The Mathematical and Physical Frameworks Behind the DirectX 12 Sphere Code

Neo

February 25, 2025

1 Introduction

This document explains the mathematical and physical frameworks that underpin a DirectX 12 application. The project integrates graphics rendering and GPU compute operations to render a dynamic 3D sphere while applying a water-like ripple effect, multiple moving light sources, and a shadow mapping technique. Topics discussed include the generation of 3D geometry, the use of transformation matrices, the implementation of a physically based lighting model enhanced by a ripple perturbation, orthographic projection for shadow mapping, and the synchronization mechanisms employed for GPU compute tasks. All vector and matrix operations in the project are handled using the glm math library, which provides a clean and intuitive API for transformations, normalizations, and other common operations. This was chosen after experimentazion with both the integrated DirectX 12 math functions and glm, glm provided better results such as surperior reflections and ripple effects, but also better unpredictable patterns in the movement of the sphere.

2 3D Geometry and Coordinate Transformations

2.1 Sphere Generation

The sphere geometry is generated by tessellating the surface of a sphere using latitude and longitude bands. A point on a sphere of radius r in spherical coordinates is defined as:

$$x = r\cos\phi\sin\theta, \quad y = r\cos\theta, \quad z = r\sin\phi\sin\theta,$$

where $\theta \in [0, \pi]$ (the polar angle) and $\phi \in [0, 2\pi]$ (the azimuthal angle). This method produces a smooth, evenly distributed mesh for the sphere.

2.2 Transformation Matrices

In the vertex shader, three key matrices are applied sequentially:

- **Model Matrix** M: Transforms object coordinates into world coordinates. In the code, M is updated every frame—via translations and rotations—to animate the sphere. The sphere's position can be made unpredictable by summing sinusoidal functions of time:

$$\text{randomX} = \sum_i \sin(\alpha_i \cdot t + \beta_i), \quad \text{randomY} = \sum_j \sin(\alpha_j \cdot t + \beta_j), \quad \text{randomZ} = \sum_k \sin(\alpha_k \cdot t + \beta_k).$$

- **View Matrix** V: Represents the camera's position and orientation, constructed using a look-at function:

$$V = \text{LookAt}(\text{eye}, \text{center}, \text{up}).$$

- **Projection Matrix** P: Projects 3D coordinates into 2D screen space using perspective projection:

$$P = \begin{bmatrix} \frac{1}{\tan(\frac{fov}{2})} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(\frac{fov}{2})} & 0 & 0 \\ 0 & 0 & \frac{f+n}{n-f} & \frac{2fn}{n-f} \\ 0 & 0 & -1 & 0 \end{bmatrix},$$

  where f and n denote the far and near clipping planes.

The vertex shader computes:

$$w = M \cdot v, \quad v' = V \cdot w, \quad p = P \cdot v',$$

with v as the original vertex position. In the implementation, glm is used for constructing and manipulating these matrices. Note that the matrices are explicitly transposed before being uploaded to the shader to match the expected layout.

3 Lighting and Shading Models with Ripple Effect

The pixel shader implements a physically based lighting model with an additional water-like ripple effect, combined with shadow lookups to attenuate lighting where the surface is in shadow.

3.1 Standard Lighting Components

- **Ambient Lighting:** Provides constant illumination,

$$I_{\text{ambient}} = k_a I_a.$$

- **Diffuse Reflection:** Follows Lambert's cosine law,

  $$I_{\text{diffuse}} = k_d I_l \max(0, \mathbf{n} \cdot \mathbf{l}),$$

  where k_d is the diffuse coefficient and $\mathbf{l}$ the light direction.

- **Specular Reflection:** Computed via a microfacet model with a Fresnel term,

  $$I_{\text{specular}} \propto \frac{D \cdot G \cdot F}{4(\mathbf{n} \cdot \mathbf{v})(\mathbf{n} \cdot \mathbf{l})},$$

  with D, G, and F representing the normal distribution, geometry, and Fresnel terms, respectively.

