**Practical No. 1 Roll No. 13,14**

**Aim : Setup DirectX 11, Window Framework and Initialize Direct3D Device**

**Class-graphicsclass.h**

bool GraphicsClass::Initialize(int screenWidth, int screenHeight, HWND hwnd)

{ return true;}

void GraphicsClass::Shutdown()

{ return;}

bool GraphicsClass::Frame()

{ return true;}

bool GraphicsClass::Render()

{ return true;}

**Class-inputclass.h**

void InputClass::Initialize()

{ int i;

for(i=0; i<256; i++)

{m\_keys[i] = false;}

return;

}

void InputClass::KeyDown(unsigned int input)

{

m\_keys[input] = true;

return;

}

void InputClass::KeyUp(unsigned int input)

{

m\_keys[input] = false;

return;

}

bool InputClass::IsKeyDown(unsigned int key)

{return m\_keys[key];}

**Class-systemclass.h**

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, PSTR pScmdline, int iCmdshow)

{

SystemClass\* System;

bool result;

System = new SystemClass;

if(!System)

{

return 0;

}

result = System->Initialize();

if(result)

{

System->Run();

}

System->Shutdown();

delete System;

System = 0;

return 0;

}

**Class-systemclass.h**

SystemClass::SystemClass()

{

m\_Input = 0;

m\_Graphics = 0;

}

bool SystemClass::Initialize()

{

int screenWidth, screenHeight;

bool result;

screenWidth = 0;

screenHeight = 0;

InitializeWindows(screenWidth, screenHeight);

m\_Input = new InputClass;

if(!m\_Input)

{return false;}

m\_Input->Initialize();

m\_Graphics = new GraphicsClass;

if(!m\_Graphics)

{return false;}

result = m\_Graphics->Initialize(screenWidth, screenHeight, m\_hwnd);

if(!result)

{return false;}

return true;}

void SystemClass::Shutdown()

{

if(m\_Graphics)

{ m\_Graphics->Shutdown();

delete m\_Graphics;

m\_Graphics = 0;

}

if(m\_Input)

{

delete m\_Input;

m\_Input = 0;

}

ShutdownWindows();

return;}

void SystemClass::Run()

{

MSG msg;

bool done, result;

ZeroMemory(&msg, sizeof(MSG));

done = false;

while(!done)

{

if(PeekMessage(&msg, NULL, 0, 0, PM\_REMOVE))

{

TranslateMessage(&msg);

DispatchMessage(&msg);

}

if(msg.message == WM\_QUIT)

{

done = true;

}

else

{

result = Frame();

if(!result)

{

done = true;

}}}return;}

bool SystemClass::Frame()

{

bool result;

if(m\_Input->IsKeyDown(VK\_ESCAPE))

{

return false;

}

result = m\_Graphics->Frame();

if(!result)

{

return false;

}

return true;}

LRESULT CALLBACK SystemClass::MessageHandler(HWND hwnd, UINT umsg, WPARAM wparam, LPARAM lparam)

{

switch(umsg)

{ case WM\_KEYDOWN:

{ m\_Input->KeyDown((unsigned int)wparam);

return 0;}

case WM\_KEYUP:

{

m\_Input->KeyUp((unsigned int)wparam);

return 0;

}

default:

{

return DefWindowProc(hwnd, umsg, wparam, lparam);

}}}

void SystemClass::InitializeWindows(int& screenWidth, int& screenHeight)

{

WNDCLASSEX wc;

DEVMODE dmScreenSettings;

int posX, posY;

ApplicationHandle = this;

m\_hinstance = GetModuleHandle(NULL);

m\_applicationName = L"Engine";

wc.style = CS\_HREDRAW | CS\_VREDRAW | CS\_OWNDC;

wc.lpfnWndProc = WndProc;

wc.cbClsExtra = 0;

wc.cbWndExtra = 0;

wc.hInstance = m\_hinstance;

wc.hIcon = LoadIcon(NULL, IDI\_WINLOGO);

wc.hIconSm = wc.hIcon;

wc.hCursor = LoadCursor(NULL, IDC\_ARROW);

wc.hbrBackground = (HBRUSH)GetStockObject(BLACK\_BRUSH);

wc.lpszMenuName = NULL;

wc.lpszClassName = m\_applicationName;

wc.cbSize = sizeof(WNDCLASSEX);

RegisterClassEx(&wc);

screenWidth = GetSystemMetrics(SM\_CXSCREEN);

screenHeight = GetSystemMetrics(SM\_CYSCREEN);

if(FULL\_SCREEN)

{

memset(&dmScreenSettings, 0, sizeof(dmScreenSettings));

dmScreenSettings.dmSize = sizeof(dmScreenSettings);

dmScreenSettings.dmPelsWidth = (unsigned long)screenWidth;

dmScreenSettings.dmPelsHeight = (unsigned long)screenHeight;

dmScreenSettings.dmBitsPerPel = 32;

dmScreenSettings.dmFields = DM\_BITSPERPEL | DM\_PELSWIDTH | DM\_PELSHEIGHT;

ChangeDisplaySettings(&dmScreenSettings, CDS\_FULLSCREEN);

posX = posY = 0;

}

else

{

screenWidth = 800;

screenHeight = 600;

posX = (GetSystemMetrics(SM\_CXSCREEN) - screenWidth) / 2;

posY = (GetSystemMetrics(SM\_CYSCREEN) - screenHeight) / 2;

}

m\_hwnd = CreateWindowEx(WS\_EX\_APPWINDOW, m\_applicationName, m\_applicationName,

WS\_OVERLAPPEDWINDOW,

posX, posY, screenWidth, screenHeight, NULL, NULL, m\_hinstance, NULL);

ShowWindow(m\_hwnd, SW\_SHOW);

SetForegroundWindow(m\_hwnd);

SetFocus(m\_hwnd);

ShowCursor(false);

return;

}

void SystemClass::ShutdownWindows()

{

ShowCursor(true);

if(FULL\_SCREEN)

{

ChangeDisplaySettings(NULL, 0);

}

DestroyWindow(m\_hwnd);

m\_hwnd = NULL;

UnregisterClass(m\_applicationName, m\_hinstance);

m\_hinstance = NULL;

ApplicationHandle = NULL;

return;}

LRESULT CALLBACK WndProc(HWND hwnd, UINT umessage, WPARAM wparam, LPARAM lparam)

{

switch(umessage)

{case WM\_DESTROY:

{ PostQuitMessage(0);

case WM\_CLOSE:

{

PostQuitMessage(0); return 0;}

default:

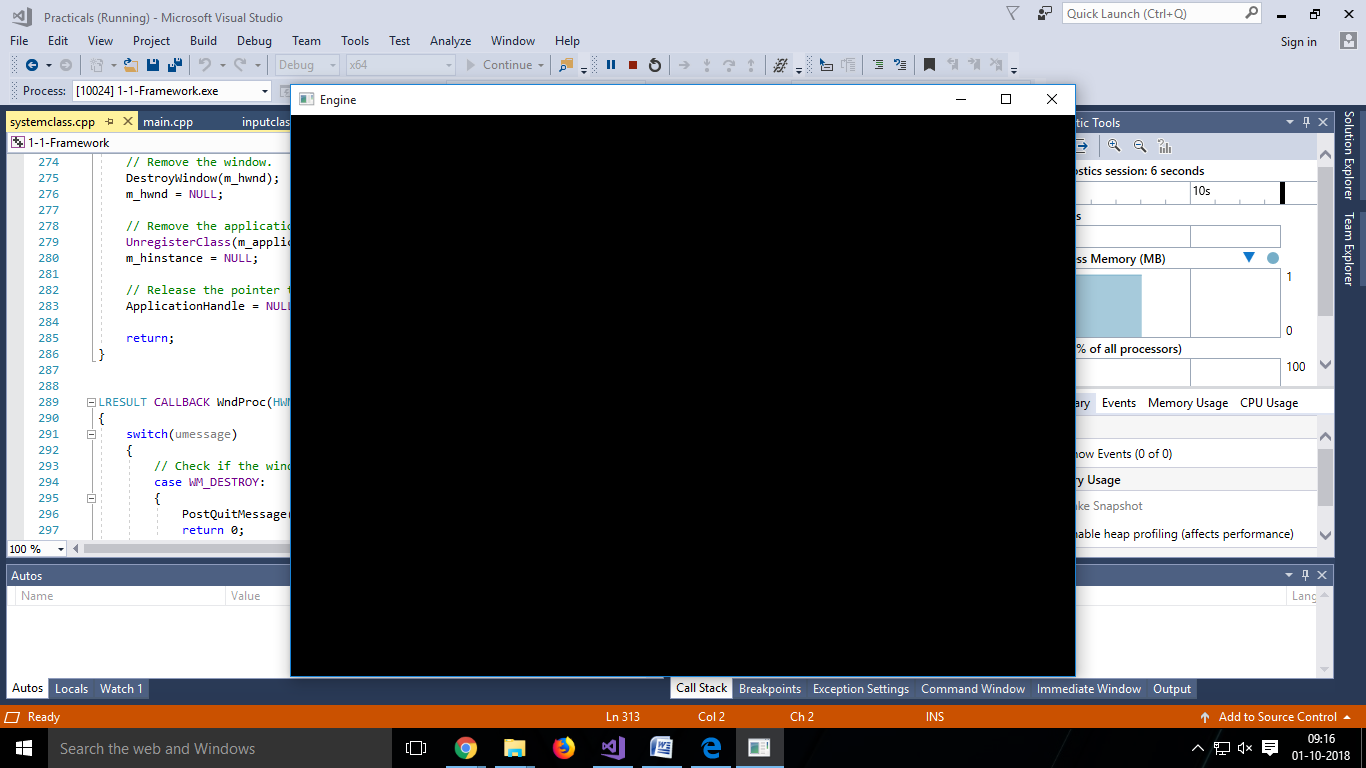
{

return ApplicationHandle->MessageHandler(hwnd, umessage, wparam, lparam);

