Advanced Shader Programming Report

Author: Chaouki Mabrouki

Advanced Shaders Programming: IMAT3907

Report

Contents

Report

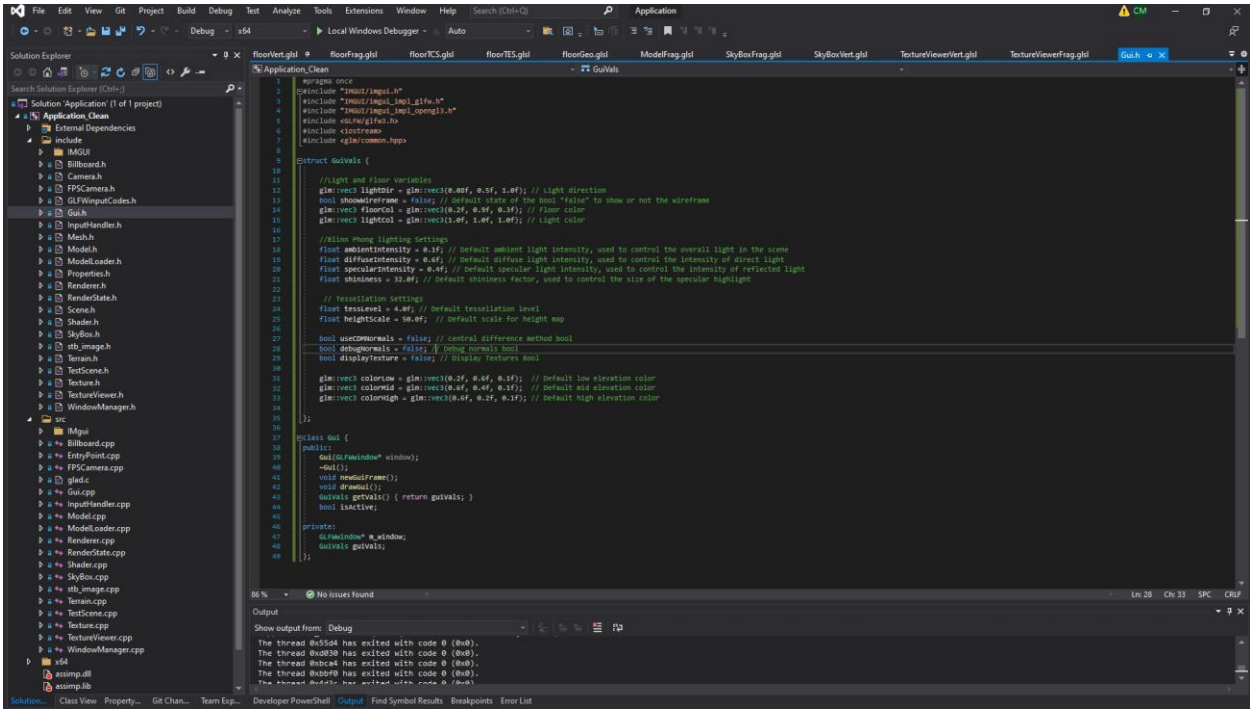
Section 1: GUI Class Integration and Interaction with TestScene Class

During the development of this project, one of the most crucial aspects was the integration of the GUI class with the TestScene class. The Gui class was built using the ImGui library, which allowed me to create an intuitive interface for controlling various parameters in the scene, such as tessellation levels and lighting properties. This interface wasn't just a tool for debugging—it was an essential part of the project that enabled real-time interaction with the scene.

