WM9M2 Computer Graphics

Harsh Dubey
University of Warwick

December 16, 2025

Abstract

This project implements a real time Computer Graphics Playground featuring multiple lighting models (no limit on numbers) including directional, spot, and point lights. The editor also incorporates several post-processing effects (post-pass) such as vignette, blur, and bloom with various texture and mapping techniques including Mip-Map.

1 Introduction

Figure 1: Rendered output of the Computer Graphics Playground

Initially, the project was to create a game, but instead I ended up writing a lot of shaders and working with DirectX 12, resulting in a Computer Graphics Playground (CGP). The CGP is about testing shader algorithms related to lighting and textures such as normal, metallic, and many more. The editor uses Assimp to parse 3D models, making the CGP capable of loading a variety of models. The editor uses ImGui for UI-related editing responses and saves the map with "Ctrl + S" via a snapshot. In order to make lighting counts unrestricted, I have implemented a structured array of data, and to make the rendered output look good when viewed from a distance, I implemented GPU-side mip-map generation shader along with post processing effects via a post-pass.

2 Method

2.1 Loading Meshes

In CGP, I implemented an Assimp importer that parses 3D models using Assimp. A Mesh geometry that uses the parsed data and structures it for each submesh, and also defines the vertex layout. The GPU mesh converts the submeshes into GPU resources using a staging buffer, which is a wrapper around `ID3D12Resource` to make data uploading easier. A model pool is used to cache already loaded

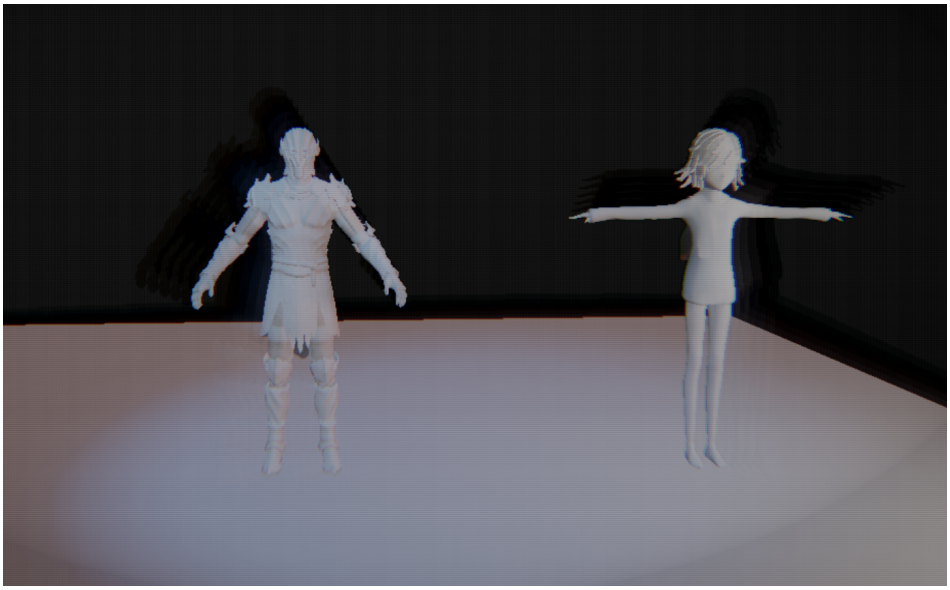


Figure 2: different format runtime time mesh loading

GPU meshes in order to avoid duplication and reloading. Finally, the model class compiles all the submeshes into a tree hierarchy and stores the transformations in correct order.

2.2 Texture

For loading images, I incorporated the stb_image package and created an image pool to cache already loaded images and prepare GPU resources. Since every image has an SRV handle associated with it, and every submesh in a model can have its own materials and properties, adding individual SRV handles directly to the root signature would be unreliable. To address this, I used a range-based texture slot system. This required a contiguous chunk of heap memory so that descriptor heap handles could be copied efficiently. As a result, I implemented a resource heap that allocates contiguous blocks of handles and manages heap memory through bulk allocation and deallocation, avoiding fragmented regions altogether (see Figure 3).

2.3 Texture Mapping

CGP implements several widely used real-time rendering techniques to produce improved visual quality, as demonstrated in the following sections.

2.3.1 Normal Mapping and Detail Normal Mapping

2.3.2 Normal Mapping and Detail Normal Mapping

Normal mapping defines the normal surface per fragment using a normal tangent-space map. A secondary detail normal map is optionally applied to introduce higher-frequency surface detail, as shown in Fig. 4.

Key points:

- Tangent-space normals transformed using the TBN basis.
- Strength-controlled blending with the geometric normal.
- Detail normal adds fine surface variation.

```
1 float3 nTS = sample * 2.0f - 1.0f;
2 float3 nWS = normalize(nTS.x * T + nTS.y * B + nTS.z * N);
3 nWS = normalize(lerp(N, nWS, Normal_Meta.y));
```

Figure 3: Multiple Textures on each submeshes

Listing 1: Key normal mapping operations

2.3.3 Height Mapping (Parallax UV Offset)

Height mapping is implemented as a parallax UV offset technique. Texture coordinates are displaced based on a height texture and the view direction, creating the illusion of depth without modifying geometry, as demonstrated in Fig. 5.

Key points:

- View-dependent UV displacement.
- Height value centered around zero.
- Fully pixel-shader based.

Figure 4: Applied normal mapping on the cloth and body

```
float height = (h - 0.5f) * Height_Meta.y;
float2 uvOut = uv0 + V.xy * height;
```

Listing 2: Key height parallax logic

2.3.4 Displacement Mapping (Screen-Space)

Displacement mapping uses a separate displacement texture to offset texture coordinates in screen space. This technique enhances perceived surface depth while remaining computationally inexpensive, as shown in Fig. 6.

