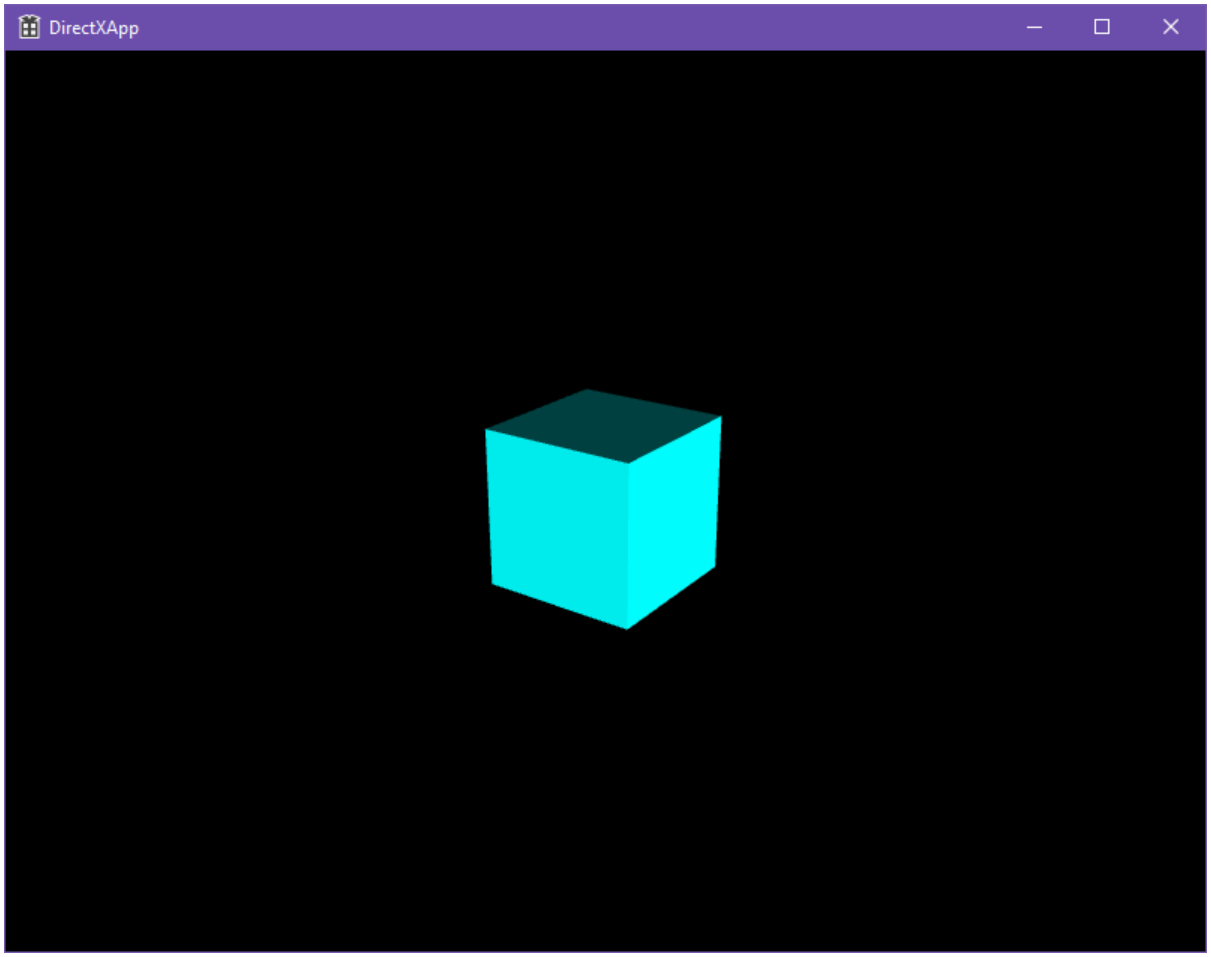
Graphics Part II: Code Tutorial for Week 4c

Dr P. Perakis

Preparation

We have provided a fully working code for the implementation of the scene graph and a `CubeNode` class to render a Cube in the `Exercise_04_2.zip` file. This code can be used as a starting point for the following work. You will also use the files provided in the `Exercise_04_3-Texture_Files.zip` file.



Introduction

In the following work you have to add texture to the lit cube example.

The changes to add texturing are shown in the following text.

⇨ First you have to add the following code files in your C++ project:

WICTextureLoader.cpp
WICTextureLoader.h

⇨ And copy the following image files in your working folder:

Wood.png
woodbox.bmp

As you will see, there are minimal changes needed.