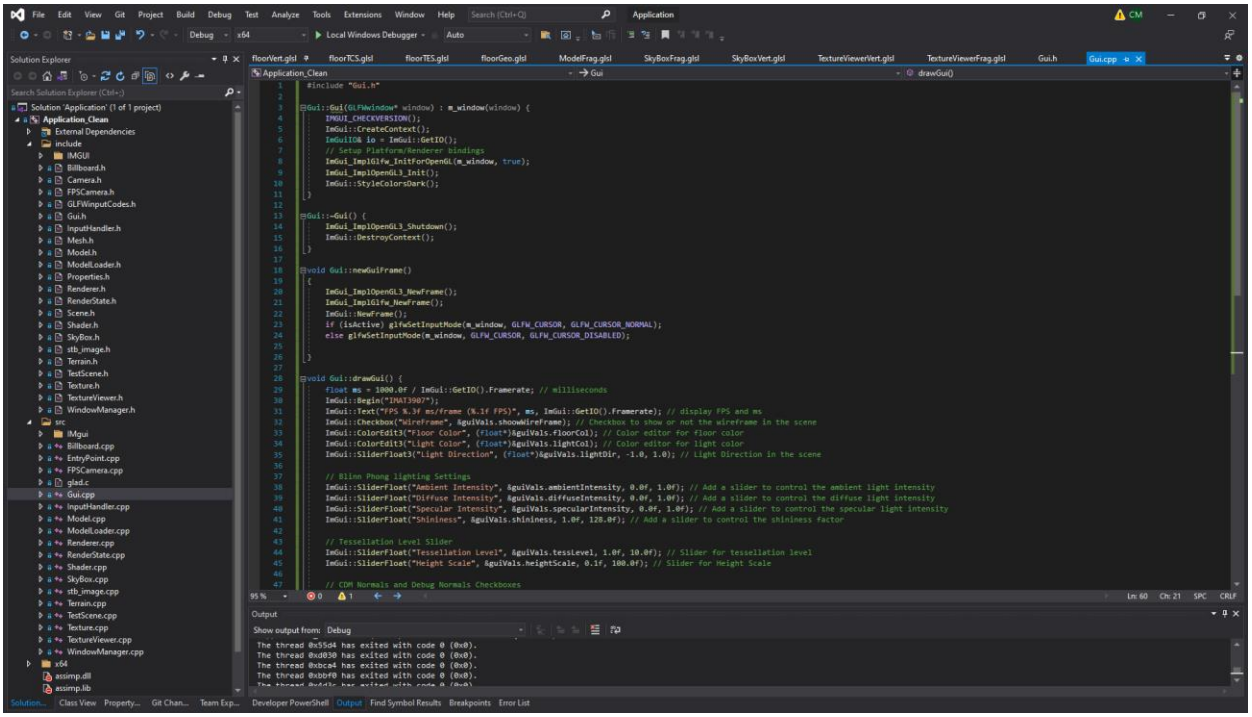
At the heart of the Gui class is the newGuiFrame() function, which initializes a new ImGui frame at the start of each rendering cycle. This setup is crucial because it resets the state, preparing the GUI to receive inputs and render controls for the current frame. Following this, the drawGui() function comes into play. It's here that all the sliders, checkboxes, and other controls are rendered on the screen.

What's fascinating about this setup is how these controls are directly linked to variables in the TestScene class. For example, when adjusting the tessellation level through a slider in the GUI, the value is instantly reflected in the TestScene class, which then updates the tessellation shaders accordingly. This real-time feedback loop was vital during development, allowing me to tweak settings and immediately see the results in the rendered scene. This kind of interaction made the development process not only more efficient but also more engaging, as it provided a hands-on way to explore the effects of different parameters on the scene.

Report



1 Gui class Header



2 Gui Class source P1

Section 2: Model Loading and Integration with Renderer and RenderState Classes

Model Loading Process

The ModelLoader class handles the loading process using Assimp, which supports multiple 3D file formats like OBJ and FBX. The loader extracts and processes key data:

- **Vertex Data**: Defines the geometry of the model.
- **Normals**: Essential for accurate lighting.
- **Texture Coordinates (UVs)**: Map textures onto the model.
- **Tangents and Bitangents**: Used in normal mapping for detailed surface textures.
- **Indices**: Define how vertices connect to form the model's geometry.

This data is stored in buffers for GPU use during rendering.

Buffer Initialization

The next step is initializing OpenGL buffers:

- **Vertex Buffer Object (VBO)**: Stores vertex attributes.
- **Element Buffer Object (EBO)**: Stores indices for efficient geometry rendering.
- **Vertex Array Object (VAO)**: Encapsulates the configuration for rendering.

These buffers are prepared by the Renderer class using the createBuffers() method, ensuring they are ready for use in the rendering pipeline.