3.2 Water-like Ripple Effect

A ripple effect is introduced by perturbing the surface normals:

- A second constant buffer passes the current time time to the shader.
- A noise function noise(p) is used, where

  $$\text{noise}(\mathbf{p}) = \text{frac}\Big(\sin\big(\mathbf{p} \cdot \mathbf{c}\big) \cdot \gamma\Big),$$

  to add randomness.

- Ripple offsets are computed as:

  $$\text{rippleX} = \sin(\text{input.world\_pos.x} \times \text{frequency} + \text{time} \times \text{speed} + \text{noise}),$$

  $$\text{rippleZ} = \sin(\text{input.world\_pos.z} \times \text{frequency} + \text{time} \times \text{speed} + \text{noise}),$$

  scaled by an amplitude factor. These offsets are added to the original normal, yielding a perturbed normal that drives lighting calculations.

3.3 Multiple Light Sources

The code supports two moving lights, each with its own position and color components. Mathematically, the total diffuse and specular terms become the sum of each light's contribution:

$$I_{\text{diffuse\_total}} = \sum_{m=1}^{2} I_{\text{diffuse}}(m), \quad I_{\text{specular\_total}} = \sum_{m=1}^{2} I_{\text{specular}}(m).$$

Each light may have a different color or intensity, and their positions vary over time via $\cos(t)$ and $\sin(t)$ functions.

4 Shadow Mapping

4.1 Orthographic Light Projection

Shadow mapping involves rendering the scene from the light's perspective. In this code, an orthographic projection is used for the light's view:

$$\text{lightProj} = \text{Orthographic}(width, height, near, far),$$

where width = 20, height = 20, and typical near = 0.1, far = 50. The light view matrix lightView is built using a look-at function with the light position as the "eye." The combined matrix is:

$$\text{light\_view\_proj} = \text{Transpose}(\text{lightView} \times \text{lightProj}).$$

4.2 Shadow Factor

After rendering the depth map from the light's perspective, the pixel shader in the main pass computes a shadow factor. If the transformed coordinates of a fragment fall outside the $[0, 1]$ range or if its depth is greater than the stored depth in the shadow map, the fragment is considered to be in shadow. This is approximated with a factor, e.g., 0.7:

$$\text{shadow} = \begin{cases} 0.7, & \text{if in shadow,} \\ 1.0, & \text{otherwise.} \end{cases}$$

Thus, the final lighting is scaled by this shadow factor.

5 GPU Compute Shader Framework

A compute shader doubles each element in a buffer:

```
resultBuffer[DTid.x] = DTid.x * 2,
```

with 1024 threads dispatched via a thread group size of 64 in the x-dimension. An unordered access view (UAV) is created to allow parallel buffer writes.

6 GPU Synchronization

A fence mechanism (via the `wait_for_gpu` function) ensures that the CPU waits for the GPU to complete its work before proceeding with resource transitions or readbacks. This is especially important when combining shadow pass rendering, main pass rendering, and compute operations in a single frame.

7 Summary

This document details the mathematical constructs and physical principles forming the backbone of the DirectX 12 application. Key points include:

- **Sphere Tessellation and Transformations:** Generation of vertices using spherical coordinates, plus unpredictable motion via sums of sine functions.
- **Physically Based Lighting and Ripple:** Ambient, diffuse, and specular terms combined with sinusoidal normal perturbation to produce water-like surface waves.
- **Shadow Mapping with Orthographic Projection:** Rendering from the light's perspective using a specialized matrix, then sampling the shadow map to attenuate lighting.
- **Multiple Moving Lights:** Two lights orbit the scene, each contributing distinct diffuse and specular components.
- **Compute Shaders and Synchronization:** Parallel operations on the GPU with careful resource transitions and fence usage.

Together, these elements provide a comprehensive framework for high-performance rendering and computation, combining advanced real-time shadows, multiple light sources, and a procedural noise-based ripple effect on the sphere's surface.