- ⇨ In CubeNode.h

```
#include "WICTextureLoader.h"
```

- ⇨ In CubeNode class:

The vertex structure includes texture coordinates

```
struct Vertex
{
    Vector3 Position;
    Vector3 Normal;
    Vector2 TexCoords;
};
```

There is a pointer variable for the texture

```
ComPtr<ID3D11ShaderResourceView> _texture;
```

And a function for loading the texture from a file

```
void BuildTexture(const wchar_t* FileName);
```

- ⇨ The implementation of this function is in `CubeNode.cpp`:

```
void CubeNode::BuildTexture(const wchar_t* FileName)
{
	// Note that in order to use CreateWICTextureFromFile, we
	// need to ensure we make a call to CoInitializeEx in our
	// Initialise method (and make the corresponding call to
	// CoUninitialize in the Shutdown method).  Otherwise,
	// the following call will throw an exception
	ThrowIfFailed(CreateWICTextureFromFile(_device.Get(), _deviceContext.Get(),
FileName, nullptr,   _texture.GetAddressOf()     ));
}
```

The provided implementation of the `vertexDesc[]`:

```
void CubeNode::BuildVertexLayout()
{
	// Create the vertex input layout. This tells DirectX the format
	// of each of the vertices we are sending to it.
	D3D11_INPUT_ELEMENT_DESC vertexDesc[] =
	{
		{ "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
D3D11_INPUT_PER_VERTEX_DATA, 0 },
		{ "NORMAL", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 12,
D3D11_INPUT_PER_VERTEX_DATA, 0 },
		{ "TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT, 0, 24,
D3D11_INPUT_PER_VERTEX_DATA, 0 }
	};
```

```
	ThrowIfFailed(_device->CreateInputLayout(vertexDesc, ARRAYSIZE(vertexDesc),
_vertexShaderByteCode->GetBufferPointer(), _vertexShaderByteCode->GetBufferSize(),
_layout.GetAddressOf()));
	_deviceContext->IASetInputLayout(_layout.Get());
}
```

- ⇨ Inside `void CubeNode::Render()`

```
// Set the texture to be used by the pixel shader
_deviceContext->PSSetShaderResources(0, 1, _texture.GetAddressOf());
```

- ⇨ Inside `void CubeNode::BuildGeometry()`. Here the texture coordinates in the vertex structure are hard-coded, but in most cases, you will load these from a model file.

```
// Create vertex buffer
Vertex vertices[] =
{
	// side 1
	{ Vector3(-1.0f, -1.0f, +1.0f), Vector3(0.0f, 0.0f, 0.0f),  Vector2(0.0f, 0.0f) },
	{ Vector3(+1.0f, -1.0f, +1.0f), Vector3(0.0f, 0.0f, 0.0f),  Vector2(0.0f, 1.0f) },
	{ Vector3(-1.0f, +1.0f, +1.0f), Vector3(0.0f, 0.0f, 0.0f),  Vector2(1.0f, 0.0f) },
	{ Vector3(+1.0f, +1.0f, +1.0f), Vector3(0.0f, 0.0f, 0.0f),  Vector2(1.0f, 1.0f) },
	// side 2
	{ Vector3(-1.0f, -1.0f, -1.0f), Vector3(0.0f, 0.0f, 0.0f),  Vector2(0.0f, 0.0f) },
	{ Vector3(-1.0f, +1.0f, -1.0f), Vector3(0.0f, 0.0f, 0.0f),  Vector2(0.0f, 1.0f) },
	{ Vector3(+1.0f, -1.0f, -1.0f), Vector3(0.0f, 0.0f, 0.0f),  Vector2(1.0f, 0.0f) },
	{ Vector3(+1.0f, +1.0f, -1.0f), Vector3(0.0f, 0.0f, 0.0f),  Vector2(1.0f, 1.0f) },
	// side 3
	{ Vector3(-1.0f, +1.0f, -1.0f), Vector3(0.0f, 0.0f, 0.0f),  Vector2(0.0f, 0.0f) },
	{ Vector3(-1.0f, +1.0f, +1.0f), Vector3(0.0f, 0.0f, 0.0f),  Vector2(0.0f, 1.0f) },
	{ Vector3(+1.0f, +1.0f, -1.0f), Vector3(0.0f, 0.0f, 0.0f),  Vector2(1.0f, 0.0f) },
	{ Vector3(+1.0f, +1.0f, +1.0f), Vector3(0.0f, 0.0f, 0.0f),  Vector2(1.0f, 1.0f) },
	// side 4
	{ Vector3(-1.0f, -1.0f, -1.0f), Vector3(0.0f, 0.0f, 0.0f),  Vector2(0.0f, 0.0f) },
	{ Vector3(+1.0f, -1.0f, -1.0f), Vector3(0.0f, 0.0f, 0.0f),  Vector2(0.0f, 1.0f) },
	{ Vector3(-1.0f, -1.0f, +1.0f), Vector3(0.0f, 0.0f, 0.0f),  Vector2(1.0f, 0.0f) },
	{ Vector3(+1.0f, -1.0f, +1.0f), Vector3(0.0f, 0.0f, 0.0f),  Vector2(1.0f, 1.0f) },
	// side 5
	{ Vector3(+1.0f, -1.0f, -1.0f), Vector3(0.0f, 0.0f, 0.0f),  Vector2(0.0f, 0.0f) },
	{ Vector3(+1.0f, +1.0f, -1.0f), Vector3(0.0f, 0.0f, 0.0f),  Vector2(0.0f, 1.0f) },
	{ Vector3(+1.0f, -1.0f, +1.0f), Vector3(0.0f, 0.0f, 0.0f),  Vector2(1.0f, 0.0f) },
	{ Vector3(+1.0f, +1.0f, +1.0f), Vector3(0.0f, 0.0f, 0.0f),  Vector2(1.0f, 1.0f) },
	// side 6
	{ Vector3(-1.0f, -1.0f, -1.0f), Vector3(0.0f, 0.0f, 0.0f),  Vector2(0.0f, 0.0f) },
	{ Vector3(-1.0f, -1.0f, +1.0f), Vector3(0.0f, 0.0f, 0.0f),  Vector2(0.0f, 1.0f) },
	{ Vector3(-1.0f, +1.0f, -1.0f), Vector3(0.0f, 0.0f, 0.0f),  Vector2(1.0f, 0.0f) },
	{ Vector3(-1.0f, +1.0f, +1.0f), Vector3(0.0f, 0.0f, 0.0f),  Vector2(1.0f, 1.0f) }
};
```

