Project title (include tutorial number?)

Mihai Constantin

# 1.Technical design

For the purpose of this prototype DirectX 11 or more specific Direct3D 11 graphics was used to create a 3D game application. Direct3D 11 Graphics helps in development of 3D applications by creating a connection with the GPU.

## 1.1 Initialise Direct3D

Intialisation of the d3d will be made be passing in a handle to our application “HWND”, screen height, screen width during the initialization process we are to create a dx factory, to get access and implement methods for generating dxgi objects. After that we can define which adapter we want to use, when using “EnumAdapters” method from the factory by pasing the value “0” we are telling it to retrieve the primary video card available to our system, which would be the dedicated graphics card if we have one. Then we target the output(monitor) with the function “EnumOutPuts” passing the value of “0” we specify to use the first monitor available that is connected to the GPU.

After all the details are set related to our devices available to the system the swapchain can be created. The swapchain will be used to swap the frames while rendering. Because we wan’t to draw the whole scene and not just one object at a time for instance, everything must be drawn on a backbuffer first and that collection of data will be swaped with the help of the swapchain to the frontbuffer. Then we define the states belonging to the rendering pipeline, a stencilbuffer and rasterizer and bind them to the devicecontext.

After initialization of the d3d graphics interface we will have access to methods for creating and accessing resources using the ID3D11Device and then while we are rendering the ID3D11Device Context will be used for setting different resources for rendering.

## 1.2 Textures

The textures will be imported into the game with the use of the DirectXTex library. Currently in the project the DDS file format will be used for the texture resources. Using this library we will be able to retrieve data that will be used to create a shader resource (CreateShaderResourceView) using the d3ddevice and the data generated using LoadFromDDSFile method from DirectXTex library. A reference to the resource will be stored in the texture being able to be assigned later to a object.

## 1.3 Model/mesh

The model will be imported using the TinyObj library, offering methods of retrieving data from “obj” file formats. The data stored in this files is related to vertex positions, texture coordinates , vertex normal and polygonal faces. Those are the ones we are interested in so when in creation of the model we have to consider iterating through all this data and structure it accordingly. Before we store the data in a format that fits the buffers for a better understanding floats will be used instead of DirectXmath library’s data structures:

struct DirectXVertexStructure

{

XMFLOAT3 position;

XMFLOAT2 texture;

XMFLOAT3 normal;

}

struct NormalVertexStructure

{

float x, y, z;

float tu , tv ;

float nx , ny , nz ;

}

Each vertex will have the data stored in a structure of type NormalVertexStructure while retrieving data from the obj file and then stored in the structure that fits the directx vertex and index buffers( DirectXVertexStructure).

## 1.4 Shaders

Now that we have the resources prepared to render something on the screen the data collected until now must be draw onto the screen with the help of shaders.

In the HLSL file of the vertex shader to position the object in the world a wvp matrix should be used so we pass the world, viewmatrix and projection matrix through a constant buffer as this will change over the run time of the application and we want to update them into the shaders as well. Defining the shaders data structures should match the data in our vertex buffer position, texture coordinates and vertex normal. Because each vertex is a point in the world the position vector should include the “w” component as well ( x,y,z,w) where “w” is set to 1. Because we define a vertex as a point with the component w equal to 1 we can properly transform the pixel.

Overview of the vertex shader data:

Constantbuffer

{

Matrix worldmatrix;

Matrix viewmatrix;

Matrix projectionmatrix;  
}

Vertexinput

{

Vector4 position = x, y , z, w; ( where w =1)

Vector2 texture= tx,ty;

Vector3 normal=x,y,z;

}

In the HLSL file of the pixel shader we adjust the colour of the pixel that belongs to the model using the texture data from the sampler and then output the colour of the pixel. This is the result that is being displayed on the rendering target(display).

## 1.5 Camera class

One of the main components of the camera is the view matrix which is used to place the models in the world. The camera is just a concept inside graphics because we are not moving the camera, rotate or scale it in any way. The View matrix is just a result of a concept of a position and a rotation of the camera in the world which will be used to transform all the objects within the world instead of moving the camera as an object in the world.

A second component is the view frustrum, his attributes:

* FoV – field of view the angle through which we can see the world
* The screen / window aspect ratio – width divided by height
* screennear – how close can be the objects to be seen
* screenfar – how far can the objects be to be seen

From the same components we can create the projection matrix. The projection matrix is used to project objects from the 3d space into a 2d space format so we can display them onto the screen.

## 1.6 Light in the scene

The light in the scene has to be implemented in the pixel shader stage.

The light in the scene is being calculated using the normals present of each face of the model.

To determine the amount of light(light intensity) of each of the faces that the object has we can calculate the dot product between the normal and the direction vector of the light. The result must be kept between the values of 0 to 1, where 0 means that the face is not touched by the light and the maximum 1 where the face is pointing straight to the light. Any value more than 0 will receive some light and to determine the final amount of diffuse colour we multiply the light diffuse colour with the light intensity.

Then for the final result that will be the colour of that pixel on the display we multiply the result of the previous multiplication with the colour of the texture:

lightintensity = saturate(dot(normal \* lightdirection);

colour = saturate(diffusecolour \* lightintensity);

colour = colour \* texturecolour;

For a better representation of the classes to be implemented so far a diagram was made:

Diagram, schematic

Description automatically generated

## 

## 1.7 GameObject

A game object is defined by a position in the world space, rotation and scale and has two other components already defined, a texture and a model (mesh).

To position, rotate and scale the object, we must first define a model matrix. This matrix will be used along with the view matrix from the camera that will position the object in the world creating a World View matrix. Now that we can position the object into the world , the 3rd component the projection matrix will be used to project the object into a 2d surface to display it on the screen.

## 1.8 GameObjectManager

The main purpose of this entity is to act as a container for all the gameobjects inside the game. This will act as a bridge between the graphics and the actual game as well. As while the game is running objects might be added or removed and this will enable us to do it dynamically.

HWND – handle to window application

HINSTANCE – handle to the instance of the application

LPCWSTR - Long Pointer to Constant Wide String

# Test plan

# GiT commit log

# Schedule