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.

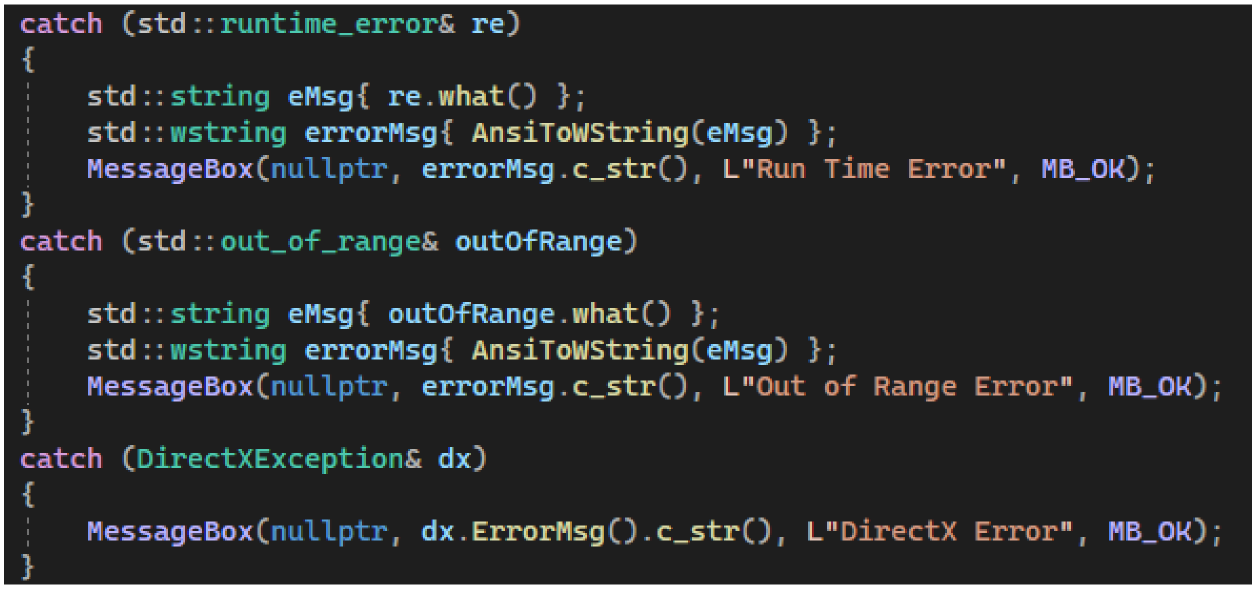
I right clicked Source Files under the Project name tab and went to Add -> Existing Item. Navigated to the directory FA Rendering Engine\Source Files and added all the .cpp files to the project. Also went to the directory FA Shapes\Source Files and all the .cpp files to the project. There are no .lib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



```
catch (std::runtime_error& re)
{
    std::string eMsg{ re.what() };
    std::wstring errorMsg{ AnsiToWString(eMsg) };
    MessageBox(nullptr, errorMsg.c_str(), L"Run Time Error", MB_OK);
}
catch (std::out_of_range& outOfRange)
{
    std::string eMsg{ outOfRange.what() };
    std::wstring errorMsg{ AnsiToWString(eMsg) };
    MessageBox(nullptr, errorMsg.c_str(), L"Out of Range Error", MB_OK);
}
catch (DirectXException& dx)
{
    MessageBox(nullptr, dx.ErrorMsg().c_str(), L"DirectX Error", MB_OK);
}
```

A DirectXExcepetion gets thrown if there Direct3D or DXGI error.

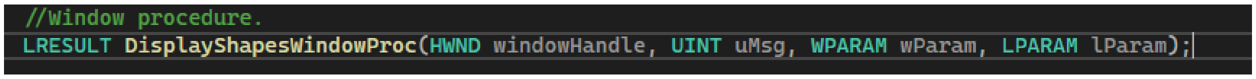
A out_of_range exception gets thrown if an index is out of bound or a key does not have a mapped value.

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.



```
//Window procedure.
LRESULT DisplayShapesWindowProc(HWND windowHandle, UINT uMsg, WPARAM wParam, LPARAM lParam);
```

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).

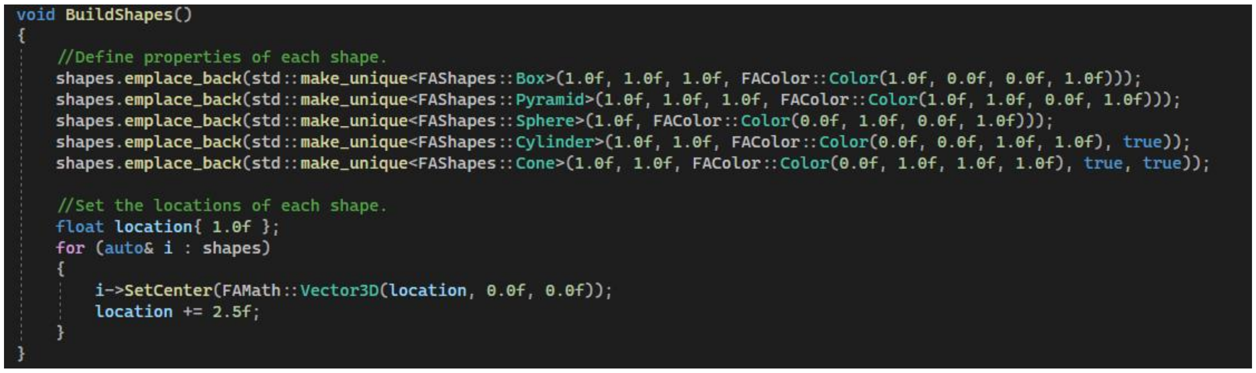
WM_DESTROY for when the user exits the window.

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.