- ⇨ Inside `bool CubeNode::Initialise()` we have to load the texture file

```
BuildTexture(L"Woodbox.bmp");
//BuildTexture(L"Wood.png");
```

Implementing the new shader

An implementation of `shader.hlsl` is provided.

```
cbuffer ConstantBuffer
{
	matrix worldViewProjection;
	matrix worldTransformation;
	float4 materialColour;
	float4 ambientLightColour;
	float4 directionalLightColour;
	float4 directionalLightVector;
};

struct VertexIn
{
	float3 InputPosition : POSITION;
	float3 Normal        : NORMAL;
	float2 TexCoord      : TEXCOORD;
};

Texture2D Texture;
SamplerState ss;

struct VertexOut
{
	float4 OutputPosition	: SV_POSITION;
	float4 Colour		: COLOR;
	float2 TexCoord		: TEXCOORD;
};

VertexOut VS(VertexIn vin)
{
	VertexOut vout;

	// Transform to homogeneous clip space.
	vout.OutputPosition = mul(worldViewProjection, float4(vin.InputPosition,
1.0f));

	// calculate the diffuse light and add it to the ambient light
	float4 vectorBackToLight = -normalize(directionalLightVector); // directional
light
	//float4 vectorBackToLight = -normalize(directionalLightVector -
mul(worldTransformation, float4(vin.InputPosition, 1.0f))); // point light

	float4 adjustedNormal = normalize(mul(worldTransformation, float4(vin.Normal,
0.0f)));
	float diffuseBrightness = saturate(dot(adjustedNormal, vectorBackToLight));

	vout.Colour = materialColour * saturate(ambientLightColour + diffuseBrightness
* directionalLightColour); // ALL

	vout.TexCoord = vin.TexCoord;

	return vout;
}

float4 PS(VertexOut pin) : SV_Target
{
	//return pin.Colour;
	return pin.Colour * Texture.Sample(ss, pin.TexCoord);
}
```

Adding animation

Now we can add a movement to the Cube.

For doing it we have to add transformations' code in the DirectXApp.cpp:

```
int _time;

void DirectXApp::CreateSceneGraph()
{
	SceneGraphPointer sceneGraph = GetSceneGraph();

	// Add your code here to build up the scene graph

	// Create a cube node using your full lighting, shader, and triangle setup
	SceneNodePointer cube = make_shared<CubeNode>(L"CubeNode");

	// Add the cube to the scene
	sceneGraph->Add(cube);

	_time = 0;

}

void DirectXApp::UpdateSceneGraph()
{
	SceneGraphPointer sceneGraph = GetSceneGraph();

	// This method is called at the frame rate frequency set in the Framework class
(by default,
	// 60 times a second).  Perform any updates to the scene graph needed for the
next frame.

	Matrix worldTransformation = Matrix::CreateRotationY(_time * XM_PI / 180.0f);

	sceneGraph->Find(L"CubeNode")->SetWorldTransform(worldTransformation);

	_time = (_time + 1) % 1000;
}
```

Note: A fully working code for the implementation of the texturing is provided to you in the `Exercise_04_3.zip` *file.*

The output of the executable

After compiling and running the executable, the application window should look like this:

Material Colour with Texture