This document summarises the changes we need to make to the lit cube example from week 3 in order to add a texture. The changes to add texturing are shown in bold text – the code that is grayed out is the code that is unchanged. As you can see, there are minimal changes needed. Here the texture coordinates in the vertex structure are hard-coded, but in most cases, you will load these from a model file.

DirectXApp.cpp:

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

// Geometry.h contains the vertex and constant buffer structures
// as well as the vertices and indices for a cube

#include "Geometry.h"

// DirectX libraries that are needed
#pragma comment(lib, "d3d11.lib")
#pragma comment(lib, "d3dcompiler.lib")

DirectXApp app;

DirectXApp::DirectXApp() : Framework(800, 600)
{
	// Initialise vectors used to create camera.  We will look
	// at this in detail later
	_eyePosition = Vector3(0.0f, 0.0f, -10.0f);
	_focalPointPosition = Vector3(0.0f, 0.0f, 0.0f);
	_upVector = Vector3(0.0f, 1.0f, 0.0f);
}

bool DirectXApp::Initialise()
{
	// The call to CoInitializeEx is needed if we are using
	// textures since the WIC library used requires it. Note this is done in the
	// DirectXFramework for you so you do not need to code it yourself if you are using the
	// framework, but it is included here for completeness

	if (FAILED(CoInitializeEx(nullptr, COINIT_APARTMENTTHREADED)))
	{
		return false;
	}
	if (!GetDeviceAndSwapChain())
	{
		return false;
	}
	OnResize(WM_EXITSIZEMOVE);
	BuildVertexNormals();
	BuildGeometryBuffers();
	BuildShaders();
	BuildVertexLayout();
	BuildConstantBuffer();
	BuildRasteriserState();
	BuildTexture();

	return true;
}
```

```
void DirectXApp::Update()
{
       _worldTransformation = Matrix::CreateRotationY(_rotationAngle * XM_PI / 180.0f);
       _rotationAngle = (_rotationAngle + 1) % 360;
}

void DirectXApp::Render()
{
       const float clearColour[] = { 0.0f, 0.0f, 0.0f, 1.0f };
       _deviceContext->ClearRenderTargetView(_renderTargetView.Get(), clearColour);
       _deviceContext->ClearDepthStencilView(_depthStencilView.Get(),
                          D3D11_CLEAR_DEPTH | D3D11_CLEAR_STENCIL, 1.0f, 0);

       _viewTransformation = XMMatrixLookAtLH(_eyePosition, _focalPointPosition, _upVector);
       _projectionTransformation = XMMatrixPerspectiveFovLH(XM_PIDIV4,
                    static_cast<float>(GetWindowWidth()) / GetWindowHeight(), 1.0f, 100.0f);

       // Calculate the world x view x projection transformation
       Matrix completeTransformation = _worldTransformation * _viewTransformation *
                                       _projectionTransformation;

       CBuffer constantBuffer;
       constantBuffer.WorldViewProjection = completeTransformation;
       constantBuffer.World = _worldTransformation;
       constantBuffer.AmbientLightColour = Vector4(0.5f, 0.5f, 0.5f, 1.0f);
       constantBuffer.DirectionalLightVector = Vector4(-1.0f, -1.0f, 1.0f, 0.0f);
       constantBuffer.DirectionalLightColour = Vector4(Colors::White);

       // Update the constant buffer. Note the layout of the constant buffer must match that
       // in the shader
       _deviceContext->VSSetConstantBuffers(0, 1, _constantBuffer.GetAddressOf());
       _deviceContext->UpdateSubresource(_constantBuffer.Get(), 0, 0, &constantBuffer, 0, 0);

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

       // Now render the cube
       // Specify the distance between vertices and the starting point in the vertex buffer
       UINT stride = sizeof(Vertex);
       UINT offset = 0;
       // Set the vertex buffer and index buffer we are going to use
       _deviceContext->IASetVertexBuffers(0, 1, _vertexBuffer.GetAddressOf(),
                                         &stride, &offset);
       _deviceContext->IASetIndexBuffer(_indexBuffer.Get(), DXGI_FORMAT_R32_UINT, 0);

