**Semester 1 – Assignment Report**

Arnav Datt Mehta

Contents

[Introduction 2](#_Toc92298950)

[Project Description and Aim 2](#_Toc92298951)

[Project Panning 2](#_Toc92298952)

[Classes In Project 2](#_Toc92298953)

[Utility Classes 2](#_Toc92298954)

[Core Classes 2](#_Toc92298955)

[Loaders And Helpers 3](#_Toc92298956)

[Texture Loader 3](#_Toc92298957)

[OBJ Loader 3](#_Toc92298958)

[JSON Loader 3](#_Toc92298959)

[ImGui 3](#_Toc92298960)

[Vector3 4](#_Toc92298961)

[Matrix 6](#_Toc92298962)

[Texture 6](#_Toc92298963)

[Light 6](#_Toc92298964)

[Core Classes 8](#_Toc92298965)

[The Core 8](#_Toc92298966)

[Game Object 9](#_Toc92298967)

[Cameras 12](#_Toc92298968)

[Camera Base (Static Camera) 12](#_Toc92298969)

[Orbit Camera 13](#_Toc92298970)

[First-Person Camera 14](#_Toc92298971)

[Sky Sphere 15](#_Toc92298972)

[The Terrain Class 16](#_Toc92298973)

[The Application Class 18](#_Toc92298974)

# Introduction

This project report aims to breakdown the working of the DirectX 11 Project made for Level 5 – Semester 1.

## Project Description and Aim

The following project is made using the DirectX 11.0 API and is written in C++ using Visual Studio 2019/2022 as the IDE of choice. It can be accessed / modified using the aforementioned IDE and executed by running the executable provided.

The aim of the project is to create an application that demonstrates various graphical implementations along with provide ways to interact with the application.

# Project Panning

Before jumping into code, a small plan of action was decided along with a couple of personal goals for this specific project.

Following are the personal goals for this project.

* All the code written in the project should be clean and easily readable / understandable.
* The code systems designed should be dynamic and decoupled which can be easily expanded on for future framework purposes.
* Code should be dynamic enough to eliminate all requirements of hard coding and magic numbers.

# Classes In Project

The classes in the project are divided into filter as per their use case

## Utility Classes

These are the helper classes of the project; they are standalone classes that can be used without dependencies that are used for the functioning of the application:

1. OBJ Loader
2. Texture Loader
3. JSON Loader
4. ImGui
5. Vector3
6. Matrix
7. Texture
8. Light

## Core Classes

These classes are the actual functional classes that work with the help of the above-mentioned classes:

1. Game Object
2. Camera
3. Orbit Camera
4. First Person Camera
5. Sky Sphere
6. Terrain
7. The HLSL Shader File

All these classes combine to provide a functioning prototype for a rendering engine. Below is an in-depth description about all the classes in the project.

# Loaders And Helpers

The project contains three loaders and four helper classes / files:

## Texture Loader

The texture loader is a class that is provided by Microsoft and is used to load texture files with the .DDS extension. (Microsoft, n.d.).

The custom Texture class uses this loader to make it easy to access texture files (in the form of Shader Resource Views).

## OBJ Loader

The OBJ loader is a class provided by the course module for this project and is used to load Wavefront OBJ files as 3D models into the application.

The Game Object class uses this loader to set meshes for each game object in the scene.

## JSON Loader

The JSON loader being used in this project is ‘JSON for Modern C++’. It is a header only loader and is used for reading and writing json files into the project. (Lohmann, n.d.)

The Game Object class uses this loader to initialize itself by reading and initializing json file.

## ImGui

Dear ImGui is a bloat-free graphical user interface library for C++. It is used to make quick and easy UI for applications and projects of such type.

In this project ImGui acts like an interface between the user and the application as the user controls the render settings using the set-up UI.

## Vector3

The Vector3 class is a custom class written with behaviors and characteristics that help re-create vectors from physics/mathematics.

The class contains:

* Overloaded class constructors to initialize the ‘x’, ‘y’ and ‘z’ components of the vector.
* Overloaded operators to perform various arithmetic functions.
* Vector math functions such as getting the Dot and Cross product, Normalization, etc.
* Static functions for clamping vector values and to convert Vector to XMFLOAT3/XMVECTOR and vice versa.
* A debug method to easily debug the vector class values by printing a formatted version of the vector (with or without a message)

Below are the code snippets that show all the Vector Math Functions in the class:

#### Dot Product



#### Cross Product



#### Normalize



#### Distance Functions



#### Debugging



#### Static Convertors



## Matrix

The matrix class is a templated class and can be used to hold data in a matrix like format and also has methods used to perform matrix arithmetic. Along with that it also has functions to convert to and from XMMATRIX and XMFLOAT4X4.

