meshSpy: A Modern OpenGL GLTF Viewer

Deferred Rendering, PBR, and Image Based Lighting

Alazar Gebremehdin, Hannibal Mussie, Yassin Bedru, Samir Bahru

2026-01-04

Outline

Introduction

Project Overview

meshSpy is a real-time 3D model viewer capable of rendering high-fidelity GLTF scenes.

Key Technologies:

- **Core:** C++ with Qt 6 (Event System & Windowing).
- **Graphics:** Modern OpenGL (4.5 Core Profile).
- **Pipeline:** Deferred Rendering Architecture.
- **Shading:** Physically Based Rendering (PBR) & Image Based Lighting (IBL).

User Interaction:

- Arcball Camera control (Orbit, Pan, Zoom).
- Real-time toggling of material channels (Albedo, Metallic, Roughness).
- Wireframe visualization.

The GLTF File Format

What is GLTF?

GLTF (GL Transmission Format) is often called the "JPEG of 3D."

Why we chose it:

- **Efficiency:** Designed for fast runtime loading by the GPU.
- **Standardization:** It is the industry standard for PBR (Physically Based Rendering). It natively supports Metallic-Roughness workflows.
- **Structure:** A JSON hierarchy describes the scene graph, while heavy data (vertices, textures) is stored in binary buffers (`.bin`).

The GLTF Schema

The GLTF structure links high-level objects to low-level binary data:

1. **Scenes & Nodes:** The hierarchy of objects.
2. **Meshes & Primitives:** The geometry data.
3. **Materials:** PBR parameters and texture references.
4. **Accessors & BufferViews:** define how to read the raw binary data (e.g., "Read floats with stride 12 starting at byte 0").

GLTF Example

A minimal example of a textured triangle in GLTF JSON:

```json
{
  "asset": { "version": "2.0" },
  "nodes": [ { "mesh": 0 } ],
  "meshes": [ {
    "primitives": [ {
      "attributes": { "POSITION": 0, "TEXCOORD_0": 1 },
      "indices": 2,
      "material": 0
    }]
  }],
  "accessors": [
    { "bufferView": 0, "componentType": 5126, "count": 3,
    "type": "VEC3" }
  ],
```

GLTF Example


```
  "bufferViews": [ { "buffer": 0, "byteLength": 36,
  "byteOffset": 0 } ]
}
```

OpenGL Objects & Buffers

Geometry Objects (VAO, VBO, EBO)

Modern OpenGL requires explicit memory management on the GPU.

- **VBO (Vertex Buffer Object):** A raw buffer of memory on the GPU storing vertex data (Positions, Normals, UVs).
- **EBO (Element Buffer Object):** Stores indices allowing us to reuse vertices (e.g., sharing a vertex between two triangles).
- **VAO (Vertex Array Object):** The **State Container**. It doesn't store data itself; it remembers **how** to read the VBOs (e.g., "Attribute 0 starts at byte 0 and uses 3 floats").

In meshSpy: We generate a VAO for every SubMesh to quickly bind its state before drawing.

Rendering Objects (FBO, Textures)

- **Textures:** Image data stored on the GPU. In our project, these store not just colors, but PBR data (Roughness/Metalness) and Normal maps.
- **FBO (Framebuffer Object):** A custom container for rendering. By default, OpenGL draws to the screen. An FBO allows us to draw into **Textures**.

Crucial for meshSpy: The **G-Buffer** is simply an FBO with 4 distinct Textures attached to it. When we render, the FBO splits the output into these 4 textures simultaneously.

Data Objects (UBO)

- **UBO (Uniform Buffer Object):** A buffer for storing global variables (Uniforms) that remain constant across many shaders, such as View and Projection matrices.

Usage: While we used standard `glUniform` calls for simplicity in this project, UBOs are the "Modern" way to share Camera data between the Geometry Pass and the Skybox Pass without re-uploading the data for every shader program.

Modern vs. Fixed Function OpenGL

meshSpy utilizes the **Programmable Pipeline**.

Fixed Function (Legacy):

- Easy to start, but inflexible.
- Light calculations are done per-vertex (Gouraud shading).
- "Black box" state machine.

Programmable Pipeline (Modern):

- **Pros:** Complete control over every pixel via Shaders.
- **Pros:** Massive performance gains via VBOs (Vertex Buffer Objects) and VAOs.
- **Implementation:** We manually define how memory is read from the GPU.

Implementation Details

In `DeferredRenderer.cpp`, we use `glVertexAttribPointer` to tell the GPU exactly how to interpret our buffer data.

```cpp
// Telling OpenGL: "The first 3 floats are Position, the next 2 are UVs"
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 5 * sizeof(float), (void*)0);

glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 5 * sizeof(float), (void*)(3 * sizeof(float)));
```

This allows us to pack arbitrary data (Tangents, PBR weights) that legacy OpenGL wouldn't understand.