Key points:

- Uses displacement texture instead of geometry tessellation.
- View direction controls displacement magnitude.
- Can be enabled independently from height mapping.

```
float height = (d - 0.5f) * Displace_Meta.y;
float2 uvOut = uv0 + V.xy * height;
```

Listing 3: Key displacement mapping logic

2.3.5 ORM Material Encoding (Occlusion, Roughness, Metallic)

Material properties are packed into an ORM texture to reduce texture fetches and memory bandwidth. Compared to Fig. 3, Fig. 7 demonstrates improved material realism, where metallic regions exhibit reduced diffuse response and sharper specular highlights.

Key points:

- Ambient occlusion, roughness, and metallic stored in RGB.
- Individual channels can be overridden.
- Supports mixed asset workflows.

```
float3 orm = SampleTex3(gORMTex, uv);
float rough = lerp(roughSing, orm.g, hasOrm);
float metal = lerp(metalSing, orm.b, hasOrm);
```

Listing 4: Key ORM mixing logic

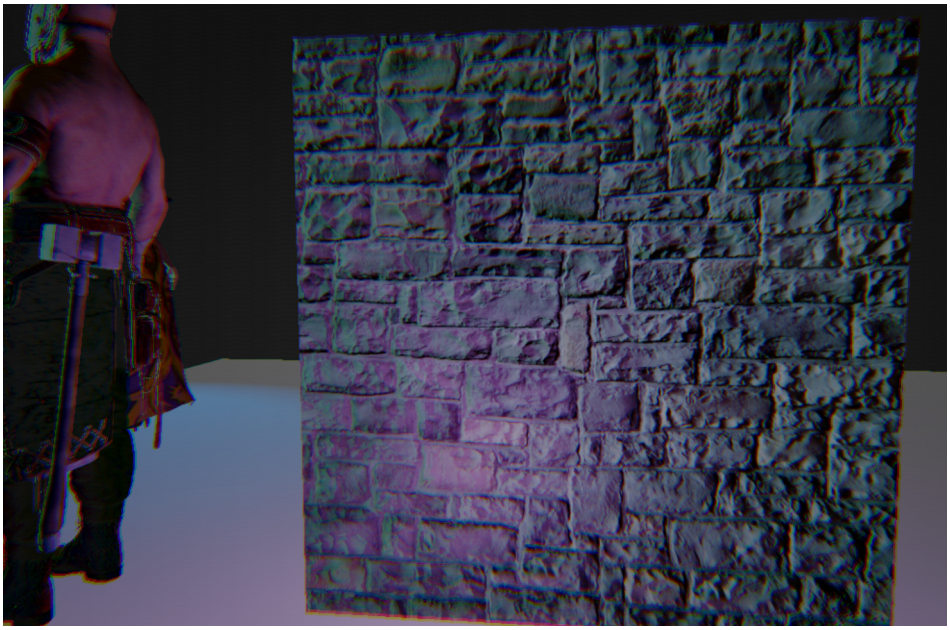


Figure 5: Applied parallax height mapping on the cube

2.3.6 Ambient Occlusion

Ambient occlusion modulates indirect lighting by darkening crevices and contact areas. In CGP, ambient occlusion can be sourced either from the ORM texture or from a separate occlusion map, depending on asset configuration.

Key points:

- Reduces indirect lighting contribution.
- Can be sourced from ORM or separate texture.

```
1 float ao = lerp(1.0f, aoTex, strength);
```

Listing 5: Ambient occlusion application

2.3.7 Specular and Glossiness Maps

Specular and glossiness textures support a traditional Blinn–Phong workflow. Glossiness controls highlight sharpness, while specular color defines reflective intensity.

Key points:

- Glossiness controls shininess exponent.
- Specular color blended using strength parameter.

```
1 float shininess = lerp(8.0f, 256.0f, gloss01);
2 float spec = pow(saturate(dot(N, H)), shininess);
```

Listing 6: Specular and gloss usage

2.3.8 Emissive Mapping

Emissive textures allow materials to emit light independently of scene lighting, useful for screens, LEDs, and stylized effects.

Key points:

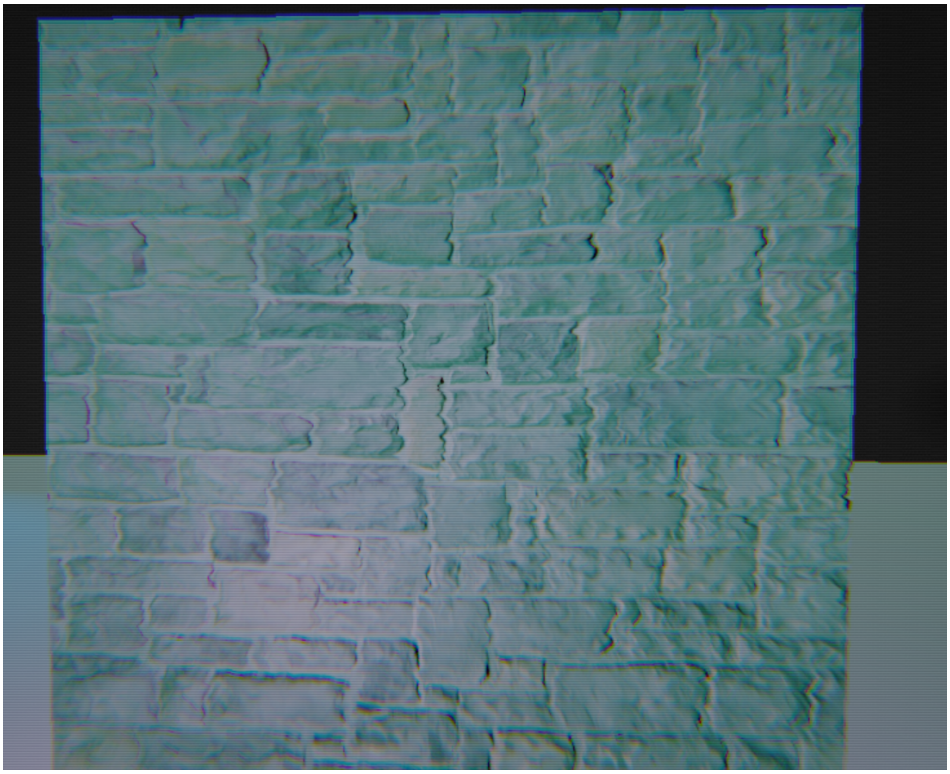


Figure 6: Applied displacement mapping on the cube

- Adds unlit radiance to the final color.
- Controlled via intensity parameter.

```
float3 emissive = SampleTex3(gEmissiveTex, uv) * Emissive_Meta.y;
```

Listing 7: Emissive contribution

2.3.9 Opacity (Alpha Cutout)

Alpha cutout mapping discards fragments based on an opacity threshold, enabling efficient rendering of partially transparent materials without depth sorting, as illustrated in Fig. 8.

