**Semester 1 – Assignment Report**

Arnav Datt Mehta

Contents

[Introduction 3](#_Toc92440689)

[Project Description and Aim 3](#_Toc92440690)

[Project Panning 3](#_Toc92440691)

[Classes In Project 3](#_Toc92440692)

[Utility Classes 3](#_Toc92440693)

[Core Classes 3](#_Toc92440694)

[Loaders And Helpers 4](#_Toc92440695)

[Texture Loader 4](#_Toc92440696)

[OBJ Loader 4](#_Toc92440697)

[JSON Loader 4](#_Toc92440698)

[ImGui 4](#_Toc92440699)

[Vector3 5](#_Toc92440700)

[Matrix 7](#_Toc92440701)

[Texture 7](#_Toc92440702)

[Light 7](#_Toc92440703)

[Core Classes 9](#_Toc92440704)

[The Core 9](#_Toc92440705)

[Game Object 10](#_Toc92440706)

[Cameras 13](#_Toc92440707)

[Camera Base (Static Camera) 13](#_Toc92440708)

[Orbit Camera 14](#_Toc92440709)

[First-Person Camera 15](#_Toc92440710)

[Sky Sphere 16](#_Toc92440711)

[The Terrain Class 17](#_Toc92440712)

[The Application Class 20](#_Toc92440713)

[The Initialization Chains 20](#_Toc92440714)

[Base Initialization Chain 20](#_Toc92440715)

[Initializing Vertex Buffers for Hard Coded Meshes 20](#_Toc92440716)

[Initializing the Index Buffers for Hard Coded Meshes 21](#_Toc92440717)

[Compiling Custom Vertex and Pixel Shaders 21](#_Toc92440718)

[Custom Initialization Chain 22](#_Toc92440719)

[Initializing Cameras 22](#_Toc92440720)

[Initializing Lights 23](#_Toc92440721)

[Initializing Textures 23](#_Toc92440722)

[Initializing Models 24](#_Toc92440723)

# 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 designed keeping the constant buffer packing in mind. That also allows for multiple lights in the scene / light arrays without ruining the packing order of the constant buffer.

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 structure that contains all the data that is passed to the GPU at runtime. It contains the WVP matrices, the directional and point lights along with a couple of control variables.

## 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.



The Q and E keys can be used to increase and decrease the height value of the camera.

### 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.







Use the WASD keys to move around in the world, arrow keys to change the look direction, page up and down to change the height of the camera and TAB to flip the camera to look behind.

## 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)*

The terrain class is initiated by passing in the heightmap file, along with the height map info. This info structure includes the width and height of the heightmap image and the height scale multiplier. These values are used when generating the heightmap vector.

#### Generating the Height Map



#### Creating The Vertex Buffer

For the vertex buffer, the vertex vector is created, whose size is the product of the dimensions of the heightmap image. Then the positions of the vertex are altered on the y-axis to give the impression of height.



#### Creating The Index Buffer

The index buffer was too created with the help of the book. Here I just set the indices in a vector and create the index buffer.



#### Terrain Buffers Struct

The above created terrain buffers are stored in a ‘Terrain Buffers Struct’. This structure is used like an interface to access the terrain buffer data when drawing the terrain.



# 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 vertices and indices of the hard coded meshes in the scene.

### Initializing Vertex Buffers for Hard Coded Meshes

There are three hard coded meshes in the scene. A cube, a pyramid and a plane. The vertices of the cube and pyramid are entered manually. Whereas the vertices of the plane are generated at run time.

#### The Plane Vertex Buffer

To generate the plane vertex buffers, the size of the plane is taken as input. There is no change in the height value of the plane.



### Initializing the Index Buffers for Hard Coded Meshes

The next step in the initialization chain is to generate the index buffer for all the hard coded meshes. Here again, the indices of the cube and the pyramid are manually input whereas the indices of the plane are generated at runtime.

#### The Plane Index Buffer

To create the plane index buffer, the size of the plane on the X and Z axis is taken as input. The generated indices are then stored a vector of type WORD.



### Compiling Custom Vertex and Pixel Shaders

Another task executed in the base initialization chain is the compilation of custom vertex and pixel shaders for various objects.

There is one custom vertex shader for the hard coded plane and two pixel shaders for the sky sphere and terrain.

## Custom Initialization Chain

This is a chain of functions that are called after the base initialization chain has completed. This chain consists of the following functions

1. Initialize Cameras
2. Initialize Lights
3. Initialize Textures
4. Initialize Models

### Initializing Cameras

This is where all various cameras in the scene are initialized. Due to the dynamic nature of the camera class, it is very easy to create multiple cameras. There are 4 cameras in the scene: 2 static cameras (front view and top-down view), 1 FPS camera and 1 orbit camera.

They all are initialized using the same data with a few changes for each camera.



At the end of the creating the cameras, the front view camera is set as the default camera of the game. The ‘m\_CurentCamera’ variable is of [Camera Mats](#_Camera_Matrices_Structure) type. This stores the view and projection matrices of the current camera being used and helps in easy switching of cameras.

### Initializing Lights

This function is a very basic function, it just defines the defaults values of the directional and point lights in the scene lights in the scene.



