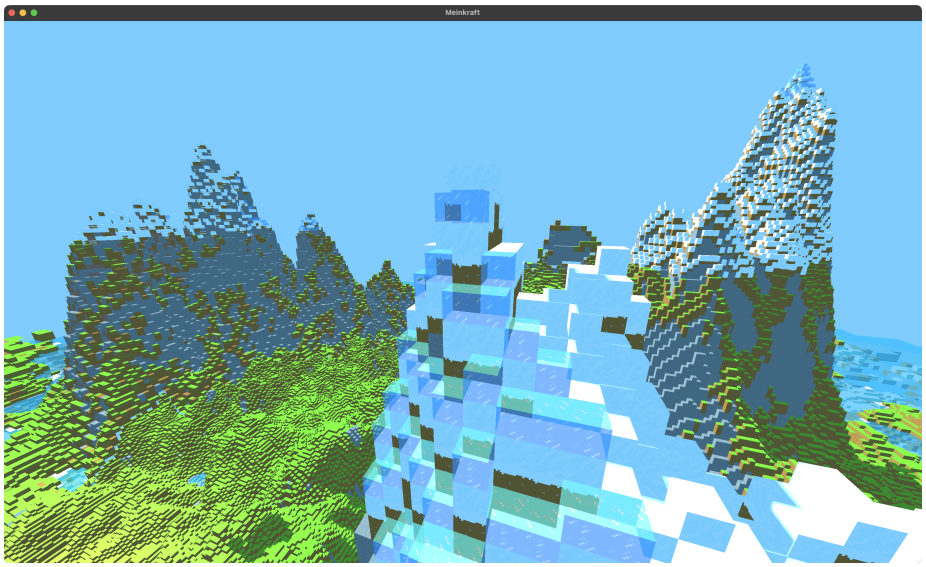

Figure 7: Transparency in the ice (the faces behind it are still rendered)