Key points:

- Binary discard using clip().
- Suitable for foliage and thin geometry.

```
clip(alpha - Opacity_Meta.z);
```

Listing 8: Alpha cutout logic

2.4 Mip Map Generation

Mipmaps are a prefiltered hierarchy of textures stored at progressively lower resolutions. They are used to reduce aliasing and shimmering when textured surfaces appear smaller on screen, while also improving texture cache efficiency. In CGP, mipmaps are generated at runtime on the GPU using a compute-shader-based downsampling approach, rather than being precomputed offline.

Implementation overview. For each mip level $m > 0$, the previous level $(m - 1)$ is bound as a shader resource view (SRV), while the destination level m is bound as an unordered access view

Figure 7: Applied ORM on the witch model

(UAV). A compute dispatch then downsamples the source mip into the destination mip. Resource state transitions are performed per subresource to switch between the shader resource and the unordered access usage.

Compute shader downsampling. The compute shader implements a simple 2×2 box filter, where each output texel is the average of four neighbouring texels from the previous mip level. This approach is efficient, easy to implement, and sufficient for CGP.

The overall workflow follows established Direct3D 12 compute-based mipmap generation techniques described in online graphics programming resources and official Microsoft documentation, including the DirectXTex mipmap generation guidelines and the compute-based texture processing examples presented by 3D Game Engine Programming [vO19].

2.5 Lighting

CGP supports three types of lights: directional, point, and spot lights. All lights are implemented through a common light interface that exposes shared parameters such as light type, colour, intensity, position, direction, and attenuation. These parameters are collected and uploaded to the GPU each frame(rebuild on dirty only) in a packed structured buffer format, allowing shaders to evaluate multiple dynamic lights efficiently.

Each renderable object maintains an associated light manager. The render queue is responsible for attaching lights to an object when new lights are added to the scene, and removing them when lights are deleted or no longer relevant. Before uploading data to the GPU, the light manager filters and packs only the required lights into a contiguous buffer and performs the necessary resource state transitions(copy dst)so the data can be accessed safely by shaders.

Distance-based culling. To reduce unnecessary lighting cost, the light interface provides a culling mechanism based on distance. If the influence radius of a light does not reach a given object, the light manager excludes it during the packing stage. This ensures that only lights capable of affecting an object are evaluated in the shader. Directional lights are treated as a special case: since they represent infinitely distant light sources, they are not distance-cullable and are always included. The system supports multiple directional lights simultaneously.

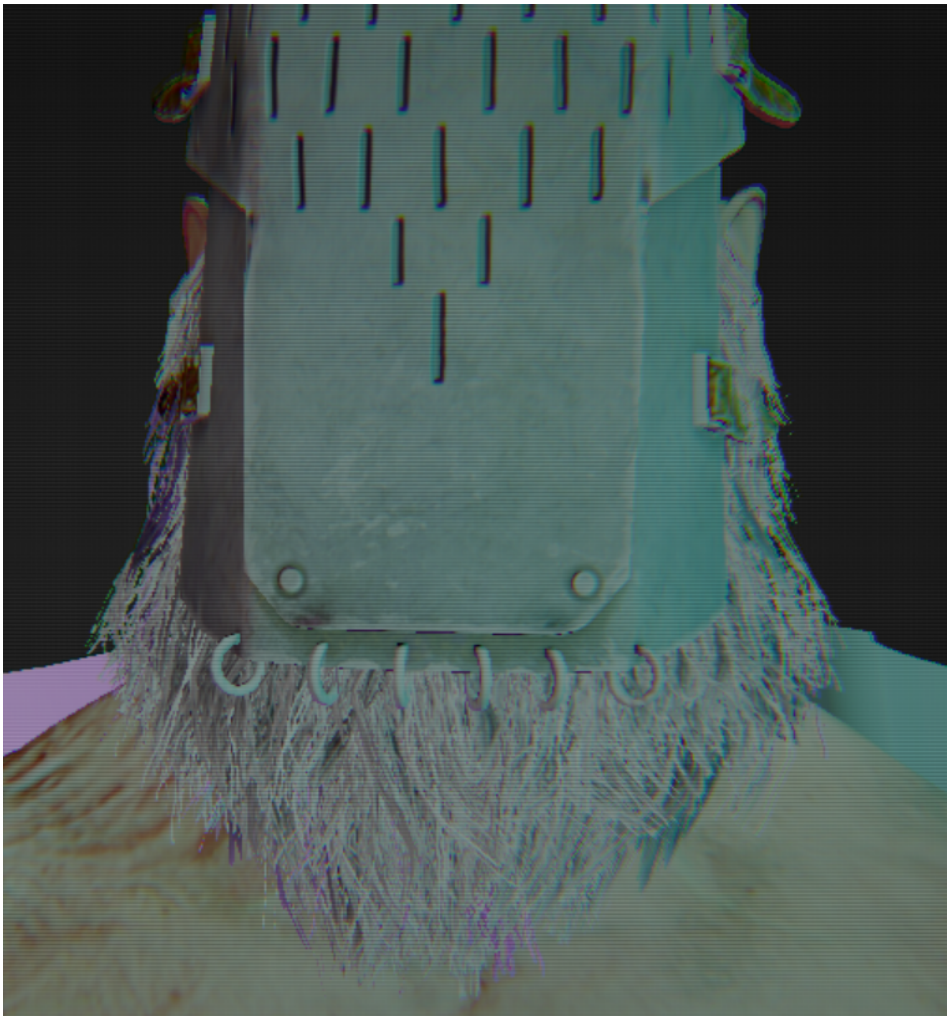


Figure 8: Alpha cutout applied to thin geometry

2.5.1 Directional Light

Directional light models illumination from an infinitely distant source, such as sunlight. Because the light source is assumed to be infinitely far away, directional light has no position or attenuation and applies a constant light direction across the entire scene. In CGP, directional lights are evaluated using Lambertian diffuse reflection, where surface brightness depends on the cosine of the angle between the surface normal and the incoming light direction [PJH18a, Wik25b].