}

}

}



**Practical No. 2 Roll No. 13,14**

**Aim : Buffers, Shaders and HLSL (Draw a triangle using Direct3D 11)**

**Class -graphicsclass.h**

GraphicsClass::GraphicsClass()

{

m\_Direct3D = 0;

m\_Camera = 0;

m\_Model = 0;

m\_ColorShader = 0;}

bool GraphicsClass::Initialize(int screenWidth, int screenHeight, HWND hwnd)

{

bool result;

m\_Direct3D = new D3DClass;

if(!m\_Direct3D)

{

return false;}

result = m\_Direct3D->Initialize(screenWidth, screenHeight, VSYNC\_ENABLED, hwnd, FULL\_SCREEN, SCREEN\_DEPTH, SCREEN\_NEAR);

if(!result)

{

MessageBox(hwnd, L"Could not initialize Direct3D.", L"Error", MB\_OK);

return false;}

m\_Camera = new CameraClass;

if (!m\_Camera)

{

return false;}

m\_Camera->SetPosition(0.0f, 0.0f, -5.0f);

m\_Model = new ModelClass;

if (!m\_Model)

{

return false;}

result = m\_Model->Initialize(m\_Direct3D->GetDevice());

if (!result)

{

MessageBox(hwnd, L"Could not initialize the model object.", L"Error", MB\_OK);

return false;}

m\_ColorShader = new ColorShaderClass;

if (!m\_ColorShader)

{

return false;}

result = m\_ColorShader->Initialize(m\_Direct3D->GetDevice(), hwnd);

if (!result)

{ MessageBox(hwnd, L"Could not initialize the color shader object.", L"Error", MB\_OK);

return false;}

return true;}

void GraphicsClass::Shutdown()

{ if (m\_ColorShader)

{

m\_ColorShader->Shutdown();

delete m\_ColorShader;

m\_ColorShader = 0;}

if (m\_Model)

{

m\_Model->Shutdown();

delete m\_Model;

m\_Model = 0;}

if (m\_Camera)

{

delete m\_Camera;

m\_Camera = 0;}

if(m\_Direct3D)

{

m\_Direct3D->Shutdown();

delete m\_Direct3D;

m\_Direct3D = 0;}return;}

bool GraphicsClass::Frame()

{ bool result;

result = Render();

if(!result)

{

return false;}

return true;}

bool GraphicsClass::Render()

{

XMMATRIX worldMatrix, viewMatrix, projectionMatrix;

bool result;

static int angle = 0;

static float translation = 0.f;

m\_Direct3D->BeginScene(0.0f, 0.0f, 0.0f, 1.0f);

m\_Camera->Render();

angle = angle++ % 360;

translation += 0.01f;

worldMatrix = XMMatrixRotationY( XMConvertToRadians(angle) );

m\_Camera->GetViewMatrix(viewMatrix);

m\_Direct3D->GetProjectionMatrix(projectionMatrix);

m\_Model->Render(m\_Direct3D->GetDeviceContext());

result = m\_ColorShader->Render(m\_Direct3D->GetDeviceContext(), m\_Model->GetIndexCount(), worldMatrix, viewMatrix, projectionMatrix);

if (!result)

{

return false;}

m\_Direct3D->EndScene();

return true;}

**Class-cameraclass.h**

CameraClass::CameraClass()

{

m\_positionX = 0.0f;

m\_positionY = 0.0f;

m\_positionZ = 0.0f;

m\_rotationX = 0.0f;

m\_rotationY = 0.0f;

m\_rotationZ = 0.0f;}

void CameraClass::SetPosition(float x, float y, float z)

{

m\_positionX = x;

m\_positionY = y;

m\_positionZ = z;

return;}

void CameraClass::SetRotation(float x, float y, float z)

{

m\_rotationX = x;

m\_rotationY = y;

m\_rotationZ = z;

return;}

XMFLOAT3 CameraClass::GetPosition()

{

return XMFLOAT3(m\_positionX, m\_positionY, m\_positionZ);}

XMFLOAT3 CameraClass::GetRotation()

{

return XMFLOAT3(m\_rotationX, m\_rotationY, m\_rotationZ);}

void CameraClass::Render()

{

XMFLOAT3 up, position, lookAt;

XMVECTOR upVector, positionVector, lookAtVector;

float yaw, pitch, roll;

XMMATRIX rotationMatrix;

up.x = 0.0f;

up.y = 1.0f;

up.z = 0.0f;

upVector = XMLoadFloat3(&up);

position.x = m\_positionX;

position.y = m\_positionY;

position.z = m\_positionZ;

positionVector = XMLoadFloat3(&position);

lookAt.x = 0.0f;

lookAt.y = 0.0f;

lookAt.z = 1.0f;

lookAtVector = XMLoadFloat3(&lookAt);

pitch = m\_rotationX \* 0.0174532925f;

yaw = m\_rotationY \* 0.0174532925f;

roll = m\_rotationZ \* 0.0174532925f;

rotationMatrix = XMMatrixRotationRollPitchYaw(pitch, yaw, roll);

lookAtVector = XMVector3TransformCoord(lookAtVector, rotationMatrix);

upVector = XMVector3TransformCoord(upVector, rotationMatrix);

lookAtVector = XMVectorAdd(positionVector, lookAtVector);

m\_viewMatrix = XMMatrixLookAtLH(positionVector, lookAtVector, upVector);

return;}

void CameraClass::GetViewMatrix(XMMATRIX& viewMatrix)

{

viewMatrix = m\_viewMatrix;

return;}

**Class- colorshaderclass.h**

ColorShaderClass::ColorShaderClass()

{

m\_vertexShader = 0;

m\_pixelShader = 0;

m\_layout = 0;

m\_matrixBuffer = 0;}

bool ColorShaderClass::Initialize(ID3D11Device\* device, HWND hwnd)

{

bool result;

WCHAR vertexshader[16] = L"color.vs";

WCHAR pixelshader[16] = L"color.ps";

result = InitializeShader(device, hwnd, vertexshader, pixelshader);

if(!result)

{

return false;

}

return true;

}

void ColorShaderClass::Shutdown()

{ ShutdownShader();

return;}

bool ColorShaderClass::Render(ID3D11DeviceContext\* deviceContext, int indexCount, XMMATRIX worldMatrix, XMMATRIX viewMatrix,

XMMATRIX projectionMatrix)

{

bool result;

result = SetShaderParameters(deviceContext, worldMatrix, viewMatrix, projectionMatrix);

if(!result)

{return false;}

RenderShader(deviceContext, indexCount);

return true;}

bool ColorShaderClass::InitializeShader(ID3D11Device\* device, HWND hwnd, WCHAR\* vsFilename, WCHAR\* psFilename)

{

HRESULT result;

ID3D10Blob\* errorMessage;

ID3D10Blob\* vertexShaderBuffer;

ID3D10Blob\* pixelShaderBuffer;

D3D11\_INPUT\_ELEMENT\_DESC polygonLayout[2];

unsigned int numElements;

D3D11\_BUFFER\_DESC matrixBufferDesc;

errorMessage = 0;

vertexShaderBuffer = 0;

pixelShaderBuffer = 0;

result = D3DCompileFromFile(vsFilename, NULL, NULL, "ColorVertexShader", "vs\_5\_0", D3D10\_SHADER\_ENABLE\_STRICTNESS, 0,

&vertexShaderBuffer, &errorMessage);

if(FAILED(result))

{if(errorMessage)

{

OutputShaderErrorMessage(errorMessage, hwnd, vsFilename);

}

else

{

MessageBox(hwnd, vsFilename, L"Missing Shader File", MB\_OK);}return false;}

result = D3DCompileFromFile(psFilename, NULL, NULL, "ColorPixelShader", "ps\_5\_0", D3D10\_SHADER\_ENABLE\_STRICTNESS, 0,

&pixelShaderBuffer, &errorMessage);

if(FAILED(result))

{

if(errorMessage)

{ OutputShaderErrorMessage(errorMessage, hwnd, psFilename);}

else

{

MessageBox(hwnd, psFilename, L"Missing Shader File", MB\_OK);}

return false;}

result = device->CreateVertexShader(vertexShaderBuffer->GetBufferPointer(), vertexShaderBuffer->GetBufferSize(), NULL, &m\_vertexShader);

if(FAILED(result))

{return false;}

result = device->CreatePixelShader(pixelShaderBuffer->GetBufferPointer(), pixelShaderBuffer->GetBufferSize(), NULL, &m\_pixelShader);

if(FAILED(result))

{return false;}

polygonLayout[0].SemanticName = "POSITION";

polygonLayout[0].SemanticIndex = 0;

polygonLayout[0].Format = DXGI\_FORMAT\_R32G32B32\_FLOAT;

polygonLayout[0].InputSlot = 0;

polygonLayout[0].AlignedByteOffset = 0;

polygonLayout[0].InputSlotClass = D3D11\_INPUT\_PER\_VERTEX\_DATA;

polygonLayout[0].InstanceDataStepRate = 0;

polygonLayout[1].SemanticName = "COLOR";

polygonLayout[1].SemanticIndex = 0;

polygonLayout[1].Format = DXGI\_FORMAT\_R32G32B32A32\_FLOAT;

polygonLayout[1].InputSlot = 0;

polygonLayout[1].AlignedByteOffset = D3D11\_APPEND\_ALIGNED\_ELEMENT;

polygonLayout[1].InputSlotClass = D3D11\_INPUT\_PER\_VERTEX\_DATA;

polygonLayout[1].InstanceDataStepRate = 0;

numElements = sizeof(polygonLayout) / sizeof(polygonLayout[0]);

result = device->CreateInputLayout(polygonLayout, numElements, vertexShaderBuffer->GetBufferPointer(), vertexShaderBuffer->GetBufferSize(), &m\_layout);

if(FAILED(result))