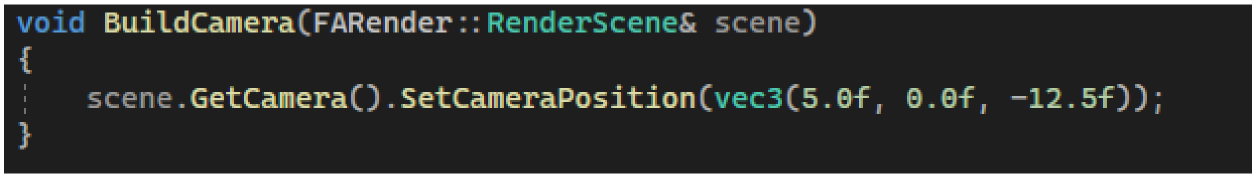
Initialization Functions

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



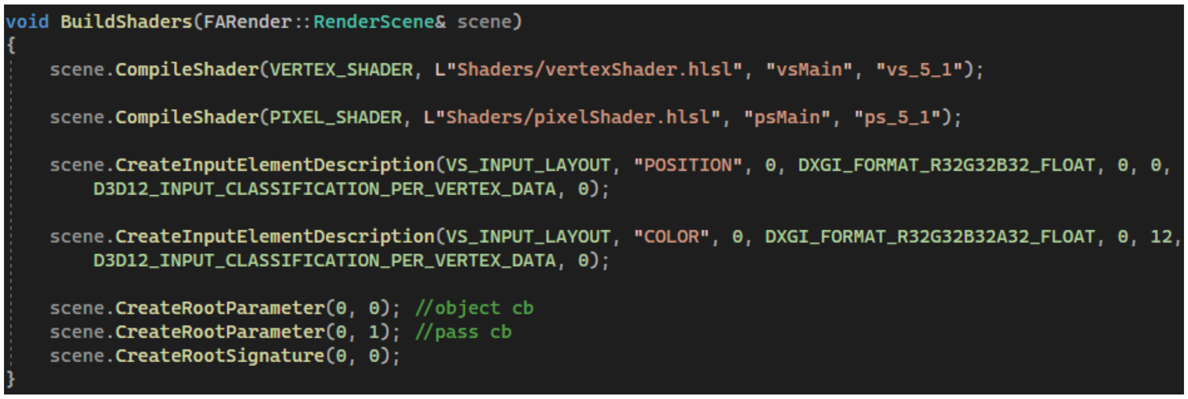
```cpp
void BuildShapes()
{
    //Define properties of each shape.
    shapes.emplace_back(std::make_unique<FAShapes::Box>(1.0f, 1.0f, 1.0f, FAColor::Color(1.0f, 0.0f, 0.0f, 1.0f)));
    shapes.emplace_back(std::make_unique<FAShapes::Pyramid>(1.0f, 1.0f, 1.0f, FAColor::Color(1.0f, 1.0f, 0.0f, 1.0f)));
    shapes.emplace_back(std::make_unique<FAShapes::Sphere>(1.0f, FAColor::Color(0.0f, 1.0f, 0.0f, 1.0f)));
    shapes.emplace_back(std::make_unique<FAShapes::Cylinder>(1.0f, 1.0f, FAColor::Color(0.0f, 0.0f, 1.0f, 1.0f), true));
    shapes.emplace_back(std::make_unique<FAShapes::Cone>(1.0f, 1.0f, FAColor::Color(0.0f, 1.0f, 1.0f, 1.0f), true, true));

    //Set the locations of each shape.
    float location{ 1.0f };
    for (auto& i : shapes)
    {
        i->SetCenter(FAMath::Vector3D(location, 0.0f, 0.0f));
        location += 2.5f;
    }
}
```

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



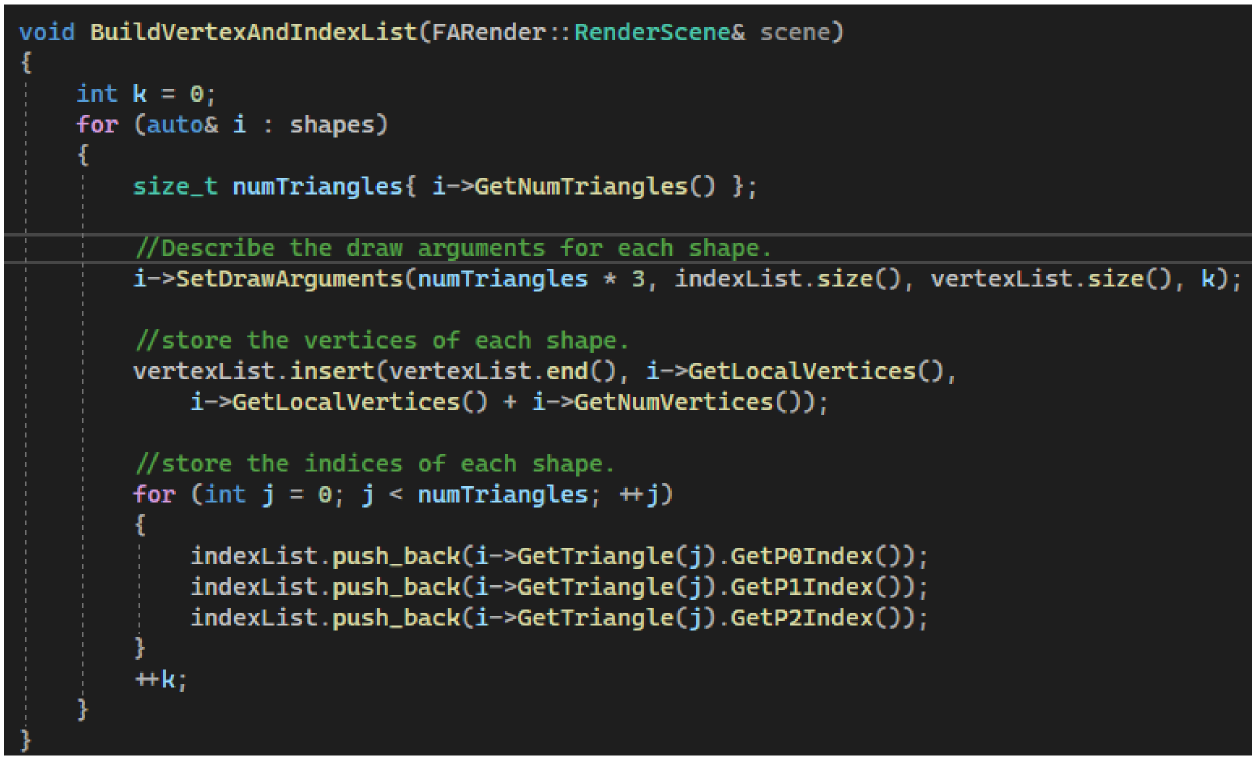
```cpp
void BuildCamera(FARender::RenderScene& scene)
{
    scene.GetCamera().SetCameraPosition(vec3(5.0f, 0.0f, -12.5f));
}
```

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.



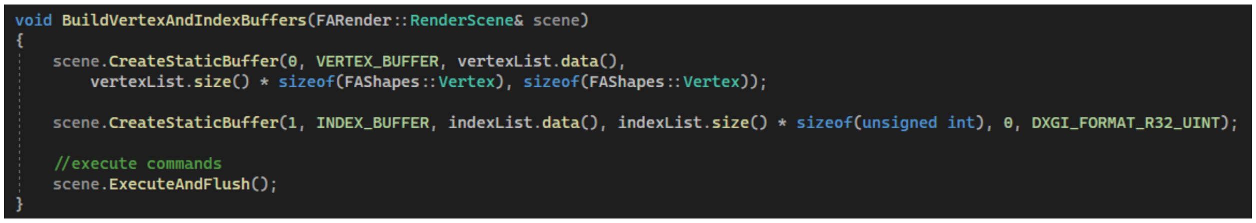
```cpp
void BuildShaders(FARender::RenderScene& scene)
{
    scene.CompileShader(VERTEX_SHADER, L"Shaders/vertexShader.hlsl", "vsMain", "vs_5_1");

    scene.CompileShader(PIXEL_SHADER, L"Shaders/pixelShader.hlsl", "psMain", "ps_5_1");

    scene.CreateInputElementDescription(VS_INPUT_LAYOUT, "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
        D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0);

    scene.CreateInputElementDescription(VS_INPUT_LAYOUT, "COLOR", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0, 12,
        D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0);

    scene.CreateRootParameter(0, 0); //object cb
    scene.CreateRootParameter(0, 1); //pass cb
    scene.CreateRootSignature(0, 0);
}
```

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.