Integration with the Renderer Class

After loading, the Renderer class manages the rendering of models:

Shader Activation

The Renderer activates shaders as needed:

- **Vertex Shader**: Transforms vertices from model to world space.
- **Geometry Shader**: Optional; can modify or generate geometry.
- **Fragment Shader**: Calculates pixel colors using lighting and textures.

The Renderer ensures that these shaders are correctly set up and that necessary uniform variables are applied.

Texture Binding

The Renderer also manages textures, binding them before rendering:

- **Diffuse Map**: Provides base color.
- **Specular Map**: Controls surface shininess.
- **Normal Map**: Adds surface detail by modifying normals.

These textures are bound to specific units, ensuring the fragment shader can access them as needed.

Setting Uniforms and Drawing the Model

With shaders and textures set, the Renderer sets additional uniforms (e.g., transformation matrices) and issues a draw call using the VAO. The drawModel() method iterates over each mesh, rendering them with the active shaders and textures.

Role of the RenderState Class

The RenderState class manages the pipeline state, ensuring efficient rendering by minimizing redundant state changes. It tracks active shaders, bound textures, and other states, interacting with the Renderer to ensure that resources are used optimally.

State Management and Interaction

By tracking the current state, the RenderState class prevents unnecessary reactivations of shaders or textures, which improves performance. It also ensures that only necessary state changes are made, reducing rendering overhead.

Justification for the Renderer and RenderState Design

The separation into Renderer and RenderState was driven by:

- **Modularity**: Allows each class to focus on specific tasks, improving code clarity and ease of maintenance.
- **Maintainability**: Simplifies extensions and updates to the rendering system.
- **Efficiency**: Reduces redundant state changes, essential for real-time rendering performance.

This design creates a flexible, efficient, and maintainable rendering system capable of handling the demands of complex 3D scenes.

Section 3: Lighting Implementation: Skybox and Billboarding with Geometry Shader

Lighting is a fundamental aspect of any rendering project, and in this case, it involved implementing a Skybox and billboarding using a geometry shader. The Skybox was relatively straightforward — it required rendering a cube with a texture that represents the surrounding environment, creating the illusion that the scene is enclosed within a larger world.

The Skybox implementation used a simple vertex and fragment shader pair. The vertex shader (SkyboxVert.glsl) handled the positions of the vertices, applying a combined projection and view matrix to ensure the Skybox moved correctly with the camera. This approach ensured that the Skybox remained static relative to the camera's movements, effectively creating a seamless background.

The fragment shader (SkyboxFrag.glsl) was responsible for sampling the cubemap texture and applying it to the faces of the Skybox cube. This texture sampling is critical because it ensures the Skybox accurately reflects the surrounding environment, contributing to the realism of the scene.

The billboarding was a bit more challenging, particularly because it required the use of a geometry shader (BillboardGeo.glsl). The goal of the geometry shader in this context was to take a single point, representing the center of the billboard, and expand it into a quad that always faces the camera. This is achieved by calculating the 'up' and 'right' vectors, which are essential for keeping the billboard correctly oriented. One of the main challenges I encountered was dealing with the texture used for the billboard. Initially, the image had an unwanted white background, which detracted from the realism. While the billboard was functional, I chose to comment it out temporarily while I worked on other aspects of the terrain.

Section 4: Implementing the Phong Lighting Model

The Phong lighting model was crucial for realistic lighting in this project, balancing performance and visual quality. Implemented in fragment shaders like ModelFrag.glsl and FloorFrag.glsl, the model consists of ambient, diffuse, and specular components, providing a detailed, dynamic lighting effect across surfaces.

The ambient component ensures visibility even in shadowed areas, while the diffuse component scatters light for depth and form. The specular component simulates reflective highlights. A key challenge was accurately transforming normal vectors into world space, critical for correct lighting. Implementing the Phong model allowed for visually rich scenes without overwhelming the GPU, offering an effective balance between detail and performance.

Section 5: The Renderer Class and Its Role in the Rendering Pipeline

The Renderer class serves as the backbone of the rendering process in this project, orchestrating how different elements of the scene are drawn to the screen. When I designed this class, I wanted to ensure it could manage the rendering of various components—such as models, terrain, and the Skybox—while maintaining an efficient and organized structure.