{

return false;}

vertexShaderBuffer->Release();

vertexShaderBuffer = 0;

pixelShaderBuffer->Release();

pixelShaderBuffer = 0;

matrixBufferDesc.Usage = D3D11\_USAGE\_DYNAMIC;

matrixBufferDesc.ByteWidth = sizeof(MatrixBufferType);

matrixBufferDesc.BindFlags = D3D11\_BIND\_CONSTANT\_BUFFER;

matrixBufferDesc.CPUAccessFlags = D3D11\_CPU\_ACCESS\_WRITE;

matrixBufferDesc.MiscFlags = 0;

matrixBufferDesc.StructureByteStride = 0;

result = device->CreateBuffer(&matrixBufferDesc, NULL, &m\_matrixBuffer);

if(FAILED(result))

{return false;}

return true;}

void ColorShaderClass::ShutdownShader()

{

if(m\_matrixBuffer)

{

m\_matrixBuffer->Release();

m\_matrixBuffer = 0;}

if(m\_layout)

{

m\_layout->Release();

m\_layout = 0;}

if(m\_pixelShader)

{

m\_pixelShader->Release();

m\_pixelShader = 0;}

if(m\_vertexShader)

{

m\_vertexShader->Release();

m\_vertexShader = 0;}

return;}

void ColorShaderClass::OutputShaderErrorMessage(ID3D10Blob\* errorMessage, HWND hwnd, WCHAR\* shaderFilename)

{

char\* compileErrors;

unsigned long long bufferSize, i;

ofstream fout;

compileErrors = (char\*)(errorMessage->GetBufferPointer());

bufferSize = errorMessage->GetBufferSize();

fout.open("shader-error.txt");

for(i=0; i<bufferSize; i++)

{

fout << compileErrors[i];}

fout.close();

errorMessage->Release();

errorMessage = 0;

MessageBox(hwnd, L"Error compiling shader. Check shader-error.txt for message.", shaderFilename, MB\_OK);

Return;}

bool ColorShaderClass::SetShaderParameters(ID3D11DeviceContext\* deviceContext, XMMATRIX worldMatrix, XMMATRIX viewMatrix,

XMMATRIX projectionMatrix)

{

HRESULT result;

D3D11\_MAPPED\_SUBRESOURCE mappedResource;

MatrixBufferType\* dataPtr;

unsigned int bufferNumber;

worldMatrix = XMMatrixTranspose(worldMatrix);

viewMatrix = XMMatrixTranspose(viewMatrix);

projectionMatrix = XMMatrixTranspose(projectionMatrix);

result = deviceContext->Map(m\_matrixBuffer, 0, D3D11\_MAP\_WRITE\_DISCARD, 0, &mappedResource);

if(FAILED(result))

{return false;}

dataPtr = (MatrixBufferType\*)mappedResource.pData; dataPtr->world = worldMatrix;

dataPtr->view = viewMatrix;

dataPtr->projection = projectionMatrix;

deviceContext->Unmap(m\_matrixBuffer, 0);

bufferNumber = 0;

deviceContext->VSSetConstantBuffers(bufferNumber, 1, &m\_matrixBuffer);

return true;}

void ColorShaderClass::RenderShader(ID3D11DeviceContext\* deviceContext, int indexCount)

{ deviceContext->IASetInputLayout(m\_layout);

deviceContext->VSSetShader(m\_vertexShader, NULL, 0);

deviceContext->PSSetShader(m\_pixelShader, NULL, 0);

deviceContext->DrawIndexed(indexCount, 0, 0);

return;}

**Class- modelclass.h**

ModelClass::ModelClass()

{

m\_vertexBuffer = 0;

m\_indexBuffer = 0;}

bool ModelClass::Initialize(ID3D11Device\* device)

{

bool result;

result = InitializeBuffers(device);

if(!result)

{

return false;}

return true;}

void ModelClass::Shutdown()

{

ShutdownBuffers();

return;}

void ModelClass::Render(ID3D11DeviceContext\* deviceContext)

{

RenderBuffers(deviceContext);

return;}

int ModelClass::GetIndexCount()

{

return m\_indexCount;}

bool ModelClass::InitializeBuffers(ID3D11Device\* device)

{

VertexType\* vertices;

unsigned long\* indices;

D3D11\_BUFFER\_DESC vertexBufferDesc, indexBufferDesc;

D3D11\_SUBRESOURCE\_DATA vertexData, indexData;

HRESULT result;

m\_vertexCount = 3;

m\_indexCount = 3;

vertices = new VertexType[m\_vertexCount];

if(!vertices)

{

return false;}

indices = new unsigned long[m\_indexCount];

if(!indices)

{

return false;

}

vertices[0].position = XMFLOAT3(-1.0f, -1.0f, 0.0f); vertices[0].color = XMFLOAT4(1.0f, 0.0f, 0.0f, 1.0f);

vertices[1].position = XMFLOAT3(0.0f, 1.0f, 0.0f vertices[1].color = XMFLOAT4(0.0f, 1.0f, 0.0f, 1.0f);

vertices[2].position = XMFLOAT3(1.0f, -1.0f, 0.0f); vertices[2].color = XMFLOAT4(0.0f, 1.0f, 0.0f, 1.0f);

indices[0] = 0;

indices[1] = 1;

indices[2] = 2;

vertexBufferDesc.Usage = D3D11\_USAGE\_DEFAULT;

vertexBufferDesc.ByteWidth = sizeof(VertexType) \* m\_vertexCount;

vertexBufferDesc.BindFlags = D3D11\_BIND\_VERTEX\_BUFFER;

vertexBufferDesc.CPUAccessFlags = 0;

vertexBufferDesc.MiscFlags = 0;

vertexBufferDesc.StructureByteStride = 0;

vertexData.pSysMem = vertices;

vertexData.SysMemPitch = 0;

vertexData.SysMemSlicePitch = 0;

result = device->CreateBuffer(&vertexBufferDesc, &vertexData, &m\_vertexBuffer);

if(FAILED(result))

{return false;}

indexBufferDesc.Usage = D3D11\_USAGE\_DEFAULT;

indexBufferDesc.ByteWidth = sizeof(unsigned long) \* m\_indexCount;

indexBufferDesc.BindFlags = D3D11\_BIND\_INDEX\_BUFFER;

indexBufferDesc.CPUAccessFlags = 0;

indexBufferDesc.MiscFlags = 0;

indexBufferDesc.StructureByteStride = 0;

indexData.pSysMem = indices;

indexData.SysMemPitch = 0;

indexData.SysMemSlicePitch = 0;

result = device->CreateBuffer(&indexBufferDesc, &indexData, &m\_indexBuffer);

if(FAILED(result))

{return false;}

delete [] vertices;

vertices = 0;

delete [] indices;

indices = 0;

return true;

}

void ModelClass::ShutdownBuffers()

{

if(m\_indexBuffer)

{

m\_indexBuffer->Release();

m\_indexBuffer = 0;

}

if(m\_vertexBuffer)

{

m\_vertexBuffer->Release();

m\_vertexBuffer = 0;

}

return;

}

void ModelClass::RenderBuffers(ID3D11DeviceContext\* deviceContext)

{

unsigned int stride;

unsigned int offset;

stride = sizeof(VertexType);

offset = 0;

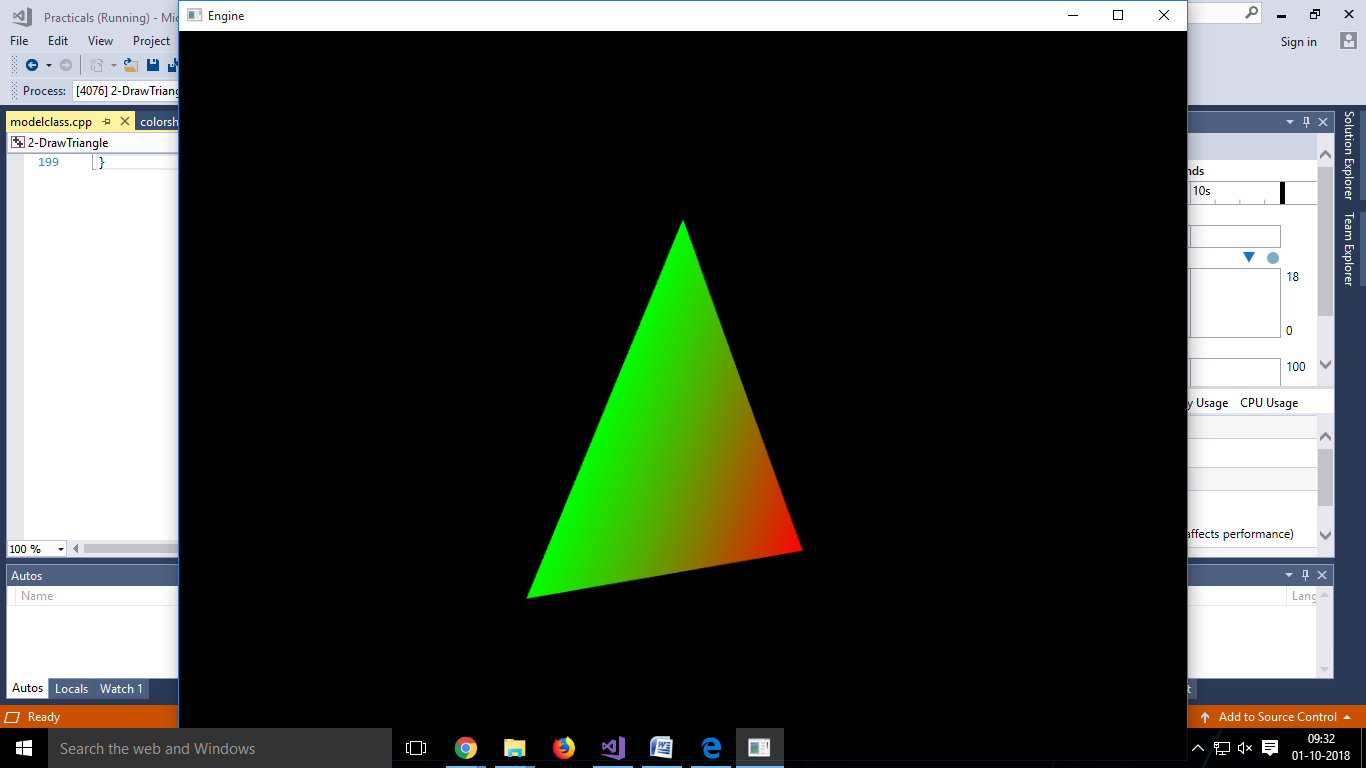
deviceContext->IASetVertexBuffers(0, 1, &m\_vertexBuffer, &stride, &offset);

deviceContext->IASetIndexBuffer(m\_indexBuffer, DXGI\_FORMAT\_R32\_UINT, 0);

deviceContext->IASetPrimitiveTopology(D3D11\_PRIMITIVE\_TOPOLOGY\_TRIANGLELIST);

return;