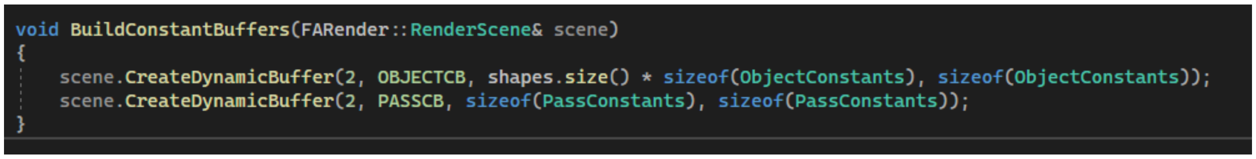
```cpp
void BuildVertexAndIndexList(FARender::RenderScene& scene)
{
    int k = 0;
    for (auto& i : shapes)
    {
        size_t numTriangles{ i->GetNumTriangles() };

        //Describe the draw arguments for each shape.
        i->SetDrawArguments(numTriangles * 3, indexList.size(), vertexList.size(), k);

        //store the vertices of each shape.
        vertexList.insert(vertexList.end(), i->GetLocalVertices(),
            i->GetLocalVertices() + i->GetNumVertices());

        //store the indices of each shape.
        for (int j = 0; j < numTriangles; ++j)
        {
            indexList.push_back(i->GetTriangle(j).GetP0Index());
            indexList.push_back(i->GetTriangle(j).GetP1Index());
            indexList.push_back(i->GetTriangle(j).GetP2Index());
        }
        ++k;
    }
}
```

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.



```cpp
void BuildVertexAndIndexBuffers(FARender::RenderScene& scene)
{
    scene.CreateStaticBuffer(0, VERTEX_BUFFER, vertexList.data(),
        vertexList.size() * sizeof(FAShapes::Vertex), sizeof(FAShapes::Vertex));

    scene.CreateStaticBuffer(1, INDEX_BUFFER, indexList.data(), indexList.size() * sizeof(unsigned int), 0, DXGI_FORMAT_R32_UINT);

    //execute commands
    scene.ExecuteAndFlush();
}
```

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).

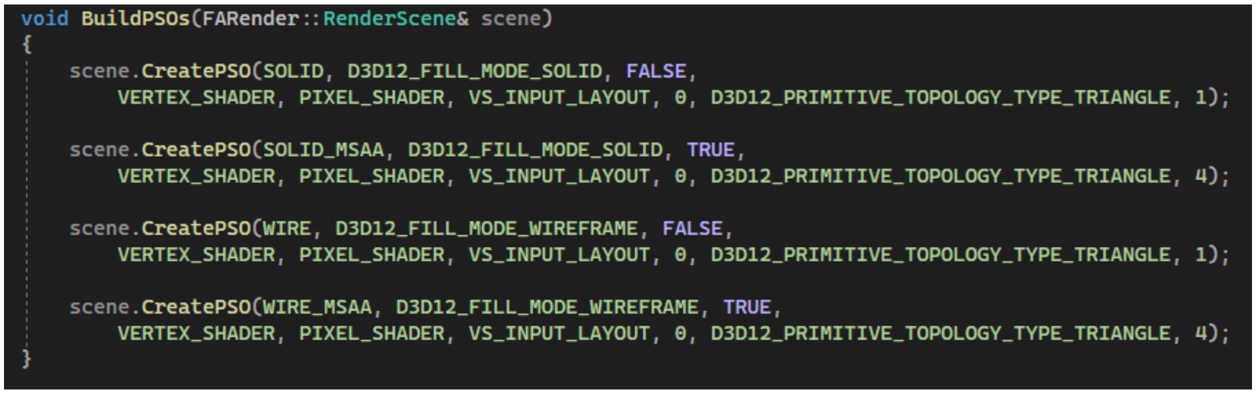


```cpp
void BuildConstantBuffers(FARender::RenderScene& scene)
{
    scene.CreateDynamicBuffer(2, OBJECTCB, shapes.size() * sizeof(ObjectConstants), sizeof(ObjectConstants));
    scene.CreateDynamicBuffer(2, PASSCB, sizeof(PassConstants), sizeof(PassConstants));
}
```

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

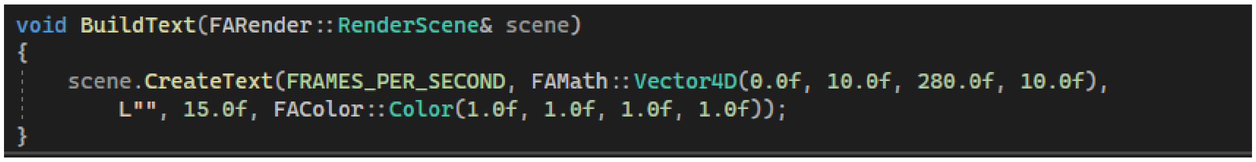
I created PSOs for rendering the shapes in solid or wireframe mode.

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



```cpp
void BuildPSOs(FARender::RenderScene& scene)
{
    scene.CreatePSO(SOLID, D3D12_FILL_MODE_SOLID, FALSE,
        VERTEX_SHADER, PIXEL_SHADER, VS_INPUT_LAYOUT, 0, D3D12_PRIMITIVE_TOPOLOGY_TYPE_TRIANGLE, 1);

    scene.CreatePSO(SOLID_MSAA, D3D12_FILL_MODE_SOLID, TRUE,
        VERTEX_SHADER, PIXEL_SHADER, VS_INPUT_LAYOUT, 0, D3D12_PRIMITIVE_TOPOLOGY_TYPE_TRIANGLE, 4);

    scene.CreatePSO(WIRE, D3D12_FILL_MODE_WIREFRAME, FALSE,
        VERTEX_SHADER, PIXEL_SHADER, VS_INPUT_LAYOUT, 0, D3D12_PRIMITIVE_TOPOLOGY_TYPE_TRIANGLE, 1);

    scene.CreatePSO(WIRE_MSAA, D3D12_FILL_MODE_WIREFRAME, TRUE,
        VERTEX_SHADER, PIXEL_SHADER, VS_INPUT_LAYOUT, 0, D3D12_PRIMITIVE_TOPOLOGY_TYPE_TRIANGLE, 4);
}
```