The Renderer class includes several key methods, each dedicated to rendering a specific type of object. For instance, the method responsible for rendering the terrain first activates the tessellation shaders, binds the heightmap texture, and sets the necessary uniform variables, such as the tessellation levels and height scale. It then issues a draw call to the GPU to render the terrain mesh.

One of the decisions I made early on was to split the rendering logic between the Renderer class and the RenderState class. The RenderState class tracks the current state of the rendering context, including which shaders are active, what textures are bound, and other relevant state variables. This separation allowed me to avoid redundant state changes, which could have negatively impacted performance.

In terms of implementation, the Renderer class is invoked each frame to update the scene. It calls the relevant methods to render each component in the correct order—starting with the Skybox, then the terrain, and finally the models. This ensures that each element is drawn correctly, with the appropriate shaders and textures applied.

Section 6: Detailed Analysis and Justification of Shaders Codes

The shaders in this project are central to achieving the desired visual effects. In this section, I will discuss each shader individually, explaining the purpose of each line and justifying the techniques used.

Billboard Geometry Shader (BillboardGeo.glsl)

The Billboard Geometry Shader is crucial for creating billboards that always face the camera. Here's a breakdown:

1. **layout(points) in; layout(triangle_strip, max_vertices = 4) out;**
    - **This line specifies the input and output primitives. The shader takes in points and outputs a triangle strip, forming a quad.**
2. **vec3 pos = gl_in[0].gl_Position.xyz;**
    - **Extracts the position of the input point, which represents the center of the billboard.**
3. **vec3 up = vec3(0.0, 1.0, 0.0) * scale;**
    - **Defines the 'up' vector for the quad, scaled appropriately.**
4. **vec3 right = normalize(cross(cameraPos - pos, up)) * scale;**
    - **Calculates the 'right' vector, which, along with 'up', defines the orientation of the quad.**
5. **EmitVertex(); EndPrimitive();**
    - **These functions are responsible for outputting the vertices that form the quad.**

Model Vertex Shader (ModelVert.glsl)

The Model Vertex Shader is responsible for transforming model vertices from model space to world space.

1. **layout (location = 0) in vec3 position;**
    - **This specifies the input attribute, which is the vertex position.**
2. **uniform mat4 model, view, projection;**
    - **The model, view, and projection matrices are used to transform the vertex position from model space to screen space.**
3. **gl_Position = projection * view * vec4(position, 1.0);**
    - **This line performs the transformation, multiplying the vertex position by the model, view, and projection matrices.**

The shader efficiently handles the transformation of vertices, ensuring that the models are correctly positioned and oriented in the scene.

Model Fragment Shader (ModelFrag.glsl)

The Model Fragment Shader calculates the final color of each pixel on the model's surface.

1. **vec3 diffuseColor = texture(diffuseMap, gUV).rgb;**
    - **Samples the diffuse texture to obtain the base color of the model.**
2. **vec3 ambient = ambientIntensity * lightColor * diffuseColor;**
    - **Calculates the ambient lighting component.**
3. **vec3 specular = specularIntensity * spec * lightColor * specularColor;**
    - **Computes the specular lighting, which adds highlights to the model.**

The fragment shader is where the Phong lighting model is implemented, allowing for realistic rendering of the model's surface with ambient, diffuse, and specular components.

Floor Tessellation Control Shader (floorTCS.glsl)

The Floor Tessellation Control Shader is responsible for determining the tessellation levels for the terrain.

1. **float TessLevelOuter0 = GetTessLevel(camToVertex0, camToVertex1);**
    - **This line calculates the tessellation level for the outer edges of the patch based on the camera's distance.**
2. **gl_TessLevelOuter[0] = TessLevelOuter0;**
    - **Sets the tessellation level for one of the edges.**

By adjusting the tessellation levels based on the camera's distance, the shader ensures that detail is preserved where it is most needed, improving both performance and visual quality.

Floor Tessellation Evaluation Shader (floorTES.glsl)

The Floor Tessellation Evaluation Shader adjusts the vertices according to the tessellation levels and applies displacement based on the heightmap.