}



**Practical No. 3 Roll No. 13,14**

**Aim : Texturing (Texture the Triangle using Direct 3D 11)**

**Class- textureclass.h**

TextureClass::TextureClass()

{

m\_targaData = 0;

m\_texture = 0;

m\_textureView = 0}

bool TextureClass::Initialize(ID3D11Device\* device, ID3D11DeviceContext\* deviceContext, char\* filename)

{

bool result;

int height, width;

D3D11\_TEXTURE2D\_DESC textureDesc;

HRESULT hResult;

unsigned int rowPitch;

D3D11\_SHADER\_RESOURCE\_VIEW\_DESC srvDesc;

result = LoadTarga(filename, height, width);

if(!result)

{return false;}

textureDesc.Height = height;

textureDesc.Width = width;

textureDesc.MipLevels = 0;

textureDesc.ArraySize = 1;

textureDesc.Format = DXGI\_FORMAT\_R8G8B8A8\_UNORM;

textureDesc.SampleDesc.Count = 1;

textureDesc.SampleDesc.Quality = 0;

textureDesc.Usage = D3D11\_USAGE\_DEFAULT;

textureDesc.BindFlags = D3D11\_BIND\_SHADER\_RESOURCE | D3D11\_BIND\_RENDER\_TARGET;

textureDesc.CPUAccessFlags = 0;

textureDesc.MiscFlags = D3D11\_RESOURCE\_MISC\_GENERATE\_MIPS;

hResult = device->CreateTexture2D(&textureDesc, NULL, &m\_texture);

if(FAILED(hResult))

{return false;}

rowPitch = (width \* 4) \* sizeof(unsigned char);

deviceContext->UpdateSubresource(m\_texture, 0, NULL, m\_targaData, rowPitch, 0);

srvDesc.Format = textureDesc.Format;

srvDesc.ViewDimension = D3D11\_SRV\_DIMENSION\_TEXTURE2D;

srvDesc.Texture2D.MostDetailedMip = 0;

srvDesc.Texture2D.MipLevels = -1;

hResult = device->CreateShaderResourceView(m\_texture, &srvDesc, &m\_textureView);

if(FAILED(hResult))

{return false;}

deviceContext->GenerateMips(m\_textureView);

delete [] m\_targaData;

m\_targaData = 0;

return true;}

void TextureClass::Shutdown()

{ if(m\_textureView)

{

m\_textureView->Release();

m\_textureView = 0;}

if(m\_texture)

{

m\_texture->Release();

m\_texture = 0;}

if(m\_targaData)

{

delete [] m\_targaData;

m\_targaData = 0;}

return;}

ID3D11ShaderResourceView\* TextureClass::GetTexture()

{return m\_textureView;}

bool TextureClass::LoadTarga(char\* filename, int& height, int& width)

{

int error, bpp, imageSize, index, i, j, k;

FILE\* filePtr;

unsigned int count;

TargaHeader targaFileHeader;

unsigned char\* targaImage;

error = fopen\_s(&filePtr, filename, "rb");

if(error != 0)

{return false;}

count = (unsigned int)fread(&targaFileHeader, sizeof(TargaHeader), 1, filePtr);

if(count != 1)

{return false;}

height = (int)targaFileHeader.height;

width = (int)targaFileHeader.width;

bpp = (int)targaFileHeader.bpp;

if(bpp != 32)

{return false;}

imageSize = width \* height \* 4;

targaImage = new unsigned char[imageSize];

if(!targaImage)

{return false;}

count = (unsigned int)fread(targaImage, 1, imageSize, filePtr);

if(count != imageSize)

{return false;}

error = fclose(filePtr);

if(error != 0)

{return false;}

m\_targaData = new unsigned char[imageSize];

if(!m\_targaData)

{

return false;}

index = 0;

k = (width \* height \* 4) - (width \* 4);

for(j=0; j<height; j++)

{

for(i=0; i<width; i++)

{

m\_targaData[index + 0] = targaImage[k + 2]; // Red.

m\_targaData[index + 1] = targaImage[k + 1]; // Green.

m\_targaData[index + 2] = targaImage[k + 0]; // Blue

m\_targaData[index + 3] = targaImage[k + 3]; // Alpha

k += 4;

index += 4;}

k -= (width \* 8);}

delete [] targaImage;

targaImage = 0;

return true;}

**Class-textureshaderclass.h**

TextureShaderClass::TextureShaderClass()

{

m\_vertexShader = 0;

m\_pixelShader = 0;

m\_layout = 0;

m\_matrixBuffer = 0;

m\_sampleState = 0;}

bool TextureShaderClass::Initialize(ID3D11Device\* device, HWND hwnd)

{

bool result;

WCHAR vertexshader[16] = L"texture.vs";

WCHAR pixelshader[16] = L"texture.ps";

result = InitializeShader(device, hwnd, vertexshader, pixelshader);

if(!result)

{

return false;}

return true;}

void TextureShaderClass::Shutdown()

{

ShutdownShader();

return;}

bool TextureShaderClass::Render(ID3D11DeviceContext\* deviceContext, int indexCount, XMMATRIX worldMatrix, XMMATRIX viewMatrix,

XMMATRIX projectionMatrix, ID3D11ShaderResourceView\* texture)

{

bool result;

result = SetShaderParameters(deviceContext, worldMatrix, viewMatrix, projectionMatrix, texture);

if(!result)

{

return false;}

RenderShader(deviceContext, indexCount);

return true;

}

bool TextureShaderClass::InitializeShader(ID3D11Device\* device, HWND hwnd, WCHAR\* vsFilename, WCHAR\* psFilename)

{

HRESULT result;

ID3D10Blob\* errorMessage;

ID3D10Blob\* vertexShaderBuffer;

ID3D10Blob\* pixelShaderBuffer;

D3D11\_INPUT\_ELEMENT\_DESC polygonLayout[2];

unsigned int numElements;

D3D11\_BUFFER\_DESC matrixBufferDesc;

D3D11\_SAMPLER\_DESC samplerDesc;

errorMessage = 0;

vertexShaderBuffer = 0;

pixelShaderBuffer = 0;

result = D3DCompileFromFile(vsFilename, NULL, NULL, "TextureVertexShader", "vs\_5\_0", D3D10\_SHADER\_ENABLE\_STRICTNESS, 0, &vertexShaderBuffer, &errorMessage);

if(FAILED(result))

{ if(errorMessage)

{

OutputShaderErrorMessage(errorMessage, hwnd, vsFilename);

}

else

{

MessageBox(hwnd, vsFilename, L"Missing Shader File", MB\_OK);}

return false;}

result = D3DCompileFromFile(psFilename, NULL, NULL, "TexturePixelShader", "ps\_5\_0", D3D10\_SHADER\_ENABLE\_STRICTNESS, 0,

&pixelShaderBuffer, &errorMessage);

if(FAILED(result))

{

if(errorMessage)

{

OutputShaderErrorMessage(errorMessage, hwnd, psFilename);}

else

{

MessageBox(hwnd, psFilename, L"Missing Shader File", MB\_OK);}

return false;}

result = device->CreateVertexShader(vertexShaderBuffer->GetBufferPointer(), vertexShaderBuffer->GetBufferSize(), NULL, &m\_vertexShader);

if(FAILED(result))

{

return false;

}

result = device->CreatePixelShader(pixelShaderBuffer->GetBufferPointer(), pixelShaderBuffer->GetBufferSize(), NULL, &m\_pixelShader);

if(FAILED(result))

{

return false;}

polygonLayout[0].SemanticName = "POSITION";

polygonLayout[0].SemanticIndex = 0;

polygonLayout[0].Format = DXGI\_FORMAT\_R32G32B32\_FLOAT;

polygonLayout[0].InputSlot = 0;

polygonLayout[0].AlignedByteOffset = 0;

polygonLayout[0].InputSlotClass = D3D11\_INPUT\_PER\_VERTEX\_DATA;

polygonLayout[0].InstanceDataStepRate = 0;

polygonLayout[1].SemanticName = "TEXCOORD";

polygonLayout[1].SemanticIndex = 0;

polygonLayout[1].Format = DXGI\_FORMAT\_R32G32\_FLOAT;

polygonLayout[1].InputSlot = 0;

polygonLayout[1].AlignedByteOffset = D3D11\_APPEND\_ALIGNED\_ELEMENT;

polygonLayout[1].InputSlotClass = D3D11\_INPUT\_PER\_VERTEX\_DATA;

polygonLayout[1].InstanceDataStepRate = 0;

numElements = sizeof(polygonLayout) / sizeof(polygonLayout[0]);

result = device->CreateInputLayout(polygonLayout, numElements, vertexShaderBuffer->GetBufferPointer(),

vertexShaderBuffer->GetBufferSize(), &m\_layout);

if(FAILED(result))

