CO2409 Assignment Report

Kyriacos Rediu

# Introduction

This report is about the assignment for the **CO2409 - Computer Graphics** module, which focuses on real-time rendering techniques using Direct3D 11. The assignment involves creating a scene and applying a variety of shading techniques and other additional visual effects. Through sufficient experimentation with vertex and pixel shaders, this work demonstrates practical understanding of modern rendering pipelines and shader-based programming.

# Shader Development Process

Shaders are the foundation of the modern graphics pipeline, allowing graphics developers to control the appearance and behaviour of rendered objects at an extreme level. A shader is a small program that runs on the GPU, written in a specialized language such as the C++ based High-Level Shading Language (.hlsl) for DirectX applications. These programs are executed in various stages of the pipeline—most commonly as vertex shaders and pixel shaders—to transform vertex data and compute final pixel colours (Croteau & Saito, 2012).

In a typical rendering pipeline, vertex shaders are first used to create the geometry of 3D models. They transform vertex positions from local object space to world space, and may also pass additional data such as normals, texture coordinates, and lighting vectors to the next stage. Once the geometry is transformed using this data, the rasterizer then determines which pixels are covered by the geometry, and then the pixel shader is executed per pixel. The pixel shader is responsible for determining the final color of each pixel, typically by combining texture data, lighting information, and material properties (Cem Yuksel, 2016).

To use shaders in a Direct3D 11 application, the shaders are first written in HLSL and compiled into Compiled Shader Object (.cso) files. These files are then loaded into the application and assigned to specific objects such as models during the rendering process. For example, a textured cube might use a shader that blends two textures using linear interpolation, while another object might use a custom shader to produce a wiggling texture effect. The application must also manage constant buffers, which are used to pass time-varying or per-object data from the CPU to the GPU (Luna, 2012).

# Implementation

Below is a comprehensive list of geometry and shader techniques utilised in this assignment, with description on their specific application:

## Basic Scene Implementation

An initial scene was created with sufficient geometry to demonstrate present techniques. As a foundation, a model for hills with a shallow point for the remaining models was used. The level furthermore contains:

- a cube and a sphere used for most shader applications,

- a teapot required by the brief

- a complex container crate model

- a ‘portal’ object

- and two point light models used for main lighting.

## Lighting Modes

All geometry in the scene utilises diffuse, specular and ambient lighting, all calculated in main pixel shader files. While ambient lighting is globally applied by a single variable in the final equation, the rest is displayed by utilising the 2 point light models. The first, static point light model is placed high up and pulsates on and off, with significant peak light strength. The second light is placed to rotate around the cube, with static but lower intensity than the first one, and is made to change colour in the full RGB range.

Initially, the light vectors are calculated and used to deduce their length/distance. The vectors are then normalized so that they can be used in lighting calculations. The diffuse lighting calculation takes the dot product between the normal and light direction to determine the light applied on a surface, after which it is scaled by the light color and attenuated properly by the length/distance. The specular light calculation takes the halfway vector of the light and camera directions, and takes its dot product with the normal, amplified by a specular power variable to calculate how shiny a surface is. This result is then scaled by diffuse lighting to ensure shine appears only where diffuse lighting exists.

Diffuse color and specular intensity are then calculated by sampling provided diffuse/specular maps and extracting the RGB and A values respectively for application. They are applied to their respective lighting and then combined altogether with the ambient colour to create the final colour for geometry.

Note that the sphere object, as per brief instructions, has a tint additionally applied.

## Wiggle/Scroll Effect

A conjunction of wiggle and scrolling effects are applied to the sphere texture by manipulating texture UV coordinates. A sine wave is applied to both U and V coordinates with a texture shift factor variable that is a multiplication of delta time to create a wiggling effect for the sphere’s texture in both axes. Both U and V coordinates are also shifted at the rate of a speed value multiplied by the same shift factor to create a scrolling effect, where the texture constantly moves in a direction.

## Linear Interpolation for Texture Change

The cube gradually changes textures back and forth from a stone to a wood texture using linear interpolation. During rendering, two textures are loaded and passed to the shader, and change to each other on the cube by extracting diffuse and specular colour values and rotating between them respectively using the lerp() function and the texture shift variable before being passed to the final colour equation.

## ‘Portal’

As mentioned, this scene includes a ‘portal’ model. This is essentially a rendered view from a different camera perspective mapped onto a framed shape model. This model is created live in the main scene file, sent to shaders using shader-resource view, and then utilises a depth buffer to be used as the texture.

## Blending modes

Four blending modes have been implemented in the project: none, additive, multiplicative, and alpha. However, only two are used. Default (none) blending is used on most of the geometry, while additive blending is used on the light models. Multiplicative and alpha blending have been experimented with but are not used due to lack of actual application in the scene.

## Controls

The controls implemented in the program are as follows:

- WASD keys control camera movement, front/back and left/right

- Arrow keys control camera rotation, up/down and left/right

- U, I, P, J, K, L keys control sphere rotation on the XYZ axes

- Comma and full stop keys control back and forth sphere movement

- P key locks FPS

- Esc key quits the application

## Code Management

The code structure is decently organized. Objects and functionalities are split into classes with their respective .h and .cpp files to promote flexibility and convenience. Arrays or sorting classes are not used, as the number of models was considered too small for this application – this is however not optimal. In a higher-grade implementation, more models would’ve been used for demonstration and therefore a form of organization for these objects would be present. Encapsulation is present to a decent degree but is not applicable everywhere as most classes exclusively contain methods and variables used in multiple points throughout the program. Global variables are unfortunately still used.

# Potential Improvements/Additions

A lot of content stated in the brief could not be implemented due to time restrictions. While specular maps have been added, normal and parallax mapping could have been simultaneously applied to the cube and sphere for multiple effect display. The light setup could have also been improved by making one of the lights function differently, such as a directional or spotlight, preferably the rotating one. Some models, specifically the teapot and container, are underutilised and could have been a display of other effects. The container could have been used to apply the cube mapping technique specified in the brief, and the teapot could have been used as a showcase for some mapping or shadowing methods. Lastly, the code management could also have been improved by fully encapsulating objects regardless of use and creating more classes for things such as textures and lights.

# References

Croteau, E. and Saito, H. (2012) *Introduction to 3D Game Programming with DirectX 11*. Mercury Learning and Information.

Luna, F.D. (2012) *Introduction to 3D Game Programming with DirectX 11*. Course Technology PTR.

Yuksel, C. (2016) *Introduction to Modern Shader Programming*. University of Utah. Lecture Notes.