Shader behaviour (key points):

- The surface normal is normalized in world space before lighting evaluation.
- A small constant ambient term is added to avoid fully black regions in unlit areas.
- The shader iterates over the packed light buffer and selects only directional lights.
- Light direction is taken from a pre-normalized value when available, with a fallback normalization for robustness.
- Lambert diffuse shading is computed using $\max(0, \mathbf{N} \cdot \mathbf{L})$ and scaled by light colour and intensity.
- A simple tone-mapping clamp is applied to prevent excessive brightness under high-intensity lighting, as visible in Fig. 10c.

```
float3 result = baseColor * 0.20f; // small ambient term

for (uint i = 0; i < min(gNumTotalLights, 256u); ++i)
{
    // Skip non-directional lights
    if (!IsDirectionalLight(i))
        continue;

    float3 Ldir = normalize(GetLightDirection(i));
    float  NdotL = saturate(dot(normalize(worldN), Ldir));

    result += baseColor * GetLightColor(i) * GetLightIntensity(i) * NdotL;
}
```

Listing 9: Directional Lambert lighting (core logic)

2.5.2 Point Light

Point lights model localized emitters such as bulbs or small lamps. Unlike directional lights, a point light has a world-space position and its contribution depends on the distance to the shaded point. In CGP, point lights are evaluated using Lambertian diffuse shading and an optional Blinn–Phong specular term. Distance attenuation is applied so that the light smoothly fades out with range and avoids unrealistically affecting the entire scene.

Key points (shader behaviour):

- The shader computes a vector from the shaded point to the light position and derives a normalized light direction.
- A distance term is computed and used to attenuate intensity, ensuring the light decays with distance.
- A smooth range fade is applied to avoid harsh cutoffs at the light radius.
- Lambert diffuse uses $\max(0, \mathbf{N} \cdot \mathbf{L})$ for the cosine-weighted response.
- The Blinn–Phong option computes a half-vector $\mathbf{H} = \text{normalize}(\mathbf{L} + \mathbf{V})$ and raises $\max(0, \mathbf{N} \cdot \mathbf{H})$ to a shininess exponent controlled by glossiness (Fig. 11c).
- Multiple point lights are accumulated additively, as shown in Fig. 11b.

Attenuation model. CGP uses a practical real-time attenuation function combining a smooth range fade with a quadratic distance falloff. The range fade ensures the light contribution smoothly approaches zero near the configured range, while the quadratic term models decreasing intensity with distance in a stable way.

```
float x = saturate(1.0f - (dist / range));
float rangeFade = x * x;
float invSq = 1.0f / (1.0f + attenuation * dist * dist);
float att = rangeFade * invSq;
```

Listing 10: Point light attenuation (core logic)

Diffuse and specular evaluation (main logic). Diffuse lighting accumulates the Lambert term scaled by attenuation. When specular is enabled, Blinn–Phong highlights are added using the half-vector formulation, with shininess derived from a glossiness parameter.

```
float3 toLight = lightPos - worldPos;
float  dist = length(toLight);
float3 Ldir = toLight / dist;

float NdotL = saturate(dot(N, Ldir));
float att   = ComputePointAttenuation(dist, range, attenuation);
```

```

diffuseAcc += baseColor * radiance * (NdotL * att);

float3 V = normalize(cameraPos - worldPos);
float3 H = normalize(Ldir + V);
float  spec = pow(saturate(dot(N, H)), shininess);

specAcc += radiance * (spec * gloss01 * att);
```

Listing 11: Point light diffuse + Blinn–Phong specular (main logic)

The underlying diffuse and specular models are based on standard Lambertian reflection and Blinn–Phong shading as commonly used in real-time rendering [Wik25b, Wik25a, PJH18b].

2.5.3 Spot Light

Spot lights extend the point light model by introducing angular attenuation through a cone constraint. In CGP, spot lights reuse the same diffuse (Lambert) and specular (Blinn–Phong) evaluation as point lights, while adding an additional cone factor that restricts illumination to a specified angular region.

Cone Attenuation (Inner/Outer Angle + Softness). The spotlight cone is defined using inner and outer cone angles represented as cosine values. Fragments inside the inner cone receive full intensity, while fragments between the inner and outer cones are smoothly attenuated. Fragments outside the outer cone receive no contribution.

A softness parameter further shapes the falloff near the cone boundary, allowing both sharp theatrical spotlights and softer, more realistic lighting.

Key differences from point lights:

- Illumination is restricted to a cone aligned with the light direction.
- A smooth transition is applied between inner and outer cone angles.
- A power curve controls edge softness at the cone boundary.

```
float cd = dot(normalize(spotDirN), normalize(-Ldir));
float t  = saturate((cd - outerCos) / max(innerCos - outerCos, 1e-5f));
float cone = pow(t, lerp(1.0f, 0.35f, softness));
```

Listing 12: Spot cone attenuation (core logic)

Spot Light Shading. After computing the cone factor, spot light shading proceeds identically to point lights. The final radiance is scaled by distance attenuation and cone attenuation before contributing to the diffuse and specular terms.

```
float att = ComputePointAttenuation(dist, range, attenuation);
float3 radiance = color * intensity * att * cone;
```

Listing 13: Spot light contribution (integration step)

Spot Light Test Scenes. To visually validate the spotlight implementation and demonstrate artistic controls, the following configurations are included:

- **multiple-spot-light:** Demonstrates accumulation of multiple spot lights with overlapping cones.
- **spot-light:** A single spotlight showing the baseline cone-limited illumination.
- **spot-soft-light:** Increased softness to produce smoother cone boundaries.
- **spot-light-inner-angle:** A narrower inner cone relative to the outer cone, resulting in an extended falloff region and enhanced perceived softness.

