Code Explanation: The DirectX 12 Sphere Application

Neo

February 25, 2025

1 Introduction

This document explains the structure and functionality of the DirectX 12 Sphere application code. The code integrates graphics rendering and GPU compute operations to display a dynamic 3D sphere with a water-like ripple effect, shadow mapping, and multiple moving lights. Key areas include initialization, resource creation, shader compilation, pipeline setup, and synchronization. The project uses the glm math library for all vector and matrix operations. glm provides a clean and intuitive API for transformations and normalization without adding any additional graphics dependencies. This consistent math framework simplifies operations in the code and ensures that lighting, normals, and procedural motion work as expected.

2 Initialization and Window Setup

2.1 SDL and Window Creation

The code initializes SDL (with video support) and creates a window at a resolution that is tweakable, with a default size of 800x600. The window retrieves the native window handle (HWND) required by DirectX 12 for rendering.

2.2 DirectX 12 Device and Factory

A DXGI factory and DirectX 12 device are created. A command queue is set up for command list execution, and a swap chain with two buffers is created. The code also checks for MSAA support and sets the quality level accordingly (up to 8x MSAA).

3 Resource Creation

3.1 Render Targets, MSAA, and Depth Buffer

Render target view (RTV) heaps are established along with multisample (MSAA) targets. A depth stencil view (DSV) heap and a depth buffer are created for proper depth testing. The code uses 8x MSAA for improved visual quality, with resource transitions for resolving the MSAA buffers into the final back buffers.

3.2 Sphere Geometry

The sphere is tessellated using latitude and longitude bands (40 each). Vertex and index buffers are generated and populated with computed positions and normals. The code uses glm for all mathematical computations (vectors, matrices, and transformations) to calculate positions, normals, and transformations. This consistent math framework simplifies the process of applying translations, rotations, and normalizations to create the dynamic sphere.

3.3 Shadow Map Resources

In order to implement shadow mapping, a separate depth resource (the "shadow map") is created with a dimension of `1024`×`1024`. This resource is used to store depth information from the light's perspective. A descriptor heap for the shadow map is also set up so that it can be sampled or transitioned appropriately.

4 Shader Compilation and Pipeline Setup

4.1 Compute Shader

A simple compute shader doubles each element of a buffer. The shader is written in HLSL, compiled, and a descriptor heap (UAV) is created. A compute pipeline state and root signature (with one descriptor table) are established, along with a dedicated command allocator and command list. These steps demonstrate how parallel GPU compute can be performed alongside graphics rendering.

4.2 Graphics Shaders

The graphics shaders include:

- A vertex shader that applies transformation matrices (model, view, and projection) to transform sphere vertices.
- A pixel shader that implements a physically based lighting model and introduces a water-like ripple effect. The ripple effect perturbs the original normal using sine functions based on world position, a noise function, and a time constant passed via a second constant buffer.

The shaders are compiled, and the root signature is updated to include two constant buffer views (one for scene constants and one for time).

4.3 Shadow Pass Pipeline

A separate graphics pipeline state is created for the shadow pass. This pipeline uses a vertex shader that outputs the position of the vertex in the light's clip space (via a light view-projection matrix) and omits the pixel shader. The goal is to render only depth information to the shadow map resource. A root signature for this pass includes a single constant buffer containing the model matrix and the light view-projection matrix.

4.4 Graphics Pipeline State

A graphics pipeline state object (PSO) is created using an input layout for position and normal data. The PSO includes default rasterizer, blend, and depth-stencil states; it specifies the render target format (`DXGI_FORMAT_R8G8B8A8_UNORM`) and depth format (`DXGI_FORMAT_D32_FLOAT`). The pipeline is configured to use the updated root signature and handles the ripple-based lighting calculations.

5 Command Recording and Execution

5.1 Shadow Map Pass

Before the main render pass, the code performs a *shadow map pass*:

- The shadow map resource is transitioned to a depth-write state.
- The scene is rendered from the light's perspective using a special vertex shader that outputs light-space coordinates.
- The depth buffer is populated with the distances of visible fragments from the light.
- The shadow map resource is then transitioned back for pixel shader usage in the main pass.

5.2 Graphics Command List

Inside the main loop, the graphics command list is reset and recorded:

- Render targets and depth buffers (including MSAA resources) are set and cleared.
- The root signature and pipeline state are set for the main pass.
- The vertex and index buffers are bound, and constant buffers are updated each frame with the scene data (e.g., light positions, camera position) and current time.

- The sphere is drawn using indexed instancing, sampling the shadow map to determine shadow visibility (e.g., a 0.7 attenuation factor).
- Resource barriers transition the MSAA render targets for resolving into the back buffers.

5.3 Compute Command List

A separate command list handles compute operations:

- The compute command list is reset, the UAV descriptor heap is set, and the compute root signature is bound.
- The compute shader is dispatched (16 groups with 64 threads each, yielding 1024 threads total).
- A readback buffer can be used to verify or log compute results.

5.4 Unpredictable Sphere Motion

In each frame, the sphere's model matrix is updated to create unpredictable motion:

- `randomX`, `randomY`, and `randomZ` are computed as sums of sine functions at different frequencies and phases.
- A translation matrix is built from these values, and a rotation is applied based on the current time.
- glm is used to perform these operations. The resulting model matrix is explicitly transposed (to match the layout expected by the HLSL shaders) before being uploaded to the scene constant buffer.

6 Synchronization and Main Loop

A fence mechanism (using the `wait_for_gpu` function) ensures that the CPU waits for the GPU to complete its commands before progressing. The main loop polls for events, updates animations (including two moving lights and the sphere transformations), records command lists for both graphics and compute operations, presents the frame, and synchronizes using the fence.

7 Summary

The code demonstrates a comprehensive DirectX 12 application that integrates:

- **Initialization and Resource Setup:** Window creation via SDL, device and swap chain setup, and resource allocation for render targets, depth buffers, and a shadow map.

- **Shader and Pipeline Configuration:** Compilation of compute and graphics shaders (with a water-like ripple effect and shadow mapping), setup of root signatures, and configuration of both compute and graphics pipeline states.
- **Command Recording and Synchronization:** Separate command lists for shadow rendering, main rendering, and compute tasks, execution of resource state transitions, and synchronization via fences.
- **Unpredictable Motion and Multiple Lights:** Two moving light sources cast dynamic shadows, and the sphere itself undergoes a combination of sinusoidal translations and rotations for a visually engaging real-time demo.
- **Integrated Math Operations:** All vector and matrix computations are handled using glm, which simplifies transformations and normalizations while preserving the DirectX 12 rendering pipeline.

These elements together form a robust framework for rendering a dynamic 3D sphere with visual ripple effects, realistic shadows, and GPU compute capabilities.