{

return false;} vertexShaderBuffer->Release();

vertexShaderBuffer = 0;

pixelShaderBuffer->Release();

pixelShaderBuffer = 0;

matrixBufferDesc.Usage = D3D11\_USAGE\_DYNAMIC;

matrixBufferDesc.ByteWidth = sizeof(MatrixBufferType);

matrixBufferDesc.BindFlags = D3D11\_BIND\_CONSTANT\_BUFFER;

matrixBufferDesc.CPUAccessFlags = D3D11\_CPU\_ACCESS\_WRITE;

matrixBufferDesc.MiscFlags = 0;

matrixBufferDesc.StructureByteStride = 0;

result = device->CreateBuffer(&matrixBufferDesc, NULL, &m\_matrixBuffer);

if(FAILED(result))

{

return false;}

samplerDesc.Filter = D3D11\_FILTER\_MIN\_MAG\_MIP\_LINEAR;

samplerDesc.AddressU = D3D11\_TEXTURE\_ADDRESS\_WRAP;

samplerDesc.AddressV = D3D11\_TEXTURE\_ADDRESS\_WRAP;

samplerDesc.AddressW = D3D11\_TEXTURE\_ADDRESS\_WRAP;

samplerDesc.MipLODBias = 0.0f;

samplerDesc.MaxAnisotropy = 1;

samplerDesc.ComparisonFunc = D3D11\_COMPARISON\_ALWAYS;

samplerDesc.BorderColor[0] = 0;

samplerDesc.BorderColor[1] = 0;

samplerDesc.BorderColor[2] = 0;

samplerDesc.BorderColor[3] = 0;

samplerDesc.MinLOD = 0;

samplerDesc.MaxLOD = D3D11\_FLOAT32\_MAX;

result = device->CreateSamplerState(&samplerDesc, &m\_sampleState);

if (FAILED(result))

{

return false;}

return true;}

void TextureShaderClass::ShutdownShader()

{

if (m\_sampleState)

{

m\_sampleState->Release();

m\_sampleState = 0;}

if(m\_matrixBuffer)

{

m\_matrixBuffer->Release();

m\_matrixBuffer = 0;}

if(m\_layout)

{

m\_layout->Release();

m\_layout = 0;}

if(m\_pixelShader)

{

m\_pixelShader->Release();

m\_pixelShader = 0;}

if(m\_vertexShader)

{

m\_vertexShader->Release();

m\_vertexShader = 0;}

return;}

void TextureShaderClass::OutputShaderErrorMessage(ID3D10Blob\* errorMessage, HWND hwnd, WCHAR\* shaderFilename)

{

char\* compileErrors;

unsigned long long bufferSize, i;

ofstream fout;

compileErrors = (char\*)(errorMessage->GetBufferPointer());

bufferSize = errorMessage->GetBufferSize();

fout.open("shader-error.txt");

for(i=0; i<bufferSize; i++)

{

fout << compileErrors[i];

}

fout.close();

errorMessage->Release();

errorMessage = 0;

MessageBox(hwnd, L"Error compiling shader. Check shader-error.txt for message.", shaderFilename, MB\_OK);

return;}

bool TextureShaderClass::SetShaderParameters(ID3D11DeviceContext\* deviceContext, XMMATRIX worldMatrix, XMMATRIX viewMatrix,

XMMATRIX projectionMatrix, ID3D11ShaderResourceView\* texture)

{

HRESULT result;

D3D11\_MAPPED\_SUBRESOURCE mappedResource;

MatrixBufferType\* dataPtr;

unsigned int bufferNumber;

worldMatrix = XMMatrixTranspose(worldMatrix);

viewMatrix = XMMatrixTranspose(viewMatrix);

projectionMatrix = XMMatrixTranspose(projectionMatrix);

result = deviceContext->Map(m\_matrixBuffer, 0, D3D11\_MAP\_WRITE\_DISCARD, 0, &mappedResource);

if(FAILED(result))

{

return false;}

dataPtr = (MatrixBufferType\*)mappedResource.pData;

dataPtr->world = worldMatrix;

dataPtr->view = viewMatrix;

dataPtr->projection = projectionMatrix;

deviceContext->Unmap(m\_matrixBuffer, 0);

bufferNumber = 0;

deviceContext->VSSetConstantBuffers(bufferNumber, 1, &m\_matrixBuffer);

deviceContext->PSSetShaderResources(0, 1, &texture);

return true;

}

void TextureShaderClass::RenderShader(ID3D11DeviceContext\* deviceContext, int indexCount)

{

deviceContext->IASetInputLayout(m\_layout);

deviceContext->VSSetShader(m\_vertexShader, NULL, 0);

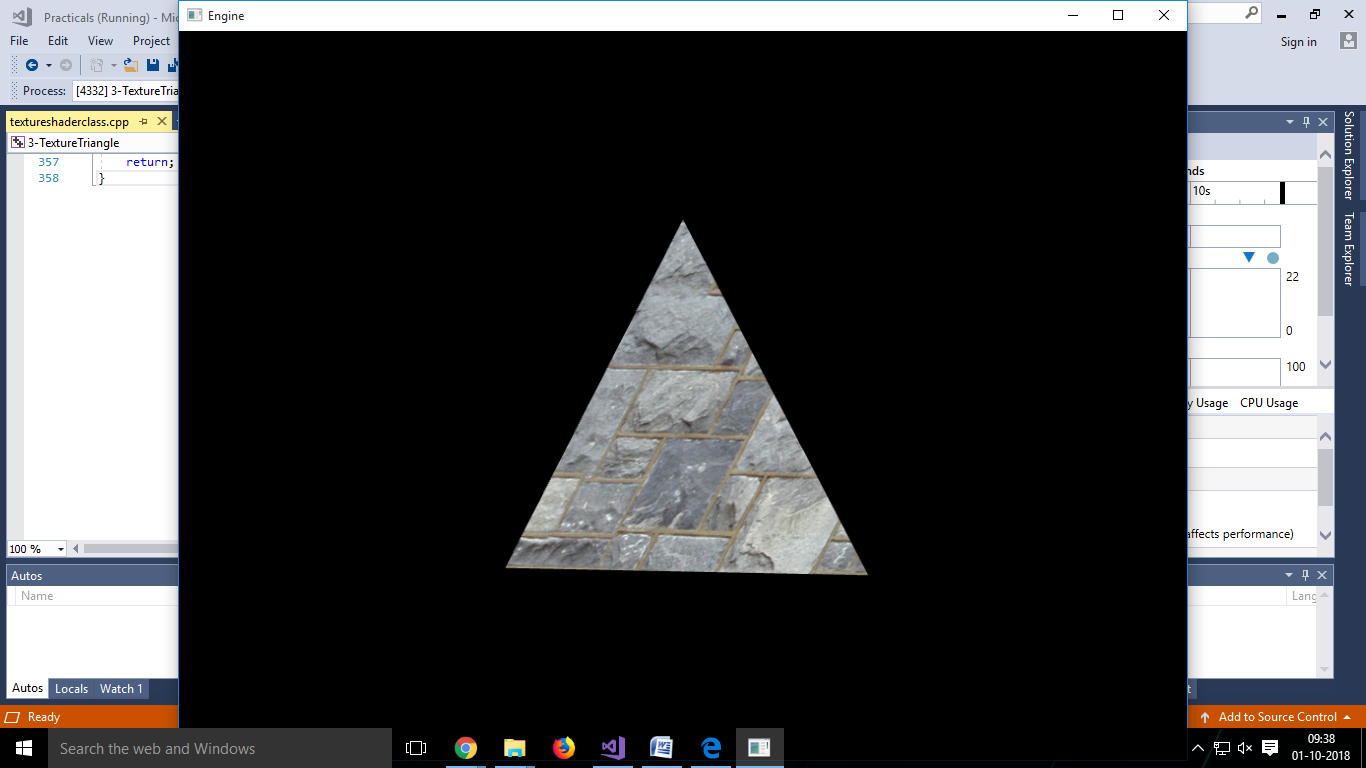
deviceContext->PSSetShader(m\_pixelShader, NULL, 0);

deviceContext->PSSetSamplers(0, 1, &m\_sampleState);

deviceContext->DrawIndexed(indexCount, 0, 0);

return;

}



**Practical No. 4 Roll No. 45,50**

**Aim : Lightning (Programmable Diffuse Lightning using Direct3D 11)**

**Class-lightclass.h**

void LightClass::SetDiffuseColor(float red, float green, float blue, float alpha)

{

m\_diffuseColor = XMFLOAT4(red, green, blue, alpha);

return;}

void LightClass::SetDirection(float x, float y, float z)

{

m\_direction = XMFLOAT3(x, y, z);

return;}

XMFLOAT4 LightClass::GetDiffuseColor()

{

return m\_diffuseColor;}

XMFLOAT3 LightClass::GetDirection()

{

return m\_direction;}

**Class- lightshaderclass.h**

LightShaderClass::LightShaderClass()

{

m\_vertexShader = 0;

m\_pixelShader = 0;

m\_layout = 0;

m\_sampleState = 0;

m\_matrixBuffer = 0;

m\_lightBuffer = 0;

}

bool LightShaderClass::Initialize(ID3D11Device\* device, HWND hwnd)

{

bool result;

WCHAR vertexshader[16] = L"light.vs";

WCHAR pixelshader[16] = L"light.ps";

result = InitializeShader(device, hwnd, vertexshader, pixelshader);

if(!result)

{

return false;}

return true;}

void LightShaderClass::Shutdown()

{

ShutdownShader();

return;}

bool LightShaderClass::Render(ID3D11DeviceContext\* deviceContext, int indexCount, XMMATRIX worldMatrix, XMMATRIX viewMatrix,

XMMATRIX projectionMatrix, ID3D11ShaderResourceView\* texture, XMFLOAT3 lightDirection, XMFLOAT4 diffuseColor)

{

bool result;

result = SetShaderParameters(deviceContext, worldMatrix, viewMatrix, projectionMatrix, texture, lightDirection, diffuseColor);

if(!result)