2.6 Post Pass (Render-To-Texture)

CGP supports post-processing through a render-to-texture (RTT) pipeline, where the main scene is rendered into an intermediate GPU texture instead of directly to the swap chain. This intermediate result is then sampled in a subsequent full-screen post pass to apply screen-space effects.

The system is built around a custom render target abstraction that wraps a `ID3D12Resource` together with its associated render target view (RTV), shader resource view (SRV), and explicit resource state tracking. This allows the same texture to be safely written during the main pass and read during post processing.

Key design points:

- A single wrapper manages the texture resource, RTV, and SRV.
- Descriptor allocation is handled via dedicated RTV and shader-visible resource heaps.
- Resource states are tracked and transitioned explicitly to satisfy DirectX 12 requirements.

Post-pass workflow:

- Render the scene into the render target texture.
- Transition the texture to a shader resource state.
- Sample the texture in a full-screen post-processing pass.

2.6.1 Post Processing Effects

The post pass supports a set of screen-space effects applied in a full-screen shader. Each effect can be enabled independently and is evaluated using the rendered scene texture as input. For clarity, each effect below includes a small representative code equation and the corresponding visual output.

Exposure (code: $C \leftarrow C \cdot \text{Exposure}$)

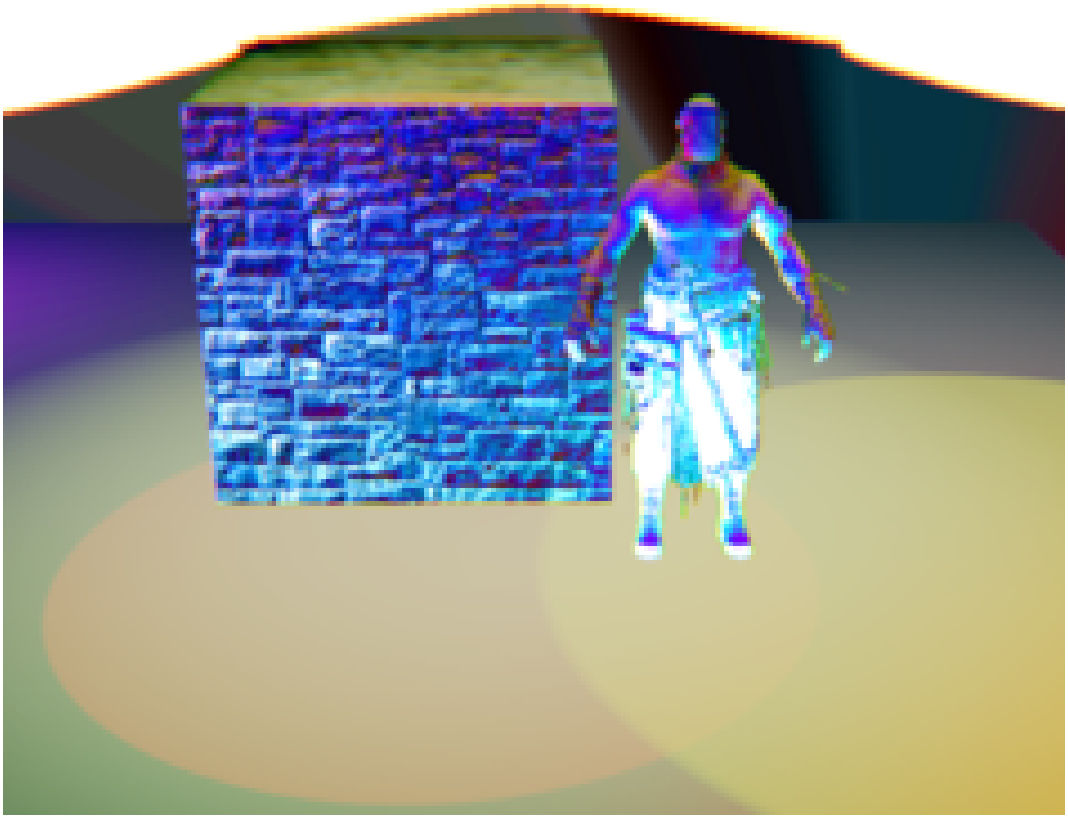
```
color.rgb *= Exposure;
```

Gamma correction (code: $C \leftarrow C^{1/\gamma}$)

```
color.rgb = pow(saturate(color.rgb), 1.0f / Gamma);
```

Contrast (code: $C \leftarrow (C - 0.5) \cdot k + 0.5$)

```
color.rgb = (color.rgb - 0.5f) * Contrast + 0.5f;
```