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



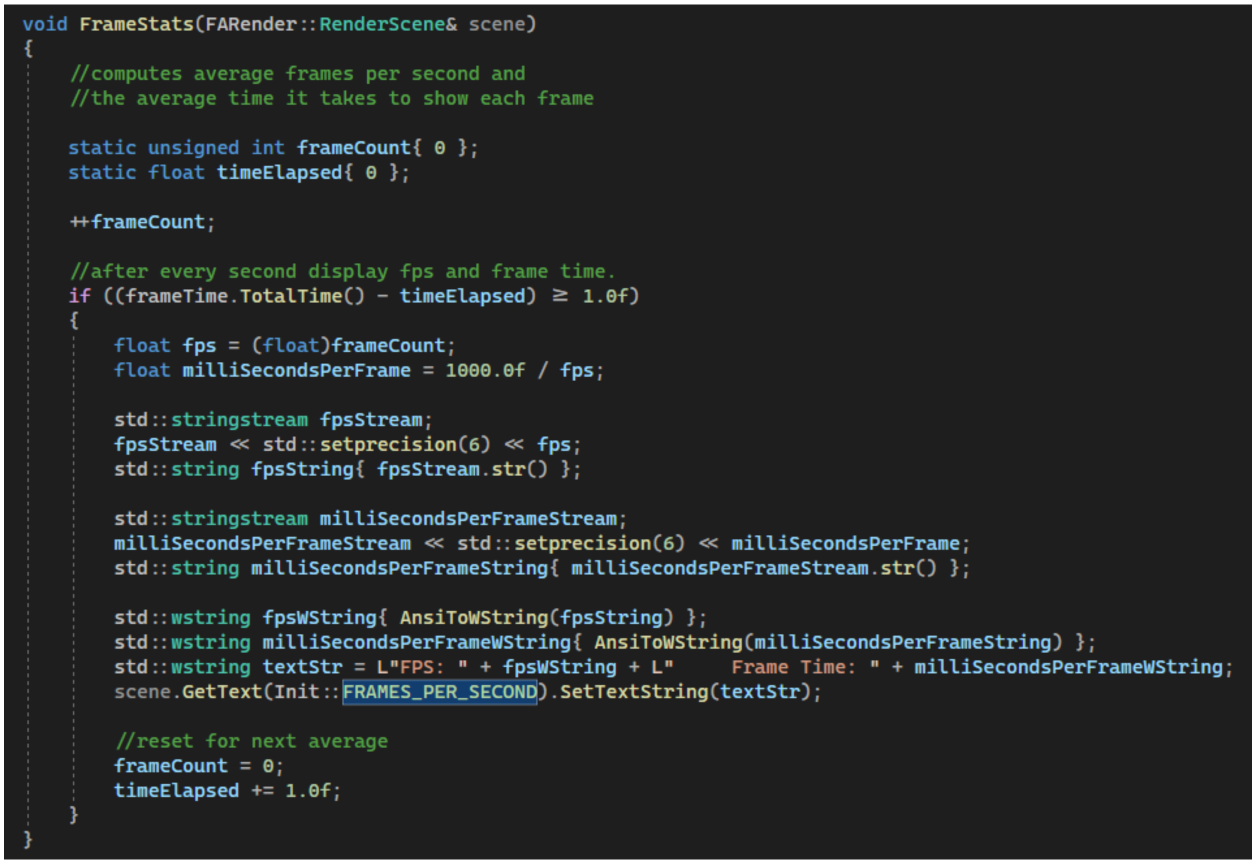
```cpp
void BuildText(FARender::RenderScene& scene)
{
    scene.CreateText(FRAMES_PER_SECOND, FAMath::Vector4D(0.0f, 10.0f, 280.0f, 10.0f),
        L"", 15.0f, FAColor::Color(1.0f, 1.0f, 1.0f, 1.0f));
}
```

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.