{

return false;}

RenderShader(deviceContext, indexCount);

return true;}

bool LightShaderClass::InitializeShader(ID3D11Device\* device, HWND hwnd, WCHAR\* vsFilename, WCHAR\* psFilename)

{

HRESULT result;

ID3D10Blob\* errorMessage;

ID3D10Blob\* vertexShaderBuffer;

ID3D10Blob\* pixelShaderBuffer;

D3D11\_INPUT\_ELEMENT\_DESC polygonLayout[3];

unsigned int numElements;

D3D11\_SAMPLER\_DESC samplerDesc;

D3D11\_BUFFER\_DESC matrixBufferDesc;

D3D11\_BUFFER\_DESC lightBufferDesc;

errorMessage = 0;

vertexShaderBuffer = 0;

pixelShaderBuffer = 0;

result = D3DCompileFromFile(vsFilename, NULL, NULL, "LightVertexShader", "vs\_5\_0", D3D10\_SHADER\_ENABLE\_STRICTNESS, 0, &vertexShaderBuffer, &errorMessage);

if(FAILED(result))

{

if(errorMessage)

{

OutputShaderErrorMessage(errorMessage, hwnd, vsFilename);

}

else

{

MessageBox(hwnd, vsFilename, L"Missing Shader File", MB\_OK);}

return false;}

result = D3DCompileFromFile(psFilename, NULL, NULL, "LightPixelShader", "ps\_5\_0", D3D10\_SHADER\_ENABLE\_STRICTNESS, 0,

&pixelShaderBuffer, &errorMessage);

if(FAILED(result))

{

if(errorMessage)

{

OutputShaderErrorMessage(errorMessage, hwnd, psFilename);}

else

{

MessageBox(hwnd, psFilename, L"Missing Shader File", MB\_OK);

}

return false;

}

result = device->CreateVertexShader(vertexShaderBuffer->GetBufferPointer(), vertexShaderBuffer->GetBufferSize(), NULL, &m\_vertexShader);

if(FAILED(result))

{

return false;}

result = device->CreatePixelShader(pixelShaderBuffer->GetBufferPointer(), pixelShaderBuffer->GetBufferSize(), NULL, &m\_pixelShader);

if(FAILED(result))

{

return false;}

polygonLayout[0].SemanticName = "POSITION";

polygonLayout[0].SemanticIndex = 0;

polygonLayout[0].Format = DXGI\_FORMAT\_R32G32B32\_FLOAT;

polygonLayout[0].InputSlot = 0;

polygonLayout[0].AlignedByteOffset = 0;

polygonLayout[0].InputSlotClass = D3D11\_INPUT\_PER\_VERTEX\_DATA;

polygonLayout[0].InstanceDataStepRate = 0;

polygonLayout[1].SemanticName = "TEXCOORD";

polygonLayout[1].SemanticIndex = 0;

polygonLayout[1].Format = DXGI\_FORMAT\_R32G32\_FLOAT;

polygonLayout[1].InputSlot = 0;

polygonLayout[1].AlignedByteOffset = D3D11\_APPEND\_ALIGNED\_ELEMENT;

polygonLayout[1].InputSlotClass = D3D11\_INPUT\_PER\_VERTEX\_DATA;

polygonLayout[1].InstanceDataStepRate = 0;

polygonLayout[2].SemanticName = "NORMAL";

polygonLayout[2].SemanticIndex = 0;

polygonLayout[2].Format = DXGI\_FORMAT\_R32G32B32\_FLOAT;

polygonLayout[2].InputSlot = 0;

polygonLayout[2].AlignedByteOffset = D3D11\_APPEND\_ALIGNED\_ELEMENT;

polygonLayout[2].InputSlotClass = D3D11\_INPUT\_PER\_VERTEX\_DATA;

polygonLayout[2].InstanceDataStepRate = 0;

numElements = sizeof(polygonLayout) / sizeof(polygonLayout[0]);

result = device->CreateInputLayout(polygonLayout, numElements, vertexShaderBuffer->GetBufferPointer(), vertexShaderBuffer->GetBufferSize(),

&m\_layout);

if(FAILED(result))

{

return false;}

vertexShaderBuffer->Release();

vertexShaderBuffer = 0;

pixelShaderBuffer->Release();

pixelShaderBuffer = 0;

samplerDesc.Filter = D3D11\_FILTER\_MIN\_MAG\_MIP\_LINEAR;

samplerDesc.AddressU = D3D11\_TEXTURE\_ADDRESS\_WRAP;

samplerDesc.AddressV = D3D11\_TEXTURE\_ADDRESS\_WRAP;

samplerDesc.AddressW = D3D11\_TEXTURE\_ADDRESS\_WRAP;

samplerDesc.MipLODBias = 0.0f;

samplerDesc.MaxAnisotropy = 1;

samplerDesc.ComparisonFunc = D3D11\_COMPARISON\_ALWAYS;

samplerDesc.BorderColor[0] = 0;

samplerDesc.BorderColor[1] = 0;

samplerDesc.BorderColor[2] = 0;

samplerDesc.BorderColor[3] = 0;

samplerDesc.MinLOD = 0;

samplerDesc.MaxLOD = D3D11\_FLOAT32\_MAX;

result = device->CreateSamplerState(&samplerDesc, &m\_sampleState);

if(FAILED(result))

{

return false;}

matrixBufferDesc.Usage = D3D11\_USAGE\_DYNAMIC;

matrixBufferDesc.ByteWidth = sizeof(MatrixBufferType);

matrixBufferDesc.BindFlags = D3D11\_BIND\_CONSTANT\_BUFFER;

matrixBufferDesc.CPUAccessFlags = D3D11\_CPU\_ACCESS\_WRITE;

matrixBufferDesc.MiscFlags = 0;

matrixBufferDesc.StructureByteStride = 0;

result = device->CreateBuffer(&matrixBufferDesc, NULL, &m\_matrixBuffer);

if(FAILED(result))

{

return false;}

D3D11\_BIND\_CONSTANT\_BUFFER or CreateBuffer will fail.

lightBufferDesc.Usage = D3D11\_USAGE\_DYNAMIC;

lightBufferDesc.ByteWidth = sizeof(LightBufferType);

lightBufferDesc.BindFlags = D3D11\_BIND\_CONSTANT\_BUFFER;

lightBufferDesc.CPUAccessFlags = D3D11\_CPU\_ACCESS\_WRITE;

lightBufferDesc.MiscFlags = 0;

lightBufferDesc.StructureByteStride = 0;

result = device->CreateBuffer(&lightBufferDesc, NULL, &m\_lightBuffer);

if(FAILED(result))

{

return false;}

return true;}

void LightShaderClass::ShutdownShader()

{

if(m\_lightBuffer)

{

m\_lightBuffer->Release();

m\_lightBuffer = 0;}

if(m\_matrixBuffer)

{

m\_matrixBuffer->Release();

m\_matrixBuffer = 0;}

if(m\_sampleState)

{

m\_sampleState->Release();

m\_sampleState = 0;}

if(m\_layout)

{

m\_layout->Release();

m\_layout = 0;}

if(m\_pixelShader)

{

m\_pixelShader->Release();

m\_pixelShader = 0;}

if(m\_vertexShader)

{

m\_vertexShader->Release();

m\_vertexShader = 0;}

return;}

void LightShaderClass::OutputShaderErrorMessage(ID3D10Blob\* errorMessage, HWND hwnd, WCHAR\* shaderFilename)

{

char\* compileErrors;

unsigned long bufferSize, i;

ofstream fout;

compileErrors = (char\*)(errorMessage->GetBufferPointer());

bufferSize = (unsigned long) errorMessage->GetBufferSize();

fout.open("shader-error.txt");

for(i=0; i<bufferSize; i++)

{

fout << compileErrors[i];}

fout.close();

errorMessage->Release();

errorMessage = 0;

MessageBox(hwnd, L"Error compiling shader. Check shader-error.txt for message.", shaderFilename, MB\_OK);

return;}

bool LightShaderClass::SetShaderParameters(ID3D11DeviceContext\* deviceContext, XMMATRIX worldMatrix, XMMATRIX viewMatrix,

XMMATRIX projectionMatrix, ID3D11ShaderResourceView\* texture, XMFLOAT3 lightDirection, XMFLOAT4 diffuseColor)

{

HRESULT result;

D3D11\_MAPPED\_SUBRESOURCE mappedResource;

unsigned int bufferNumber;

MatrixBufferType\* dataPtr;

LightBufferType\* dataPtr2;

worldMatrix = XMMatrixTranspose(worldMatrix);

viewMatrix = XMMatrixTranspose(viewMatrix);

projectionMatrix = XMMatrixTranspose(projectionMatrix);

result = deviceContext->Map(m\_matrixBuffer, 0, D3D11\_MAP\_WRITE\_DISCARD, 0, &mappedResource);

if(FAILED(result))

{

return false;}

dataPtr = (MatrixBufferType\*)mappedResource.pData;

dataPtr->world = worldMatrix;

dataPtr->view = viewMatrix;

dataPtr->projection = projectionMatrix;

deviceContext->Unmap(m\_matrixBuffer, 0);

bufferNumber = 0;

deviceContext->VSSetConstantBuffers(bufferNumber, 1, &m\_matrixBuffer);

deviceContext->PSSetShaderResources(0, 1, &texture);

result = deviceContext->Map(m\_lightBuffer, 0, D3D11\_MAP\_WRITE\_DISCARD, 0, &mappedResource);

if(FAILED(result))

{

return false;}

dataPtr2 = (LightBufferType\*)mappedResource.pData;

dataPtr2->diffuseColor = diffuseColor;

dataPtr2->lightDirection = lightDirection;

dataPtr2->padding = 0.0f;

deviceContext->Unmap(m\_lightBuffer, 0);

bufferNumber = 0;

deviceContext->PSSetConstantBuffers(bufferNumber, 1, &m\_lightBuffer);

return true;}

void LightShaderClass::RenderShader(ID3D11DeviceContext\* deviceContext, int indexCount)