### Initializing Textures

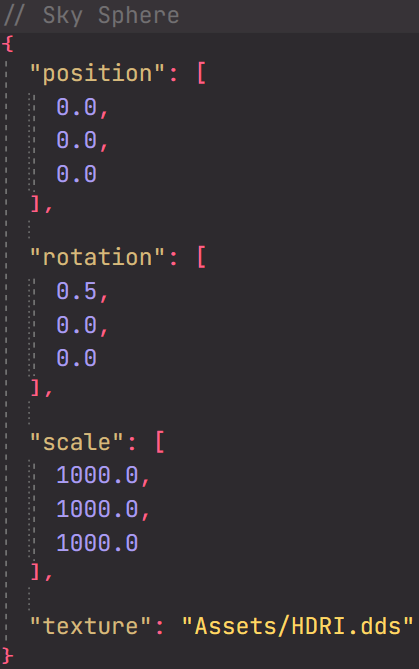
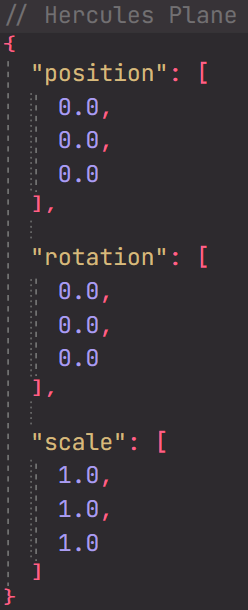
This function loads in the texture for the cube and stores it in a shader resource view and creates a linear sampler.



### Initializing Models

In this function all the visible objects in the scene are initialized. Starting with the 3D plane game object provided, followed by the sky sphere and the terrain. (The name of the plane is Hercules)



Here is how the JSON file for the Hercules Plane and the Sky Sphere look like:

# The Update Function

The update function performs three main tasks:

1. Update the game objects
2. Move the hard coded meshes
3. Update cameras and current camera view and projection matrices

The update function also updates the ‘time’ variable, which is passed to the shader via the constant buffer



# The Draw Function

In this function everything is draw onto the screen. This function too is divided into three parts:

1. Render Setup
2. Draw Game Objects
3. ImGui

## Render Setup

Here is where render target is cleared and the constant buffer is updated with the variables.



## Drawing Game Objects

In this part of the draw function, everything seen in the application is drawn. The application is divided into three scenes.

**Scene 1:** Demonstration of imported OBJ file  
**Scene 2:** Demonstration of hard coded meshes and different vertex buffer  
**Scene 3:** Demonstration of terrain generation and different pixel shader.

There are a couple of things common to all scenes – the sky sphere and render options. The use can choose whether the scene is globally displayed in transparency or wireframe. When in any of the previous mentioned render states, the sky sphere is not rendered, allowing for better visibility.

Many of the features of this part of the draw function is controlled using ImGui UI.

### Setting Transparency

The global transparency state and its color is controlled by bool set via ImGui.



### Drawing the Sky Sphere

Since the sky sphere is an unlit surface, it uses a different pixel shader. Along with that, since it is a very large inverted sphere, the face culling state too has to be changed. The rasterizer state and the pixel shader have to be set and reset before and after rendering the sky sphere.



### Drawing Scenes

The various scenes are drawn only of a boolean for the scene is set active.

#### Scene 1

Scene 1 is simplest of all scenes as it contains only the Hercules Plane and nothing else. It uses the draw textured function of the game object class to draw the plane with provided texture.



#### Scene 2

Scene 2 has the greatest number of objects being rendered on the screen at once. This is the scene where all the hard coded meshes are drawn all at once. I start by drawing the pyramid, followed by the cube and lastly the grid plane. But before the scene is drawn the texture is changed to a display a texture of a crate.

The plane is drawn at the end because it uses a different vertex shader that creates circular waves. The vertex shader has to be set and reset before and after rendering the grid plane.

#### Scene 3

This scene has the largest of all objects being rendered onto the screen: The Terrain.

To draw the terrain, first the vertex buffer and the index buffer are retrieved from the terrain mesh and applied. After which the pixel shader is changed. Since the terrain does not use any textures, the default pixel shader cannot be used.

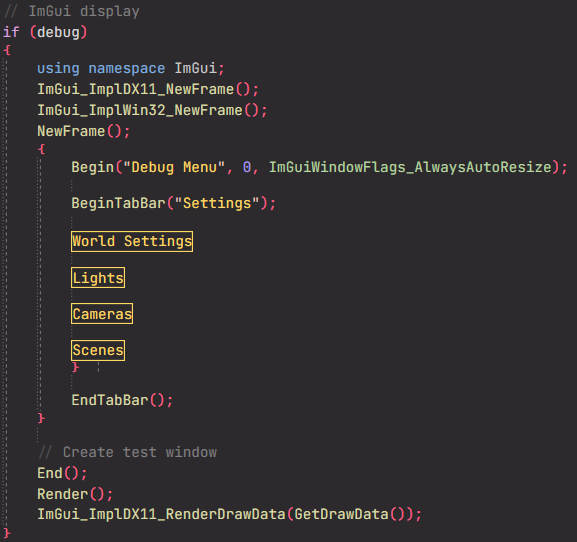
After drawing the terrain, the pixel shader is reset to the default pixel shader.



And with that all the Game Objects have successfully been rendered onto the screen.

### ImGui UI

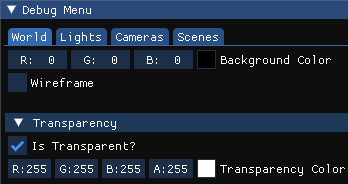
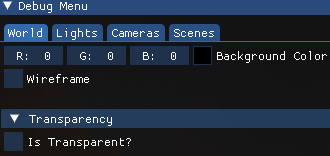
I am using ImGui as my means of interaction between the user and the application.



As you can see it has 4 sections

#### World Settings

Here is where I let the user change the render states from solid/wireframe and allow them to set transparency (and its color).



#### Light Settings

In this section I control the directional and point light parameters. The user can change the direction/position of the lights, along with other paramaters.