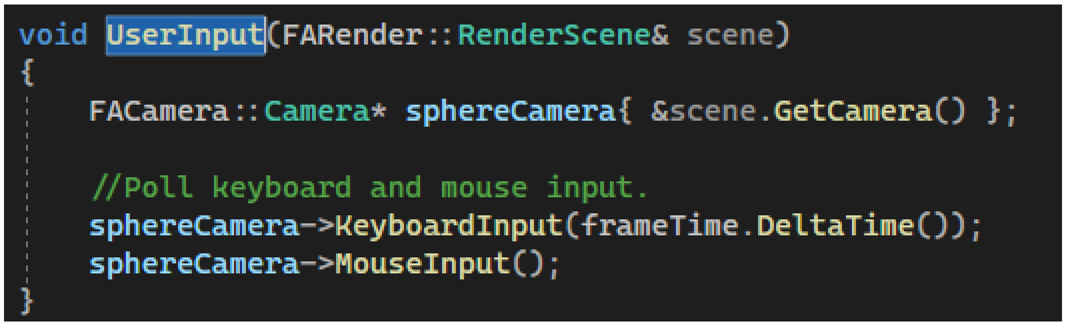
Message Loop Functions

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



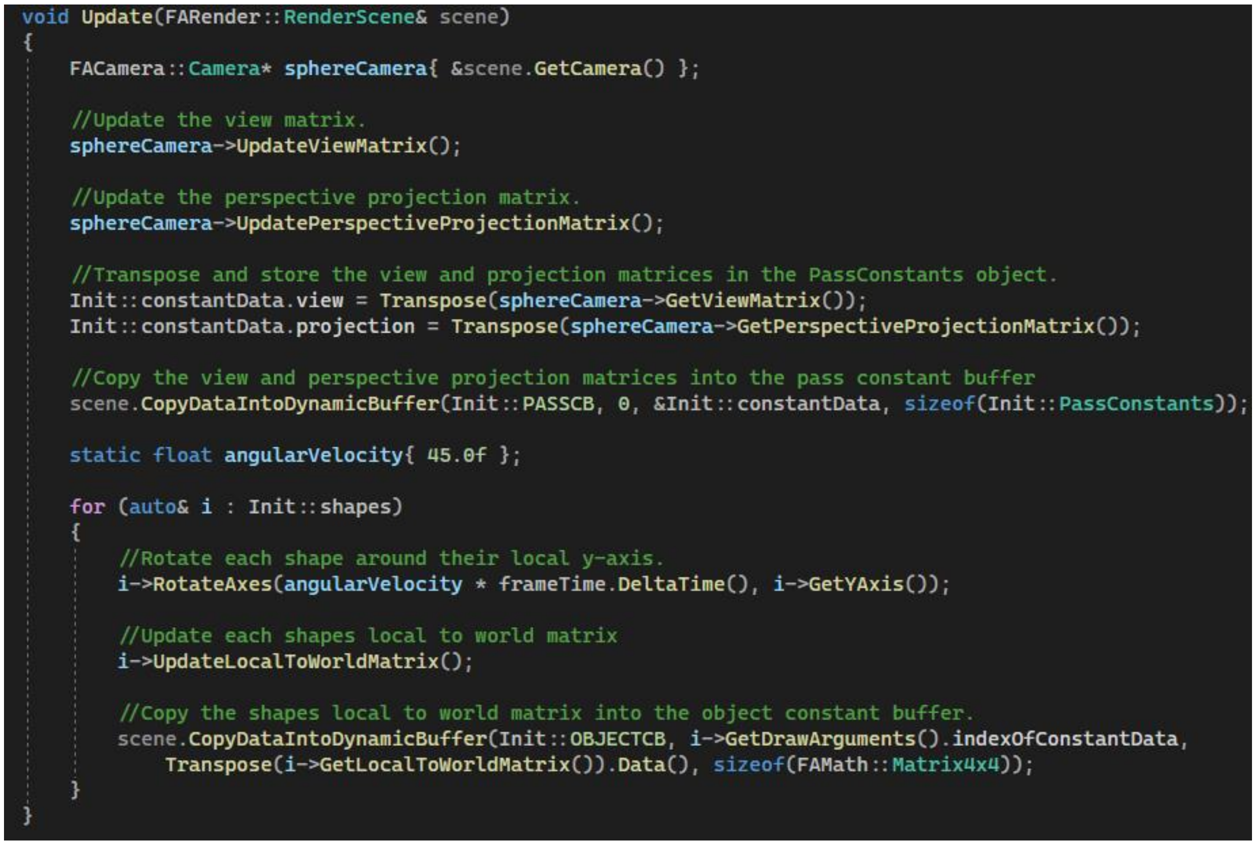
```cpp
void FrameStats(FARender::RenderScene& scene)
{
    //computes average frames per second and
    //the average time it takes to show each frame

    static unsigned int frameCount{ 0 };
    static float timeElapsed{ 0 };

    ++frameCount;

    //after every second display fps and frame time.
    if ((frameTime.TotalTime() - timeElapsed) >= 1.0f)
    {
        float fps = (float)frameCount;
        float milliSecondsPerFrame = 1000.0f / fps;

        std::stringstream fpsStream;
        fpsStream << std::setprecision(6) << fps;
        std::string fpsString{ fpsStream.str() };

        std::stringstream milliSecondsPerFrameStream;
        milliSecondsPerFrameStream << std::setprecision(6) << milliSecondsPerFrame;
        std::string milliSecondsPerFrameString{ milliSecondsPerFrameStream.str() };

        std::wstring fpsWString{ AnsiToWString(fpsString) };
        std::wstring milliSecondsPerFrameWString{ AnsiToWString(milliSecondsPerFrameString) };
        std::wstring textStr = L"FPS: " + fpsWString + L"     Frame Time: " + milliSecondsPerFrameWString;
        scene.GetText(Init::FRAMES_PER_SECOND).SetTextString(textStr);

        //reset for next average
        frameCount = 0;
        timeElapsed += 1.0f;
    }
}
```

I created a function called UserInput that polled keyboard and mouse input for the scenes 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 move the mouse left and right and up and down.



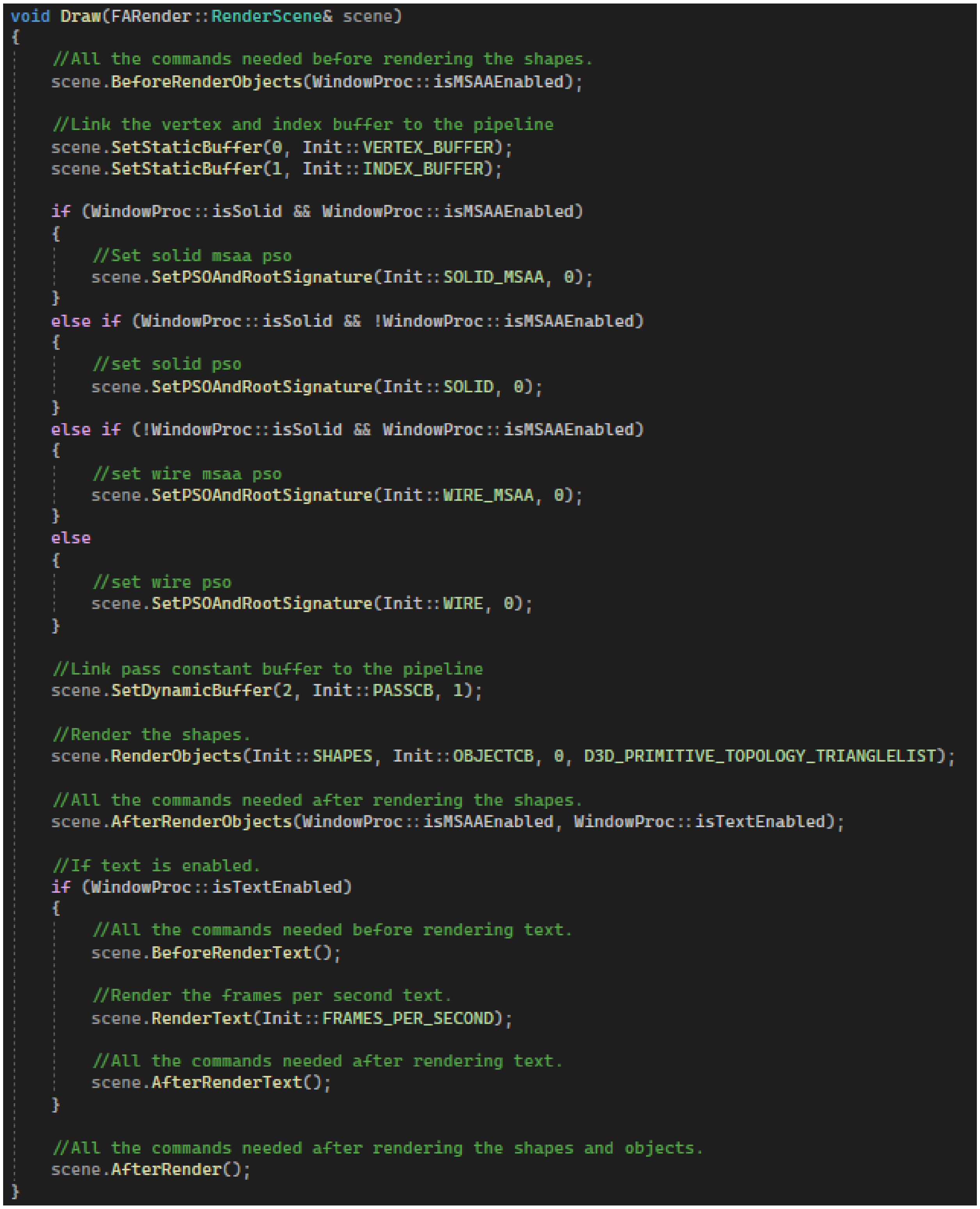
```cpp
void UserInput(FARender::RenderScene& scene)
{
    FACamera::Camera* sphereCamera{ &scene.GetCamera() };

    //Poll keyboard and mouse input.
    sphereCamera->KeyboardInput(frameTime.DeltaTime());
    sphereCamera->MouseInput();
}
```