{

deviceContext->IASetInputLayout(m\_layout);

deviceContext->VSSetShader(m\_vertexShader, NULL, 0);

deviceContext->PSSetShader(m\_pixelShader, NULL, 0);

.

deviceContext->PSSetSamplers(0, 1, &m\_sampleState);

// Render the triangle.

deviceContext->DrawIndexed(indexCount, 0, 0);

return;

}



**Practical No. 5 Roll No. 13,14**

**Aim : Specular Lightning (Programmable Spot Lightning using Direct3D 11)**

**Class-lightshaderclass.h**

LightShaderClass::LightShaderClass()

{

m\_vertexShader = 0;

m\_pixelShader = 0;

m\_layout = 0;

m\_sampleState = 0;

m\_matrixBuffer = 0;

m\_cameraBuffer = 0;

m\_lightBuffer = 0;

}

bool LightShaderClass::Initialize(ID3D11Device\* device, HWND hwnd)

{

bool result;

WCHAR vertexshader[16] = L"light.vs";

WCHAR pixelshader[16] = L"light.ps";

result = InitializeShader(device, hwnd, vertexshader, pixelshader);

if(!result)

{

return false;}

return true;}

void LightShaderClass::Shutdown()

{

ShutdownShader();

return;}

bool LightShaderClass::Render(ID3D11DeviceContext\* deviceContext, int indexCount, XMMATRIX worldMatrix, XMMATRIX viewMatrix,

XMMATRIX projectionMatrix, ID3D11ShaderResourceView\* texture, XMFLOAT3 lightDirection, XMFLOAT4 ambientColor,

XMFLOAT4 diffuseColor, XMFLOAT3 cameraPosition, XMFLOAT4 specularColor, float specularPower)

{

bool result;

result = SetShaderParameters(deviceContext, worldMatrix, viewMatrix, projectionMatrix, texture, lightDirection, ambientColor, diffuseColor, cameraPosition, specularColor, specularPower);

if(!result)

{

return false;}

RenderShader(deviceContext, indexCount);

return true;}

bool LightShaderClass::InitializeShader(ID3D11Device\* device, HWND hwnd, WCHAR\* vsFilename, WCHAR\* psFilename)

{

HRESULT result;

ID3D10Blob\* errorMessage;

ID3D10Blob\* vertexShaderBuffer;

ID3D10Blob\* pixelShaderBuffer;

D3D11\_INPUT\_ELEMENT\_DESC polygonLayout[3];

unsigned int numElements;

D3D11\_SAMPLER\_DESC samplerDesc;

D3D11\_BUFFER\_DESC matrixBufferDesc;

D3D11\_BUFFER\_DESC cameraBufferDesc;

D3D11\_BUFFER\_DESC lightBufferDesc;

errorMessage = 0;

vertexShaderBuffer = 0;

pixelShaderBuffer = 0;

result = D3DCompileFromFile(vsFilename, NULL, NULL, "LightVertexShader", "vs\_5\_0", D3D10\_SHADER\_ENABLE\_STRICTNESS, 0, &vertexShaderBuffer, &errorMessage);

if(FAILED(result))

{ if(errorMessage)

{ OutputShaderErrorMessage(errorMessage, hwnd, vsFilename);

}

else

{

MessageBox(hwnd, vsFilename, L"Missing Shader File", MB\_OK);

}

return false;}

result = D3DCompileFromFile(psFilename, NULL, NULL, "LightPixelShader", "ps\_5\_0", D3D10\_SHADER\_ENABLE\_STRICTNESS, 0,

&pixelShaderBuffer, &errorMessage);

if(FAILED(result))

{ if(errorMessage)

{

OutputShaderErrorMessage(errorMessage, hwnd, psFilename);

}

else

{

MessageBox(hwnd, psFilename, L"Missing Shader File", MB\_OK);

}

return false;}

result = device->CreateVertexShader(vertexShaderBuffer->GetBufferPointer(), vertexShaderBuffer->GetBufferSize(), NULL, &m\_vertexShader);

if(FAILED(result))

{

return false;}

result = device->CreatePixelShader(pixelShaderBuffer->GetBufferPointer(), pixelShaderBuffer->GetBufferSize(), NULL, &m\_pixelShader);

if(FAILED(result))

{

return false;}

polygonLayout[0].SemanticName = "POSITION";

polygonLayout[0].SemanticIndex = 0;

polygonLayout[0].Format = DXGI\_FORMAT\_R32G32B32\_FLOAT;

polygonLayout[0].InputSlot = 0;

polygonLayout[0].AlignedByteOffset = 0;

polygonLayout[0].InputSlotClass = D3D11\_INPUT\_PER\_VERTEX\_DATA;

polygonLayout[0].InstanceDataStepRate = 0;

polygonLayout[1].SemanticName = "TEXCOORD";

polygonLayout[1].SemanticIndex = 0;

polygonLayout[1].Format = DXGI\_FORMAT\_R32G32\_FLOAT;

polygonLayout[1].InputSlot = 0;

polygonLayout[1].AlignedByteOffset = D3D11\_APPEND\_ALIGNED\_ELEMENT;

polygonLayout[1].InputSlotClass = D3D11\_INPUT\_PER\_VERTEX\_DATA;

polygonLayout[1].InstanceDataStepRate = 0;

polygonLayout[2].SemanticName = "NORMAL";

polygonLayout[2].SemanticIndex = 0;

polygonLayout[2].Format = DXGI\_FORMAT\_R32G32B32\_FLOAT;

polygonLayout[2].InputSlot = 0;

polygonLayout[2].AlignedByteOffset = D3D11\_APPEND\_ALIGNED\_ELEMENT;

polygonLayout[2].InputSlotClass = D3D11\_INPUT\_PER\_VERTEX\_DATA;

polygonLayout[2].InstanceDataStepRate = 0;

numElements = sizeof(polygonLayout) / sizeof(polygonLayout[0]);

result = device->CreateInputLayout(polygonLayout, numElements, vertexShaderBuffer->GetBufferPointer(), vertexShaderBuffer->GetBufferSize(),

&m\_layout);

if(FAILED(result))

{

return false;}

vertexShaderBuffer->Release();

vertexShaderBuffer = 0;

pixelShaderBuffer->Release();

pixelShaderBuffer = 0;

samplerDesc.Filter = D3D11\_FILTER\_MIN\_MAG\_MIP\_LINEAR;

samplerDesc.AddressU = D3D11\_TEXTURE\_ADDRESS\_WRAP;

samplerDesc.AddressV = D3D11\_TEXTURE\_ADDRESS\_WRAP;

samplerDesc.AddressW = D3D11\_TEXTURE\_ADDRESS\_WRAP;

samplerDesc.MipLODBias = 0.0f;

samplerDesc.MaxAnisotropy = 1;

samplerDesc.ComparisonFunc = D3D11\_COMPARISON\_ALWAYS;

samplerDesc.BorderColor[0] = 0;

samplerDesc.BorderColor[1] = 0;

samplerDesc.BorderColor[2] = 0;

samplerDesc.BorderColor[3] = 0;

samplerDesc.MinLOD = 0;

samplerDesc.MaxLOD = D3D11\_FLOAT32\_MAX;

result = device->CreateSamplerState(&samplerDesc, &m\_sampleState);

if(FAILED(result))

{

return false;}

matrixBufferDesc.Usage = D3D11\_USAGE\_DYNAMIC;

matrixBufferDesc.ByteWidth = sizeof(MatrixBufferType);

matrixBufferDesc.BindFlags = D3D11\_BIND\_CONSTANT\_BUFFER;

matrixBufferDesc.CPUAccessFlags = D3D11\_CPU\_ACCESS\_WRITE;

matrixBufferDesc.MiscFlags = 0;

matrixBufferDesc.StructureByteStride = 0;

result = device->CreateBuffer(&matrixBufferDesc, NULL, &m\_matrixBuffer);

if(FAILED(result))

{

return false;}

cameraBufferDesc.Usage = D3D11\_USAGE\_DYNAMIC;

cameraBufferDesc.ByteWidth = sizeof(CameraBufferType);

cameraBufferDesc.BindFlags = D3D11\_BIND\_CONSTANT\_BUFFER;

cameraBufferDesc.CPUAccessFlags = D3D11\_CPU\_ACCESS\_WRITE;

cameraBufferDesc.MiscFlags = 0;

cameraBufferDesc.StructureByteStride = 0;

result = device->CreateBuffer(&cameraBufferDesc, NULL, &m\_cameraBuffer);

if(FAILED(result))

{

return false;}

D3D11\_BIND\_CONSTANT\_BUFFER or CreateBuffer will fail.

lightBufferDesc.Usage = D3D11\_USAGE\_DYNAMIC;

lightBufferDesc.ByteWidth = sizeof(LightBufferType);

lightBufferDesc.BindFlags = D3D11\_BIND\_CONSTANT\_BUFFER;

lightBufferDesc.CPUAccessFlags = D3D11\_CPU\_ACCESS\_WRITE;

lightBufferDesc.MiscFlags = 0;

lightBufferDesc.StructureByteStride = 0;

result = device->CreateBuffer(&lightBufferDesc, NULL, &m\_lightBuffer);

if(FAILED(result))

{

return false;}

return true;}

void LightShaderClass::ShutdownShader()

{

if(m\_lightBuffer)

{

m\_lightBuffer->Release();

m\_lightBuffer = 0;}

if(m\_cameraBuffer)

{

m\_cameraBuffer->Release();

m\_cameraBuffer = 0;}

if(m\_matrixBuffer)

{

m\_matrixBuffer->Release();

m\_matrixBuffer = 0;}

if(m\_sampleState)

{

m\_sampleState->Release();

m\_sampleState = 0;}

if(m\_layout)

{

m\_layout->Release();

m\_layout = 0;}

if(m\_pixelShader)