1. **float h = texture(heightMap, tesUV).r;**
    - **Samples the heightmap to determine the displacement.**
2. **tesPosInWS.y += h * heightScale;**
    - **Displaces the vertex along the y-axis based on the heightmap value.**

This shader plays a critical role in creating detailed and realistic terrain by displacing vertices according to the heightmap.

Floor Geometry Shader (floorGEO.glsl)

The Floor Geometry Shader computes the necessary vectors for lighting, such as the tangent and bitangent vectors.

1. **vec3 tangent = ...;**
    - **This line calculates the tangent vector, which is essential for normal mapping.**
2. **gTangent = normalize(tangent);**
    - **Normalizes the tangent vector.**

The geometry shader ensures that the terrain has the correct lighting information, which is crucial for achieving realistic shading effects.

Floor Fragment Shader (floorFrag.glsl)

The Floor Fragment Shader calculates the final color of each pixel on the terrain.

1. **vec3 finalColor = ambient + diffuse + specular;**
    - **Combines the ambient, diffuse, and specular components to determine the final color.**

This shader applies the Phong lighting model to the terrain, ensuring it is rendered with appropriate lighting effects.

Section 7: Level of Detail (LOD): Distance-Dependent Tessellation

In this project, tessellation was managed based on the camera's distance from the terrain to balance performance and visual quality. The tessellation control shader (floorTCS.glsl) adjusts the detail level for each terrain patch using a linear heuristic—closer areas receive higher detail, while distant regions are simplified.

This approach is refined in the tessellation evaluation shader (floorTES.glsl), where tessellated vertices are displaced using a heightmap to simulate complex terrain geometry. Normals are calculated using the Central Difference Method (CDM), ensuring accurate lighting even with displacement. The geometry shader (floorGEO.glsl) then prepares the data for final lighting calculations, optimizing both quality and performance.

Section 8: Terrain Generation from Heightmap

Terrain generation was achieved using a heightmap to create realistic landscapes. The vertex shader (floorVert.glsl) sets initial vertex attributes, while the tessellation control shader (floorTCS.glsl) adjusts detail based on camera distance.

The tessellation evaluation shader (floorTES.glsl) displaces vertices along the y-axis according to the heightmap, forming the terrain's shape. Normals are recalculated using CDM for accurate lighting.

The geometry shader (floorGEO.glsl) then processes these vertices for advanced shading, and the fragment shader (floorFrag.glsl) applies the final lighting model. This process allowed for detailed, realistic terrain without manual modeling.

Section 9: Implementing the Central Difference Method (CDM) vs. Normal Mapping

This project used both the Central Difference Method (CDM) and normal mapping for accurate terrain lighting. CDM calculates normals by comparing adjacent vertex heights, reflecting the true shape of the terrain.

Normal mapping adds surface details without increasing geometry complexity by using a texture to modify lighting calculations. Together, these techniques provided a realistic, detailed terrain surface, balancing visual appeal with computational efficiency.

Section 10: Challenges in Terrain Generation and Lighting Adjustments

Terrain generation and lighting adjustments posed several challenges, particularly in achieving realistic detail. Initial issues with flat shading were resolved by implementing distance-dependent tessellation, improving detail near the camera.

Switching to per-pixel lighting with the Blinn-Phong model significantly enhanced terrain appearance, capturing surface contours more accurately. Adjusting light direction and intensity further improved visual depth, ensuring the terrain appeared vibrant and lifelike.

Section 11: Component Justifications

Each component in this project was carefully chosen to meet specific goals, with a focus on balancing performance and visual quality. The geometry shader for billboarding ensures objects like trees always face the camera, a common need in real-time rendering.

Distance-dependent tessellation was implemented to optimize performance, reducing detail in distant objects while maintaining high quality up close. Heightmaps were used for terrain generation, providing complex landscapes without extensive manual modeling.

CDM was chosen for its accuracy in lighting calculations, ensuring realistic shading even with terrain displacement.