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.



```
void Update(FARender::RenderScene& scene)
{
    FACamera::Camera* sphereCamera{ &scene.GetCamera() };

    //Update the view matrix.
    sphereCamera->UpdateViewMatrix();

    //Update the perspective projection matrix.
    sphereCamera->UpdatePerspectiveProjectionMatrix();

    //Transpose and store the view and projection matrices in the PassConstants object.
    Init::constantData.view = Transpose(sphereCamera->GetViewMatrix());
    Init::constantData.projection = Transpose(sphereCamera->GetPerspectiveProjectionMatrix());

    //Copy the view and perspective projection matrices into the pass constant buffer
    scene.CopyDataIntoDynamicBuffer(Init::PASSCB, 0, &Init::constantData, sizeof(Init::PassConstants));

    static float angularVelocity{ 45.0f };

    for (auto& i : Init::shapes)
    {
        //Rotate each shape around their local y-axis.
        i->RotateAxes(angularVelocity * frameTime.DeltaTime(), i->GetYAxis());

        //Update each shapes local to world matrix
        i->UpdateLocalToWorldMatrix();

        //Copy the shapes local to world matrix into the object constant buffer.
        scene.CopyDataIntoDynamicBuffer(Init::OBJECTCB, i->GetDrawArguments().indexOfConstantData,
            Transpose(i->GetLocalToWorldMatrix()).Data(), sizeof(FAMath::Matrix4x4));
    }
}
```

I created a Draw function that rendered the shapes.



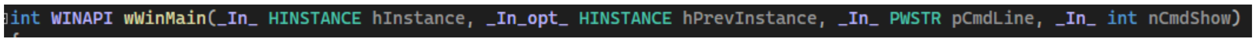
```cpp
void Draw(FARender::RenderScene& scene)
{
    //All the commands needed before rendering the shapes.
    scene.BeforeRenderObjects(WindowProc::isMSAAEnabled);

    //Link the vertex and index buffer to the pipeline
    scene.SetStaticBuffer(0, Init::VERTEX_BUFFER);
    scene.SetStaticBuffer(1, Init::INDEX_BUFFER);

    if (WindowProc::isSolid && WindowProc::isMSAAEnabled)
    {
        //Set solid msaa pso
        scene.SetPSOAndRootSignature(Init::SOLID_MSAA, 0);
    }
    else if (WindowProc::isSolid && !WindowProc::isMSAAEnabled)
    {
        //set solid pso
        scene.SetPSOAndRootSignature(Init::SOLID, 0);
    }
    else if (!WindowProc::isSolid && WindowProc::isMSAAEnabled)
    {
        //set wire msaa pso
        scene.SetPSOAndRootSignature(Init::WIRE_MSAA, 0);
    }
    else
    {
        //set wire pso
        scene.SetPSOAndRootSignature(Init::WIRE, 0);
    }

    //Link pass constant buffer to the pipeline
    scene.SetDynamicBuffer(2, Init::PASSCB, 1);

    //Render the shapes.
    scene.RenderObjects(Init::SHAPES, Init::OBJECTCB, 0, D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST);

    //All the commands needed after rendering the shapes.
    scene.AfterRenderObjects(WindowProc::isMSAAEnabled, WindowProc::isTextEnabled);

    //If text is enabled.
    if (WindowProc::isTextEnabled)
    {
        //All the commands needed before rendering text.
        scene.BeforeRenderText();

        //Render the frames per second text.
        scene.RenderText(Init::FRAMES_PER_SECOND);

        //All the commands needed after rendering text.
        scene.AfterRenderText();
    }

    //All the commands needed after rendering the shapes and objects.
    scene.AfterRender();
}
```

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.

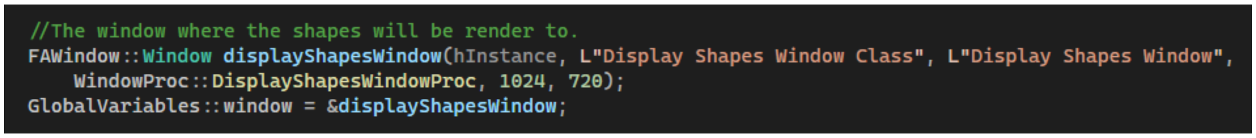
Main

The main entry point for a windows desktop application:



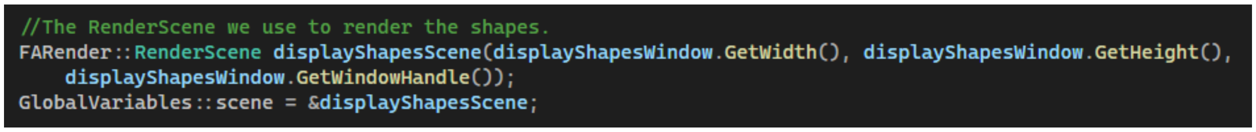
```
int WINAPI wWinMain(_In_ HINSTANCE hInstance, _In_opt_ HINSTANCE hPrevInstance, _In_ PWSTR pCmdLine, _In_ int nCmdShow)
```

Created the window used to render the shapes to.



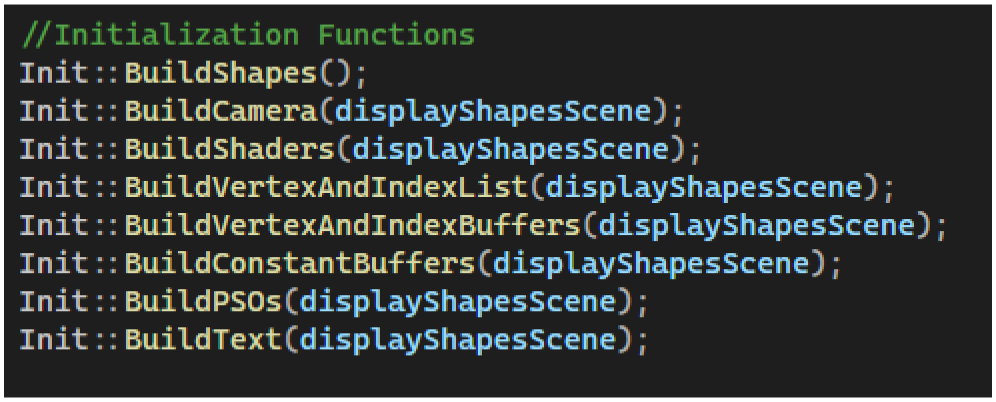
```
//The window where the shapes will be render to.
FAWindow::Window displayShapesWindow(hInstance, L"Display Shapes Window Class", L"Display Shapes Window",
    WindowProc::DisplayShapesWindowProc, 1024, 720);
GlobalVariables::window = &displayShapesWindow;
```