The Shader Pipeline

Vertex & Fragment Shaders

1. Vertex Shader (`geometry.vert`):

- Handles **Transformation**. Converts 3D Local Space coordinates to 2D Clip Space.
- Matrix Math: $P \times V \times M \times Vertex$.

2. Fragment Shader (`geometry.frag`):

- Handles **Shading**. Calculates the color of individual pixels.
- In our Deferred pipeline, this doesn't calculate light; it just outputs raw material data to textures.

Analysis: geometry.frag

This shader populates our G-Buffer (Geometric Buffer). Instead of outputting a color to the screen, it outputs data to 4 distinct textures at once.

```glsl
// geometry.frag
layout (location = 0) out vec4 gPosition; // World Position
layout (location = 1) out vec4 gNormal;   // Surface Normal
layout (location = 2) out vec4 gAlbedo;   // Base Color
layout (location = 3) out vec4 gPBR;      // Metal/Rough/AO

void main() {
    gPosition = vec4(FragPos, gl_FragCoord.z);
    // Calculating Normal Mapping (TBN Matrix)
    gNormal.rgb = getNormalFromMap();
    // Packing PBR data into channels
    gPBR.r = uMetallicFactor;
```

Analysis: geometry.frag

```
    gPBR.g = uRoughnessFactor;
}
```

Rendering Systems

Forward vs. Deferred Rendering

Forward Rendering (Traditional):

- Geometry is drawn and lit in a single pass.
- Complexity: $O(\mathcal{Geometry} \times \mathcal{Lights})$.
- **Cons:** Expensive if there are many lights or heavy overdraw.

Deferred Rendering (meshSpy):

- Decouples geometry from lighting.
- Complexity: $O(\mathcal{Geometry} + \mathcal{Screen\ Resolution} \times \mathcal{Lights})$.
- **Pros:** Lighting calculation only happens for visible pixels.

The G-Buffer Approach

We implemented a **Multi-Render Target (MRT)** G-Buffer in `GBuffer.cpp`.

Pass 1: Geometry Pass

- Binds the G-Buffer FBO.
- Renders the scene.
- Stores Position, Normal, Albedo, and PBR parameters into floating-point textures (`GL_RGBA16F`).

Pass 2: Lighting Pass

- Binds the default framebuffer.
- Draws a single full-screen Quad.
- Samples the G-Buffer textures pixel-by-pixel to calculate lighting.

Physically Based Rendering (PBR)

The Principles

PBR attempts to simulate the physical interaction of light and matter.

1. **Microfacet Theory:** Surfaces are composed of microscopic mirrors. Rougher surfaces scatter light randomly (blurrier reflections).
2. **Energy Conservation:** Reflected light cannot exceed incoming light ($k_D + k_S \leqslant 1.0$).
3. **Metallic Workflow:**
   - **Dielectrics (Plastic/Wood):** White specular highlight, colored diffuse.
   - **Metals:** Colored specular reflection, no diffuse.

Shader Implementation

In `lighting.frag`, we implement the **Cook-Torrance BRDF**.

$$f_r = \frac{DFG}{4(\omega_o \cdot n)(\omega_i \cdot n)}$$

- **D (Distribution):** Trowbridge-Reitz GGX. How aligned are microfacets to the half-vector?
- **F (Fresnel):** Fresnel-Schlick. Reflections are stronger at grazing angles.
- **G (Geometry):** Self-shadowing of microfacets.

glsl

```glsl
// Fresnel Calculation in lighting.frag
vec3 fresnelSchlickRoughness(float cosTheta, vec3 F0, float roughness)
{
    return F0 + (max(vec3(1.0 - roughness), F0) - F0)
               * pow(clamp(1.0 - cosTheta, 0.0, 1.0), 5.0);
```

Shader Implementation


```
}
```

Image Based Lighting (IBL)

Lighting with the Sky

Instead of manual point lights, we use an HDR Environment Map to light the scene. This provides realistic reflections.

Implementation Strategy (Split-Sum Approximation):

1. **Diffuse Irradiance:** We sample a highly blurred version of the environment map (high mipmap level) to simulate ambient light scattering.
2. **Specular Reflection:** We sample the environment map based on the surface roughness.

glsl

```glsl
// lighting.frag
// Rougher surfaces sample from higher (blurrier) mip levels
vec3 prefilteredColor = textureLod(environmentMap, uvSpec,
                                   Roughness *
                                   MAX_REFLECTION_LOD).rgb;
```

Limitations & Future Work

Current Status & Next Steps

Current Limitations:

- **Transparency:** Deferred rendering struggles with transparent objects (glass) because G-Buffers only store the closest surface.
- **Shadows:** No shadow mapping implemented (objects don't cast shadows on themselves).

Future Work:

- **SSAO (Screen Space Ambient Occlusion):** To add depth to crevices.
- **Forward Pass:** Implementing a second pass specifically for transparent objects.
- **Optimize:** Compute Shader culling for complex scenes.

Current Status & Next Steps

Thank You

Q & A