## Texture

The texture class is used as a container class that contains the Shader Resource View for a texture



## Light

The light class is a header only file, it contains all the structs regarding lighting. There are two types of lights in the project: Directional Light and Point Lights. Each light has two properties, the light intensity properties and the light material properties. Both the intensity and the material have the color for the ambient, diffuse and specular lighting

The Intensity Struct

The Material Struct  


#### The Directional Light Struct



#### The Point Light Struct



All these struct were made keeping in mind the constant buffer packing. Further references for making the Point lights were taken from the book 3D Game Programming with DirectX.

# Core Classes

There are eight core classes in this project that work in conjunction with the helper classes.

## The Core

The ‘Core’ is a header only file, it contains all of the common includes for easy setting up of includes and preventing circular dependencies. Along with that it also contains the struct for ‘Simple Vertex’ and ‘Constant Buffer’.

#### The Simple Vertex Struct



#### The Constant Buffer Struct

The constant buffer is made with buffer packing size mind.

## Game Object

Every visible object in the application is a Game Object. The game object can be an empty data structure or it can have a mesh assigned to it. The game object class can be initialized in two different ways. Either by passing the initialization data or by passing initialization file, the device context and the constant buffer. The initialization file is a JSON file. It contains the position, rotation and scale of the object.

If the initialization file is passed, then the file is read using the JSON parser, then the initialization data struct is created from the file data.

#### The Game Object Constructor (Init File Based)

#### 

#### The Game Object Init Data



#### The Draw Functions

There are two draw functions in the Game Object class. One is to draw just the mesh and the second is to draw the mesh with a texture (provided there is a texture assigned).

##### The Standard Draw Function



##### The Draw Textured Function



The game object also has getter and setter functions for the world matrix, position, rotation and scale. Both the update and draw functions can be overridden by child classes.

The Update function just calls another function which updates the world matrix

#### Updating the World Matrix



#### Setting Game Object Mesh

The mesh of the game object can be set by passing the mesh file path. It makes use of the OBJ Loader.

## Cameras

There are three different types of cameras in the project.

### Camera Base (Static Camera)

The camera base class is a static camera class. This class contains all the behavior to make custom cameras by just inheriting from this class.

Along with that this class also has two important structures that are used for initializing and storing the camera data:

#### Camera Initialization Data Structure

This structure is used to initialize the camera with all the details required to set up the camera matrix.



#### Camera Matrices Structure

This structure stores the view and projection matrices. This structure is used for updating the ‘current camera’ in the main program as the view and projection matrices of different cameras are set as current



Apart from the two structures, the camera class also has getter and setter functions for position, up, look at direction, near and far plane.

And finally, there is a virtual Set Lens function which creates the view and projection matrices for the camera.

#### The Set Lens Function (Virtual)



### Orbit Camera

The Orbit Camera Class inherits from the camera base class and overrides the update and the set lens functions.

#### The Update Function

The Update function of the orbit camera does the heavy lifting rotating the orbit camera around the look at point. It is done using the circle equations. Using Sin and Cos functions, I get the X and Y coordinates on the circumference of the circle of set radius. The also angle is constantly updated and clamped between 0 and 360. After which it is converted to radians and used in the calculations.



### First-Person Camera

The First-Person Camera also inherits from the base camera class and overrides the Update and Set Lens functions. The initialization of this camera is also different.  
Along with the initialization data, the look direction is also passed into the constructor. The look direction is a normalized vector that dictates what direction the camera will be looking at when it’s view and projection matrices are constructed.

#### The Update Function

The update function handles the change in movement, look direction and height.







## Sky Sphere

The Sky Sphere class inherits from the Game Object class. Since the sky sphere is just a sphere with a texture, I decided to convert it to its own class/object.

The sky sphere too has two constructors similar to the game object, which means that this too can be initialized using a JSON file. The difference between the JSON file for initializing a game object and sky sphere is that the sky sphere has a key for storing the texture file path as well.

#### The Constructors



## The Terrain Class

The terrain class does exactly what it sounds like, generate a large chunk of terrain. It first generates a height map from an `RAW` input image. This height map is then multiplied with the height scale when creating the vertices for the vertex buffer. *(Notes taken from 3D Game Programming With DirectX)*

#### Generating the Height Map



#### Creating The Vertex Buffer



#### Creating The Index Buffer



# The Application Class

The Application class is where all the above-mentioned class come together to create a working graphics application. Here is a walkthrough as to how the application class works.

It is divided into three sections

1. The window procedure callbacks, constructor, destructor and cleanup methods
2. The initialization chains
   1. Base initialization chain
   2. My custom initialization chain
3. The update and draw functions

## The Initialization Chains

As mentioned above the, the initialization chain consists of two other chains, one required for the setup of the application, and the other, required for the setup of the custom components.

### Base Initialization Chain

The two important functions I would like to talk about in the base initialization chain, are the ‘Initialize Vertex Buffer’ and the ‘Initialize Index Buffer’ functions. These are the functions where I define the