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



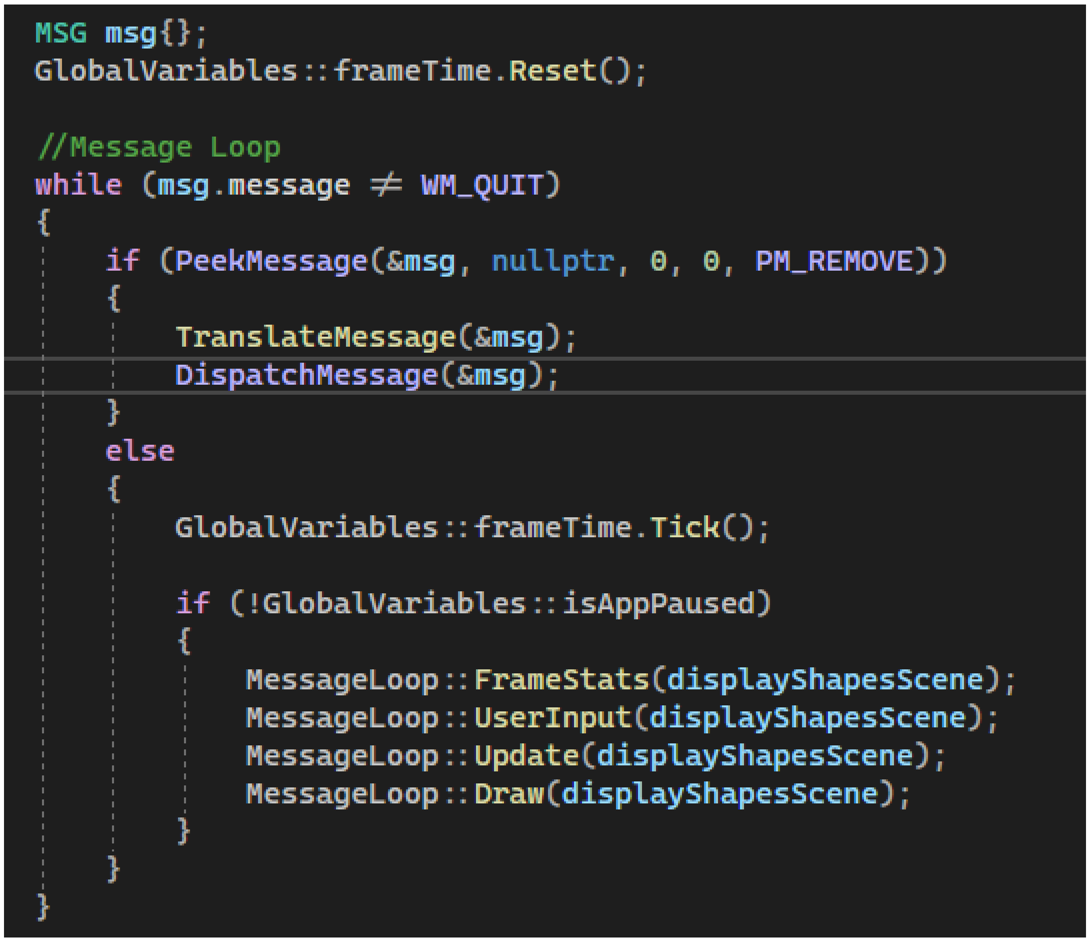
```
//The RenderScene we use to render the shapes.
FARender::RenderScene displayShapesScene(displayShapesWindow.GetWidth(), displayShapesWindow.GetHeight(),
    displayShapesWindow.GetWindowHandle());
GlobalVariables::scene = &displayShapesScene;
```

Called all the initialization functions.



```
//Initialization Functions
Init::BuildShapes();
Init::BuildCamera(displayShapesScene);
Init::BuildShaders(displayShapesScene);
Init::BuildVertexAndIndexList(displayShapesScene);
Init::BuildVertexAndIndexBuffers(displayShapesScene);
Init::BuildConstantBuffers(displayShapesScene);
Init::BuildPSOs(displayShapesScene);
Init::BuildText(displayShapesScene);
```

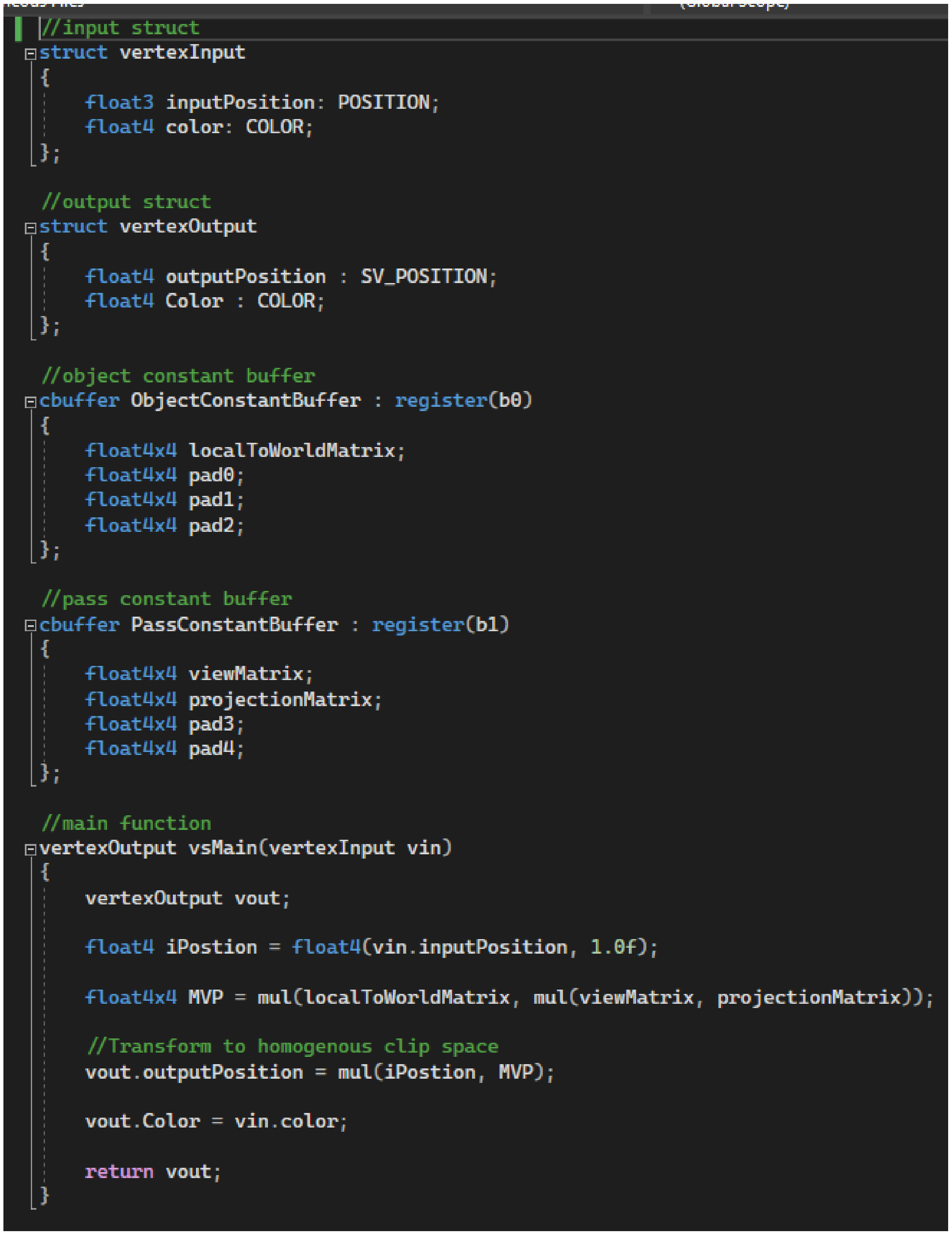
Created a message loop where I called all the MessageLoop functions.



```
MSG msg{};
GlobalVariables::frameTime.Reset();

//Message Loop
while (msg.message ≠ WM_QUIT)
{
    if (PeekMessage(&msg, nullptr, 0, 0, PM_REMOVE))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    else
    {
        GlobalVariables::frameTime.Tick();

        if (!GlobalVariables::isAppPaused)
        {
            MessageLoop::FrameStats(displayShapesScene);
            MessageLoop::UserInput(displayShapesScene);
            MessageLoop::Update(displayShapesScene);
            MessageLoop::Draw(displayShapesScene);
        }
    }
}
```