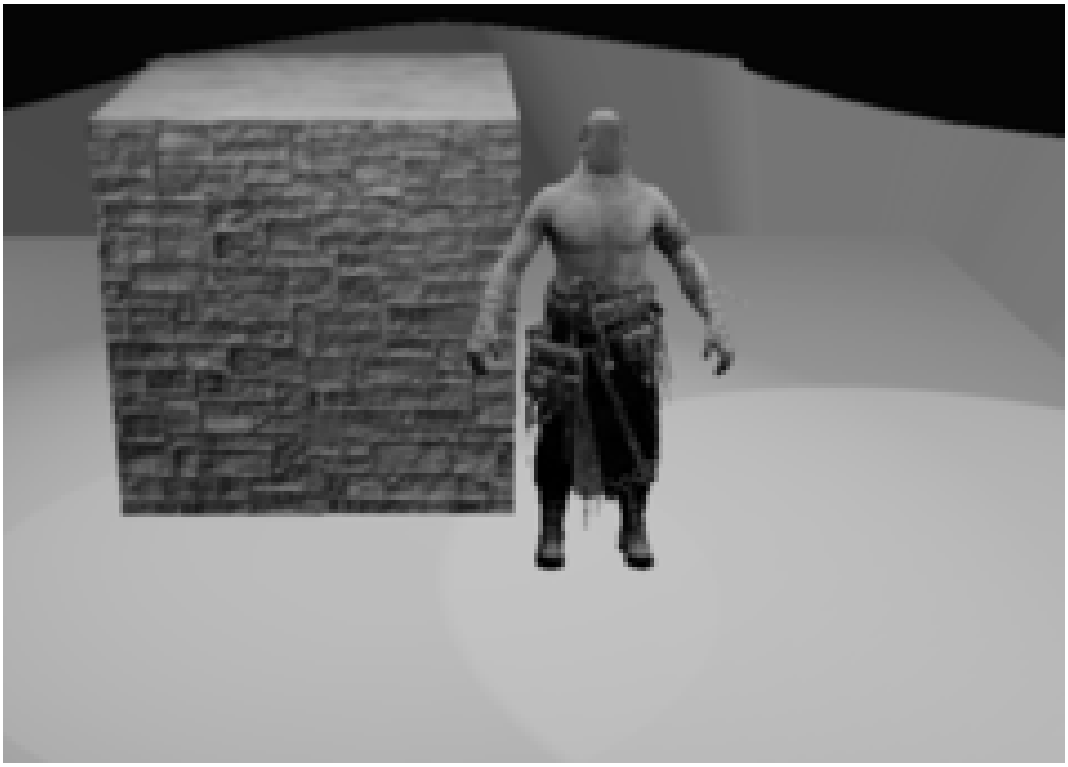


Saturation (code: $C \leftarrow \text{lerp}(L, C, s)$)

```
float luma = dot(color.rgb, float3(0.2126f, 0.7152f, 0.0722f));
color.rgb = lerp(float3(luma, luma, luma), color.rgb, Saturation);
```

Grayscale (code: $C \leftarrow \text{lerp}(C, L, g)$)

```
float luma = dot(color.rgb, float3(0.2126f, 0.7152f, 0.0722f));
color.rgb = lerp(color.rgb, float3(luma, luma, luma), Grayscale);
```



Invert (code: $C \leftarrow \text{lerp}(C, 1 - C, i)$)

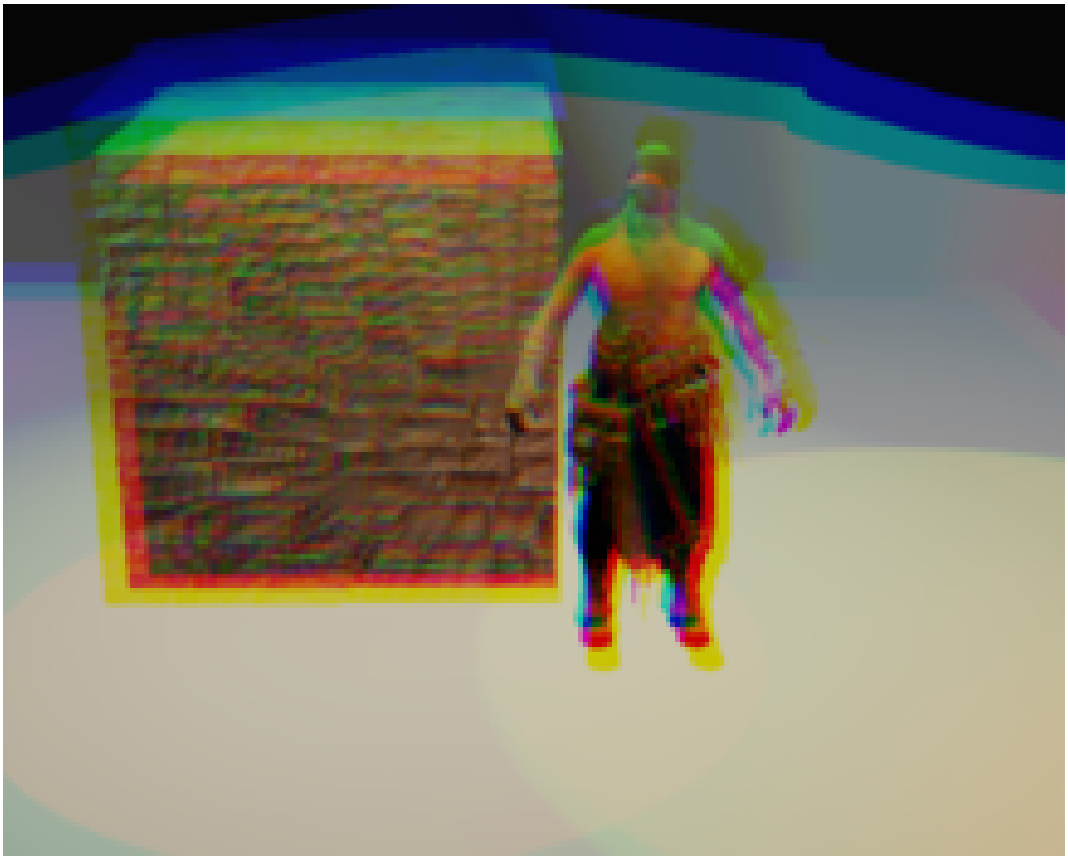
```
color.rgb = lerp(color.rgb, 1.0f - color.rgb, Invert);
```

Vignette (code: $C \leftarrow C \cdot (1 - v \cdot r^p)$)

```
float2 d = uv * 2.0f - 1.0f;
float r = length(d);
float vig = 1.0f - Vignette * pow(saturate(r), VignettePower);
color.rgb *= vig;
```

Chromatic Aberration (code: $C \leftarrow (R(uv + \Delta), G(uv), B(uv - \Delta))$)

```
float2 d = (uv - 0.5f);
float2 o = normalize(d) * AberrationAmount;
float r = S(uv + o).r;
float g = S(uv).g;
float b = S(uv - o).b;
color.rgb = float3(r, g, b);
```