{

m\_pixelShader->Release();

m\_pixelShader = 0;}

if(m\_vertexShader)

{

m\_vertexShader->Release();

m\_vertexShader = 0;}

return;}

void LightShaderClass::OutputShaderErrorMessage(ID3D10Blob\* errorMessage, HWND hwnd, WCHAR\* shaderFilename)

{

char\* compileErrors;

unsigned long bufferSize, i;

ofstream fout;

compileErrors = (char\*)(errorMessage->GetBufferPointer());

bufferSize = errorMessage->GetBufferSize();

fout.open("shader-error.txt");

for(i=0; i<bufferSize; i++)

{

fout << compileErrors[i];}

fout.close();

errorMessage->Release();

errorMessage = 0;

MessageBox(hwnd, L"Error compiling shader. Check shader-error.txt for message.", shaderFilename, MB\_OK);

return;}

bool LightShaderClass::SetShaderParameters(ID3D11DeviceContext\* deviceContext, XMMATRIX worldMatrix, XMMATRIX viewMatrix,

XMMATRIX projectionMatrix, ID3D11ShaderResourceView\* texture, XMFLOAT3 lightDirection,

XMFLOAT4 ambientColor, XMFLOAT4 diffuseColor, XMFLOAT3 cameraPosition, XMFLOAT4 specularColor,

float specularPower)

{

HRESULT result;

D3D11\_MAPPED\_SUBRESOURCE mappedResource;

unsigned int bufferNumber;

MatrixBufferType\* dataPtr;

LightBufferType\* dataPtr2;

CameraBufferType\* dataPtr3;

worldMatrix = XMMatrixTranspose(worldMatrix);

viewMatrix = XMMatrixTranspose(viewMatrix);

projectionMatrix = XMMatrixTranspose(projectionMatrix);

result = deviceContext->Map(m\_matrixBuffer, 0, D3D11\_MAP\_WRITE\_DISCARD, 0, &mappedResource);

if(FAILED(result))

{

return false;}

dataPtr = (MatrixBufferType\*)mappedResource.pData;

dataPtr->world = worldMatrix;

dataPtr->view = viewMatrix;

dataPtr->projection = projectionMatrix;

deviceContext->Unmap(m\_matrixBuffer, 0);

bufferNumber = 0;

deviceContext->VSSetConstantBuffers(bufferNumber, 1, &m\_matrixBuffer);

result = deviceContext->Map(m\_cameraBuffer, 0, D3D11\_MAP\_WRITE\_DISCARD, 0, &mappedResource);

if(FAILED(result))

{

return false;}

dataPtr3 = (CameraBufferType\*)mappedResource.pData;

dataPtr3->cameraPosition = cameraPosition;

dataPtr3->padding = 0.0f;

deviceContext->Unmap(m\_cameraBuffer, 0);

bufferNumber = 1;

deviceContext->VSSetConstantBuffers(bufferNumber, 1, &m\_cameraBuffer);

deviceContext->PSSetShaderResources(0, 1, &texture);

result = deviceContext->Map(m\_lightBuffer, 0, D3D11\_MAP\_WRITE\_DISCARD, 0, &mappedResource);

if(FAILED(result))

{

return false;}

dataPtr2 = (LightBufferType\*)mappedResource.pData;

dataPtr2->ambientColor = ambientColor;

dataPtr2->diffuseColor = diffuseColor;

dataPtr2->lightDirection = lightDirection;

dataPtr2->specularColor = specularColor;

dataPtr2->specularPower = specularPower;

deviceContext->Unmap(m\_lightBuffer, 0);

bufferNumber = 0;

deviceContext->PSSetConstantBuffers(bufferNumber, 1, &m\_lightBuffer);

return true;

}

void LightShaderClass::RenderShader(ID3D11DeviceContext\* deviceContext, int indexCount)

{

deviceContext->IASetInputLayout(m\_layout);

deviceContext->VSSetShader(m\_vertexShader, NULL, 0);

deviceContext->PSSetShader(m\_pixelShader, NULL, 0);

deviceContext->PSSetSamplers(0, 1, &m\_sampleState);

deviceContext->DrawIndexed(indexCount, 0, 0);

return;}

**Class-modelclass.h**

ModelClass::ModelClass()

{

m\_vertexBuffer = 0;

m\_indexBuffer = 0;

m\_Texture = 0;

m\_model = 0;

}

bool ModelClass::Initialize(ID3D11Device\* device, ID3D11DeviceContext\* deviceContext, char\* modelFilename, char\* textureFilename)

{

bool result;

result = LoadModel(modelFilename);

if(!result)

{

return false;}

result = InitializeBuffers(device);

if(!result)

{

return false;}

result = LoadTexture(device, deviceContext, textureFilename);

if(!result)

{

return false;}

return true;}

void ModelClass::Shutdown()

{

ReleaseTexture();

ShutdownBuffers();

ReleaseModel();

return;}

void ModelClass::Render(ID3D11DeviceContext\* deviceContext)

{

RenderBuffers(deviceContext);

return;}

int ModelClass::GetIndexCount()

{

return m\_indexCount;}

ID3D11ShaderResourceView\* ModelClass::GetTexture()

{

return m\_Texture->GetTexture();}

bool ModelClass::InitializeBuffers(ID3D11Device\* device)

{

VertexType\* vertices;

unsigned long\* indices;

D3D11\_BUFFER\_DESC vertexBufferDesc, indexBufferDesc;

D3D11\_SUBRESOURCE\_DATA vertexData, indexData;

HRESULT result;

int i;

vertices = new VertexType[m\_vertexCount];

if(!vertices)

{

return false;}

indices = new unsigned long[m\_indexCount];

if(!indices)

{

return false;}

for(i=0; i<m\_vertexCount; i++)

{

vertices[i].position = XMFLOAT3(m\_model[i].x, m\_model[i].y, m\_model[i].z);

vertices[i].texture = XMFLOAT2(m\_model[i].tu, m\_model[i].tv);

vertices[i].normal = XMFLOAT3(m\_model[i].nx, m\_model[i].ny, m\_model[i].nz);

indices[i] = i;}

vertexBufferDesc.Usage = D3D11\_USAGE\_DEFAULT;

vertexBufferDesc.ByteWidth = sizeof(VertexType) \* m\_vertexCount;

vertexBufferDesc.BindFlags = D3D11\_BIND\_VERTEX\_BUFFER;

vertexBufferDesc.CPUAccessFlags = 0;

vertexBufferDesc.MiscFlags = 0;

vertexBufferDesc.StructureByteStride = 0;

vertexData.pSysMem = vertices;

vertexData.SysMemPitch = 0;

vertexData.SysMemSlicePitch = 0;

result = device->CreateBuffer(&vertexBufferDesc, &vertexData, &m\_vertexBuffer);

if(FAILED(result))

{

return false;}

indexBufferDesc.Usage = D3D11\_USAGE\_DEFAULT;

indexBufferDesc.ByteWidth = sizeof(unsigned long) \* m\_indexCount;

indexBufferDesc.BindFlags = D3D11\_BIND\_INDEX\_BUFFER;

indexBufferDesc.CPUAccessFlags = 0;

indexBufferDesc.MiscFlags = 0;

indexBufferDesc.StructureByteStride = 0;

indexData.pSysMem = indices;

indexData.SysMemPitch = 0;

indexData.SysMemSlicePitch = 0;

result = device->CreateBuffer(&indexBufferDesc, &indexData, &m\_indexBuffer);

if(FAILED(result))

{

return false;}

delete [] vertices;

vertices = 0;

delete [] indices;

indices = 0;

return true;}

void ModelClass::ShutdownBuffers()

{

if(m\_indexBuffer)

{

m\_indexBuffer->Release();

m\_indexBuffer = 0;}

if(m\_vertexBuffer)

{

m\_vertexBuffer->Release();

m\_vertexBuffer = 0;}

return;}

void ModelClass::RenderBuffers(ID3D11DeviceContext\* deviceContext)

{

unsigned int stride;

unsigned int offset;

stride = sizeof(VertexType);

offset = 0;

deviceContext->IASetVertexBuffers(0, 1, &m\_vertexBuffer, &stride, &offset);

deviceContext->IASetIndexBuffer(m\_indexBuffer, DXGI\_FORMAT\_R32\_UINT, 0);

deviceContext->IASetPrimitiveTopology(D3D11\_PRIMITIVE\_TOPOLOGY\_TRIANGLELIST);

return;

}

bool ModelClass::LoadTexture(ID3D11Device\* device, ID3D11DeviceContext\* deviceContext, char\* filename)

{

bool result;

m\_Texture = new TextureClass;

if (!m\_Texture)

{

return false;}

result = m\_Texture->Initialize(device, deviceContext, filename);

if (!result)

{

return false;}

return true;}

void ModelClass::ReleaseTexture()

{

if(m\_Texture)

{

m\_Texture->Shutdown();

delete m\_Texture;

m\_Texture = 0;}

return;}

bool ModelClass::LoadModel(char\* filename)

{

ifstream fin;

char input;

int i;

fin.open(filename);

if(fin.fail())

{

return false;}

fin.get(input);

while(input != ':')

{

fin.get(input);}

fin >> m\_vertexCount;

m\_indexCount = m\_vertexCount;

m\_model = new ModelType[m\_vertexCount];

if(!m\_model)

{

return false;}

fin.get(input);

while(input != ':')

{

fin.get(input);}

fin.get(input);

fin.get(input);

for(i=0; i<m\_vertexCount; i++)

{

fin >> m\_model[i].x >> m\_model[i].y >> m\_model[i].z;

fin >> m\_model[i].tu >> m\_model[i].tv;

fin >> m\_model[i].nx >> m\_model[i].ny >> m\_model[i].nz;

}

fin.close();

return true;}

void ModelClass::ReleaseModel()

{

if(m\_model)

{

delete [] m\_model;

m\_model = 0;}

return;}