Section 12: Shader Pipeline Integration

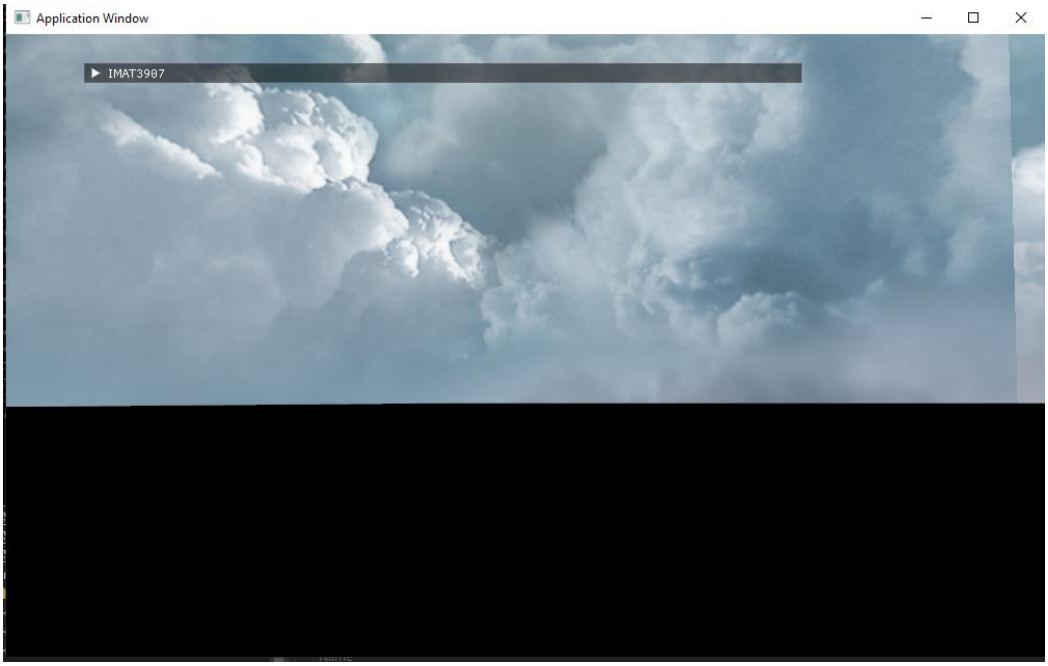
Shaders were tightly integrated into the rendering pipeline, each playing a critical role. Terrain rendering begins with vertex transformations, followed by dynamic tessellation and heightmap-based displacement. Geometry and fragment shaders then handle lighting and shading.
Model rendering uses a similar pipeline, transforming vertices and applying textures and lighting. The Skybox and billboards follow simpler paths but integrate seamlessly into the overall rendering process, ensuring a cohesive final image.

Section 13: Project Development Aspects

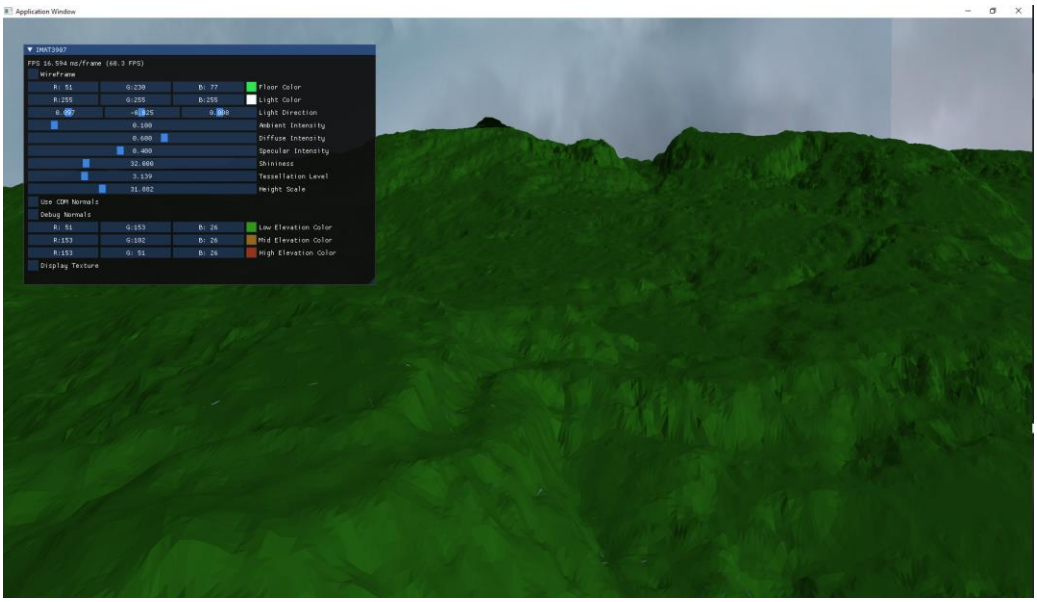
Throughout this project, I faced and overcame several challenges that significantly contributed to my development as a graphics programmer. Integrating the first-person camera required careful input management, while GUI integration with ImGui provided real-time parameter adjustments.
Tessellation and heightmap implementation were challenging but rewarding, particularly in balancing detail and performance. The billboarding issue, where textures had unwanted backgrounds, highlighted the importance of texture preparation in the shader pipeline. Each challenge offered valuable learning experiences, enhancing my understanding of real-time rendering.