Scanlines (code: $C \leftarrow C \cdot (1 - a \cdot \sin(\omega y + t))$)

```
float s = 0.5f + 0.5f * sin((uv.y * ScanFreq + Time * ScanSpeed) * 6.28318f);
color.rgb *= (1.0f - ScanAmount * s);
```

Film Grain (code: $C \leftarrow C + a \cdot \text{noise}(uv, t)$)

```
float n = Hash(uv * GrainScale + Time);
color.rgb += (n - 0.5f) * GrainAmount;
```

Dithering (code: $C \leftarrow C + a \cdot (bayer(x, y) - 0.5)$)

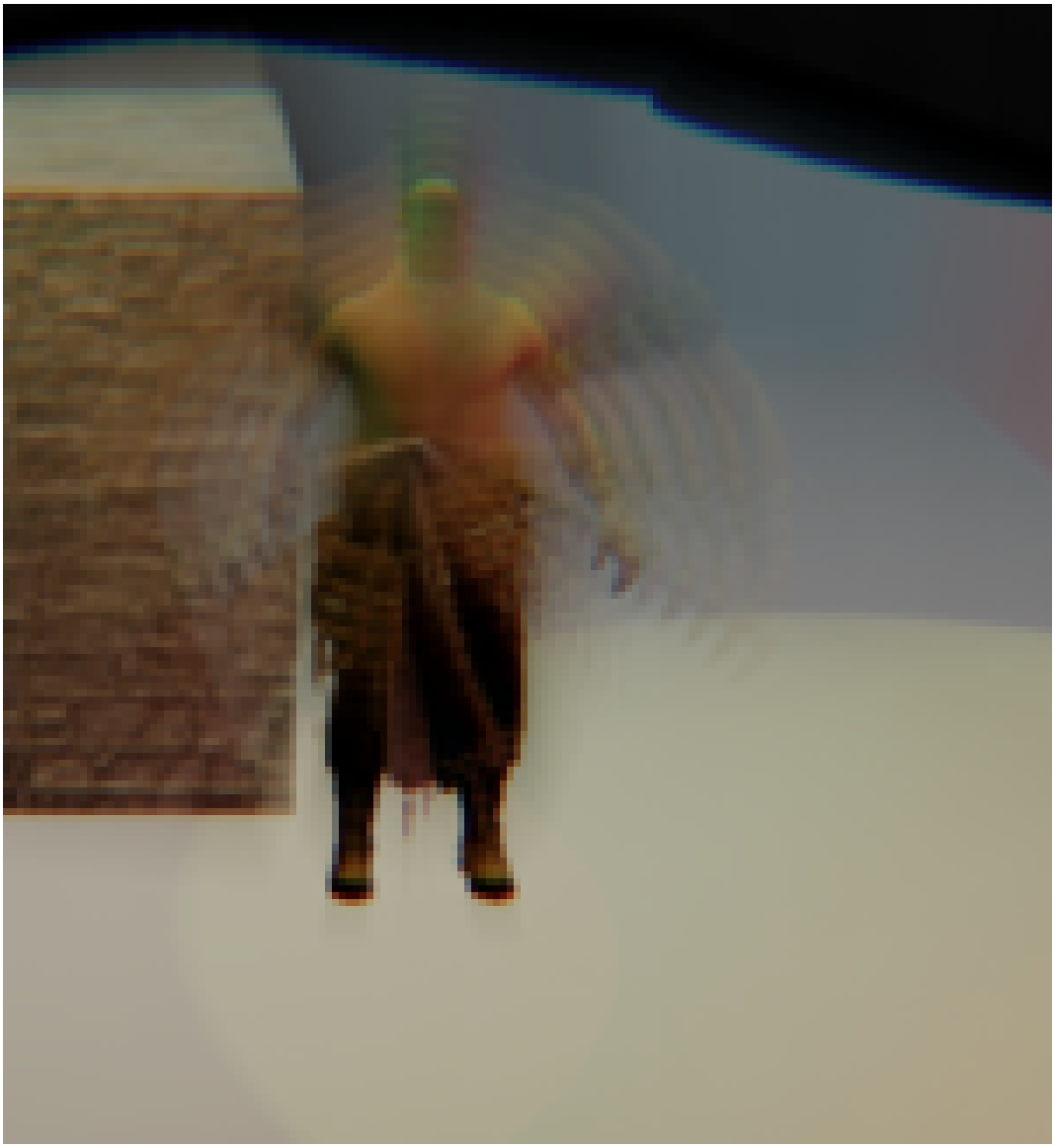
```
float d = Bayer4x4(pixelXY) - 0.5f;
color.rgb += d * DitherAmount;
```

Fade (code: $C \leftarrow \text{lerp}(C, F, f)$)

```
color.rgb = lerp(color.rgb, FadeColor.rgb, FadeAmount);
```

Radial Blur (mouse movement/click) (code: $C \leftarrow \frac{1}{N} \sum S(uv + t_i(uv - m))$)

```
float2 dir = (uv - MouseUV);
float w = saturate(MouseStrength);
color.rgb = (S(uv) + S(uv - dir*w) + S(uv - dir*2*w) + S(uv - dir*3*w)) * 0.25
    f;
```



3 Conclusion

CGP successfully implemented some real-time rendering techniques. This process provided great learning with DirectX 12, reading papers on computer graphics, and getting to know shaders a lot better. While the game experience is limited, the graphics output met my learning objectives.

References

[PJH18a] Matt Pharr, Wenzel Jakob, and Greg Humphreys. Lambertian reflection. `https://www.pbr-book.org/3ed-2018/Reflection_Models/Lambertian_Reflection`, 2018. Accessed: December 2025.

[PJH18b] Matt Pharr, Wenzel Jakob, and Greg Humphreys. Reflection models. `https://www.pbr-book.org/3ed-2018/Reflection_Models`, 2018. Accessed: December 2025.

[vO19] Jeremiah van Oosten. Learning directx 12 – textures and mipmaps. `https://www.3dgep.com/learning-directx-12-4/`, 2019. Accessed: December 2025.

[Wik25a] Wikipedia contributors. Blinn–phong reflection model. `https://en.wikipedia.org/wiki/Blinn%E2%80%93Phong_reflection_model`, 2025. Accessed: December 2025.

