How I used my Math Engine, Rendering Engine and 3D shape classes to display a box, pyramid, sphere, cylinder and cone

This program was made using Visual Studio 2022 v143. The Windows SDK Version was 10.0.20348.0. It was compiled using C++17.

Setup

Created a project in Visual Studio with the application type being Desktop Application (.exe).

After I made the project and a main.cpp file, I right clicked the project name and went to properties, then C/C++ -> General. In Additional Include Directories and I included the locations of the directories FA Math Engine\Header Files, the FA Rendering Engine\Header files, and the FA Shapes\Header Files. In the rendering engine header files, there is a directx directory that also was included. This allowed me to use all the necessary functions from my engines and shape classes to make the program.

I right clicked Source Files under the Project name tab and went to Add -> Existing Item. Navigated to the directory FA Rendering Engine\Source Files and added all the .cpp files to the project. Also went to the directory FA Shapes\Source Files and all the .cpp files to the project. There are no .lib files so the .cpp files need to be compiled and linked to be able to use the functions.

I also included all the header files in the project too, by right clicking Header Files under the Project name tab and going to Add ->Existing Item. I navigated to the directories of FA Math Engine\Header Files, FA Rendering Engine\Header Files and FA Shapes/Header Files and add all the files in those directories to the project.

Making the Program

Error Handling

In main I handled the exceptions my functions could potentially throw.

A DirectXExcepetion gets thrown if there Direct3D or DXGI error.

An out_of_range exception gets thrown if an index is out of bound or a key does not have a mapped value.

All other errors are handled by runtime_errors.

When an error happens a description of the error gets shown through a MessageBox and the program terminates.

Window Procedure

Created a window procedure to handle window events.

The events that I handled were:

WM_ACTIVATE for when the window gets activated/deactivated.

WM_SIZE for when the window gets resized.

WM_ENTERESIZEMOVE for when the user grabs the resize bars.

WM_EXITSIZEMOVE for when the user releases the resize bars.

WM_CHAR for when the user presses 1, 2 or 3. If the user pressed 1 it enabled/disabled multi-sampling. If the user pressed 2 it enabled/disabled text. If the user pressed 3 the mode was switched (solid/wireframe).

WM_DESTROY for when the user exits the window.

The window procedure was declared in the WindowProcedure.h file and defined in the WindowProcedure.cpp file.

The non-local variables used in the window procedure were declared in the DisplayShapesGlobalVariables.h file.

Initialization Functions

I created a function called BuildShapes to declare and define the properties of the shapes.

I created a function called BuildCamera to set the location of the camera.

I created a function called BuildShaders to load and compile my hlsl shaders, create the input element descriptions of the shaders, and create a root signature that described the constant data my shaders expect.

I created a function called BuildVertexAndIndexList to describe the draw arguments of each shape, store the vertices of each shape in a vertex list, and store the indices of each shape in an index list.

I created a function called BuildVertexAndIndexBuffers to create static vertex and index buffers and store the vertices and indices in the vertex and index list in their respective buffers.

I created a function called BuildConstantBuffers to create dynamic constant buffers. I created a constant buffer for storing constant data related to objects (local to world matrices) and a I created constant buffer to store constant data not related to objects (view and projection matrices).

I created a function BuildPSOs to create the PSOs the program uses.

I created PSOs for rendering the shapes in solid or wireframe mode.

I created PSOs for rendering the shapes in solid or wireframe mode using multi-sampling.

I created a function called BuildText to create a text object to render the FPS of the scene.

All the initialization functions were declared in the InitFunctions.h file and defined in the InitFunctions.cpp file. They were all under the Init namespace.

The non-local variables used in the initialization functions were declared in the file DisplayShapesGlobalVariable.h.

Message Loop Functions

I created a function called FrameStats that calculated the FPS of the scene and stored the value in FRAMES_PER_SECOND text object.

I created a function called UserInput that polled keyboard and mouse input for the scene's camera. The user can move the camera use wasd or the arrow keys. The user can rotate the camera by pressing down the left click of the mouse and moving the mouse left and right and up and down.

I created a function called Update to update the scene camera view and perspective projection matrices and copy them into the pass constant buffer. The function also rotates each shape on their local y-axis, updates each shape's local to world to matrix and copies the matrix into the object constant buffer.

I created a Draw function that rendered the shapes.

All the message loop functions were declared in the MessageFunctions.h file and defined in the MessageFunctions.cpp file. They were all under the MessageLoop namespace.

The non-local variables used in the message loop functions were declared in the file DisplayShapesGlobalVariable.h.

Main

The main entry point for a windows desktop application:

Created the window used to display the shapes.

Created a RenderScene object that was used to render the shapes.

Called all the initialization functions.

Created a message loop where I called all the MessageLoop functions.

Shaders Used

Vertex Shader:

```
//input struct
struct vertexInput
{
    float3 inputPosition: POSITION;
    float4 color: COLOR;
};

//output struct
struct vertexOutput
{
    float4 outputPosition : SV_POSITION;
    float4 Color : COLOR;
};

//object constant buffer
cbuffer ObjectConstantBuffer : register(b0)
{
    float4x4 localToWorldMatrix;
    float4x4 pad0;
    float4x4 pad1;
    float4x4 pad2;
};

//pass constant buffer
cbuffer PassConstantBuffer : register(b1)
{
    float4x4 viewMatrix;
    float4x4 projectionMatrix;
    float4x4 pad3;
    float4x4 pad4;
};

//main function
vertexOutput vsMain(vertexInput vin)
{
    vertexOutput vout;

    float4 iPostion = float4(vin.inputPosition, 1.0f);

    float4x4 MVP = mul(localToWorldMatrix, mul(viewMatrix, projectionMatrix));

    //Transform to homogenous clip space
    vout.outputPosition = mul(iPostion, MVP);

    vout.Color = vin.color;

    return vout;
}
```

Pixel Shader:

```
//output struct from vertex shader
struct vertexOutput
{
    float4 outputPosition : SV_POSITION;
    float4 Color : COLOR;
};

float4 psMain(vertexOutput vout) : SV_TARGET
{
    //return the color
    return vout.Color;

}
```