3 first try of implementing Billboards

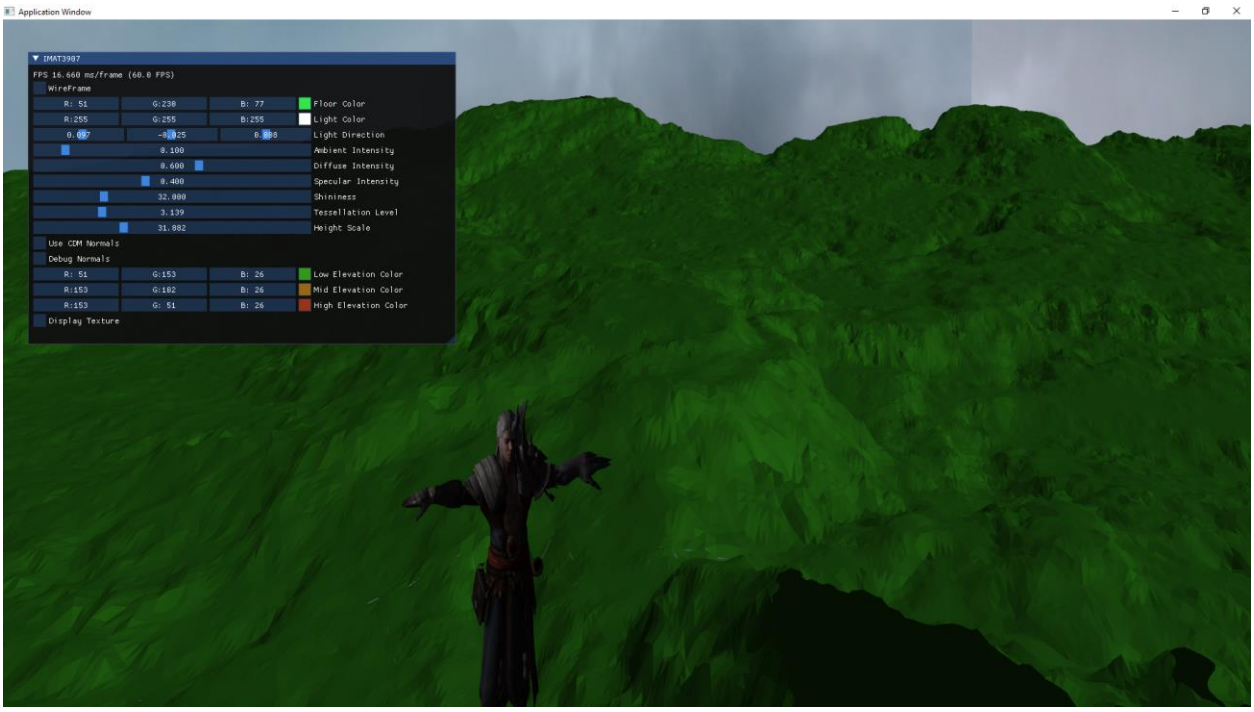
Report



4Struggling with making the terrain be based on the height Map



5 Current result without texture



6 Current result with model in the Scene