[Wik25b] Wikipedia contributors. Lambertian reflectance. `https://en.wikipedia.org/wiki/Lambertian_reflectance`, 2025. Accessed: December 2025.

(a) Mip level 0 (full resolution)

(b) Mip level 2

(c) Mip level 4

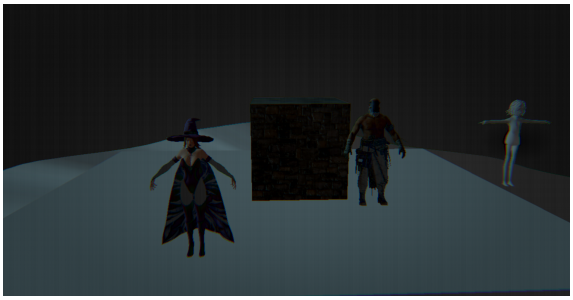
(d) Mip level 6

(e) Mip level 8

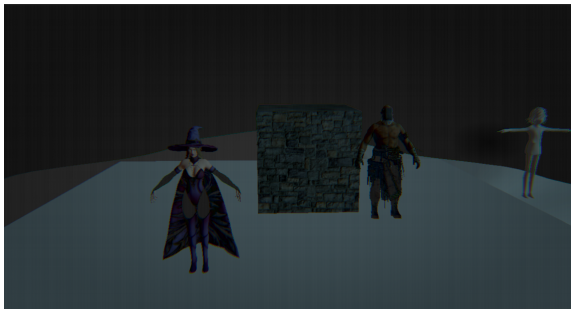
(f) Mip level 10

(g) Mip level 12

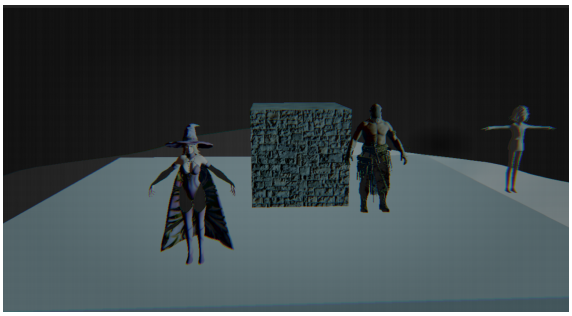
Figure 9: Compute-generated mipmap levels produced by CGP. Each successive mip level is generated by downsampling the previous level using a 2×2 box filter, progressively removing high-frequency detail.



(a) Directional light from the left

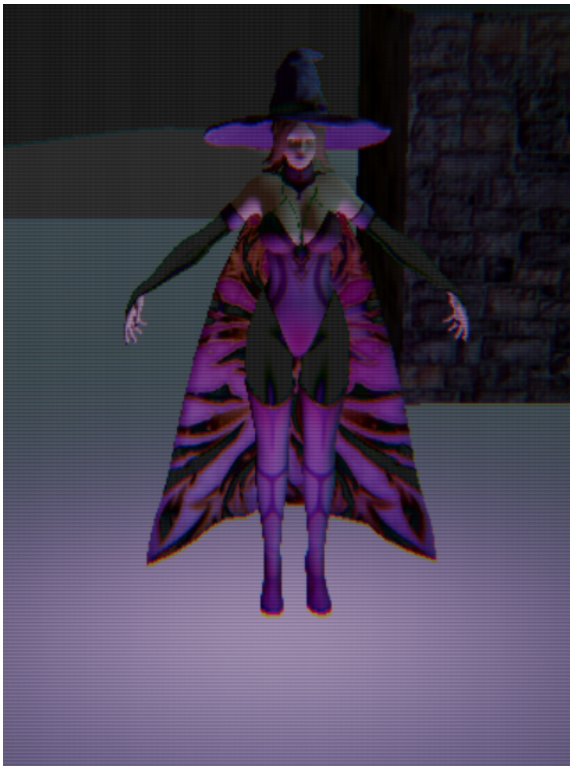


(b) Directional light from the right



(c) High-intensity directional light

Figure 10: Directional lighting results under varying light directions and intensity.

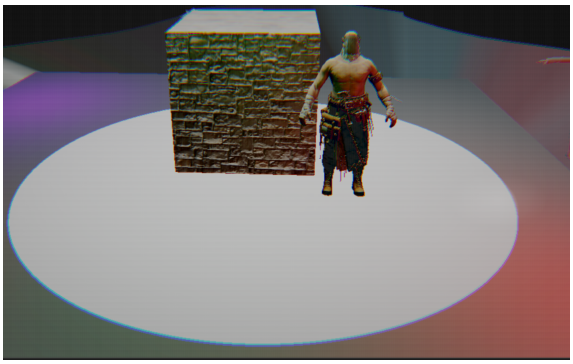


(a) Single point light

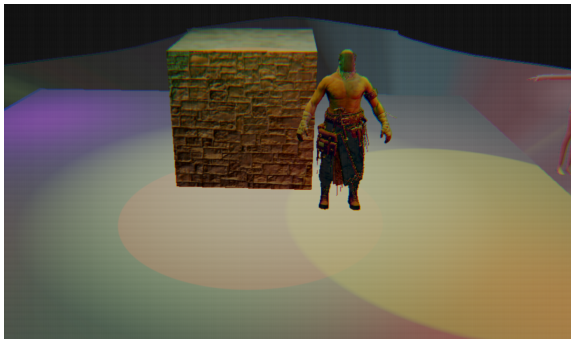
(b) Multiple point lights

(c) Diffuse + specular (Blinn–Phong)

Figure 11: Point light evaluation in CGP for single and multiple lights, and with Blinn–Phong specular highlights.



(a) Single spot light



(b) Multiple spot lights

(c) Soft spot light (increased cone softness)

(d) Spot light with narrower inner cone

Figure 12: Spot light evaluation in CGP. The images demonstrate cone-restricted illumination, accumulation of multiple spot lights, softness control at the cone boundary, and the effect of adjusting the inner cone angle.