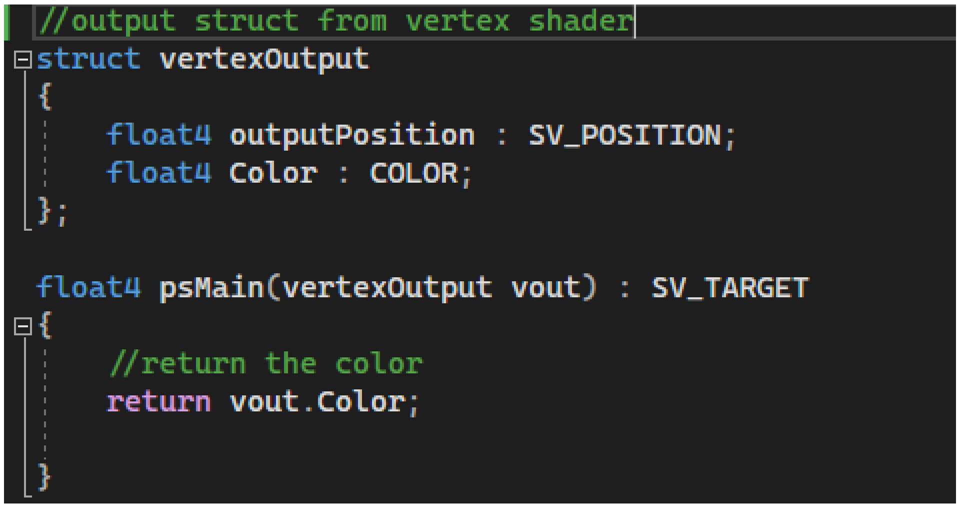
Shaders Used

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;
}
```

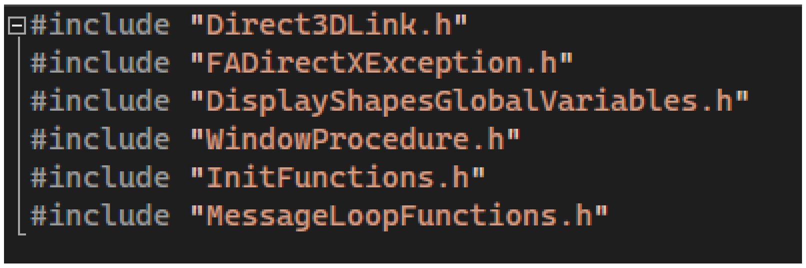
Pixel Shader:



```
//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;

}
```

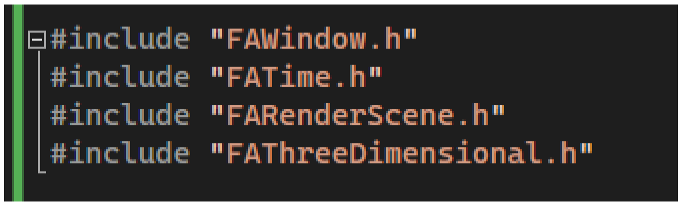
Included Files

In main.cpp:



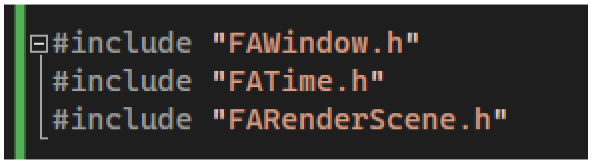
```
#include "Direct3DLink.h"
#include "FADirectXException.h"
#include "DisplayShapesGlobalVariables.h"
#include "WindowProcedure.h"
#include "InitFunctions.h"
#include "MessageLoopFunctions.h"
```

In DisplayShapesGlobalVariables.h



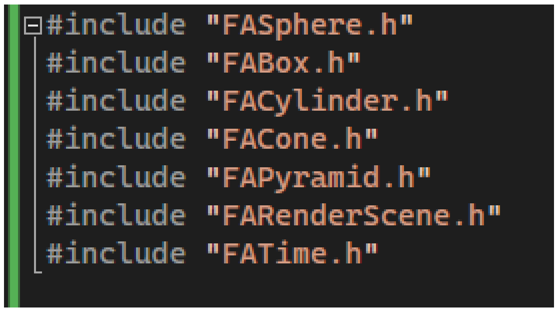
```
#include "FAWindow.h"
#include "FATime.h"
#include "FARenderScene.h"
#include "FAThreeDimensional.h"
```

In WindowProcedure.h:



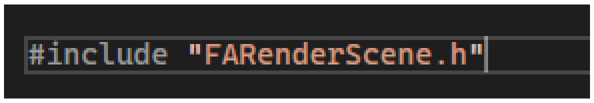
```
#include "FAWindow.h"
#include "FATime.h"
#include "FARenderScene.h"
```

In InitFunctions.h:



```
#include "FASphere.h"
#include "FABox.h"
#include "FACylinder.h"
#include "FACone.h"
#include "FAPyramid.h"
#include "FARenderScene.h"
#include "FATime.h"
```

In MessageLoopFunctions.h



```
#include "FARenderScene.h"
```