       // Specify the layout of the polygons (it will rarely be different to this)
       _deviceContext->IASetPrimitiveTopology(D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST);

       // Specify the layout of the input vertices.  This must match the layout of the
       // input vertices in the shader
       _deviceContext->IASetInputLayout(_layout.Get());

       // Specify the vertex and pixel shaders we are going to use
       _deviceContext->VSSetShader(_vertexShader.Get(), 0, 0);
       _deviceContext->PSSetShader(_pixelShader.Get(), 0, 0);

       // Specify details about how the object is to be drawn
       _deviceContext->RSSetState(_rasteriserState.Get());

       // Now draw the first cube
       _deviceContext->DrawIndexed(ARRAYSIZE(indices), 0, 0);

       // Update the window
       ThrowIfFailed(_swapChain->Present(0, 0));
}
```

```
void DirectXApp::Shutdown()
{
    // Required because we called CoInitialize above.  Note this is done in the
    // DirectXFramework for you so you do not need to code it yourself if you are using the
    // framework, but it is included here for completeness
    CoUninitialize();
}

// OnResize is called by the framework whenever Windows gets a WM_Size message. We need to
recreate
// the draw and depth buffers to reflect the revised height and width of the window.

void DirectXApp::OnResize(WPARAM wParam)
{
    // We only want to resize the buffers when the user has
    // finished dragging the window to the new size.  Windows
    // sends a value of WM_EXITSIZEMOVE to WM_SIZE when the
    // resizing is complete.
    if (wParam != WM_EXITSIZEMOVE)
    {
        return;
    }
    // Free any existing render and depth views (which
    // would be the case if the window was being resized)
    _renderTargetView = nullptr;
    _depthStencilView = nullptr;
    _depthStencilBuffer = nullptr;

    ThrowIfFailed(_swapChain->ResizeBuffers(1, GetWindowWidth(),
                              GetWindowHeight(), DXGI_FORMAT_R8G8B8A8_UNORM, 0));

    // Create a drawing surface for DirectX to render to
    ComPtr<ID3D11Texture2D> backBuffer;
    ThrowIfFailed(_swapChain->GetBuffer(0, IID_PPV_ARGS(&backBuffer)));
    ThrowIfFailed(_device->CreateRenderTargetView(backBuffer.Get(), NULL,
                                     _renderTargetView.GetAddressOf()));

    // The depth buffer is used by DirectX to ensure
    // that pixels of closer objects are drawn over pixels of more
    // distant objects.

    // First, we need to create a texture (bitmap) for the depth buffer
    D3D11_TEXTURE2D_DESC depthBufferTexture = { 0 };
    depthBufferTexture.Width = GetWindowWidth();
    depthBufferTexture.Height = GetWindowHeight();
    depthBufferTexture.ArraySize = 1;
    depthBufferTexture.MipLevels = 1;
    depthBufferTexture.SampleDesc.Count = 4;
    depthBufferTexture.Format = DXGI_FORMAT_D32_FLOAT;
    depthBufferTexture.Usage = D3D11_USAGE_DEFAULT;
    depthBufferTexture.BindFlags = D3D11_BIND_DEPTH_STENCIL;

    // Create the depth buffer.
    ComPtr<ID3D11Texture2D> depthBuffer;
    ThrowIfFailed(_device->CreateTexture2D(&depthBufferTexture, NULL,
                  depthBuffer.GetAddressOf()));
    ThrowIfFailed(_device->CreateDepthStencilView(depthBuffer.Get(), 0,
                              _depthStencilView.GetAddressOf()));

    // Bind the render target view buffer and the depth stencil view buffer to the
    // output-merger stage
    // of the pipeline.
    _deviceContext->OMSetRenderTargets(1, _renderTargetView.GetAddressOf(),
                              _depthStencilView.Get());
```

```
       // Specify a viewport of the required size
       D3D11_VIEWPORT viewPort = { 0 };
       viewPort.Width = static_cast<float>(GetWindowWidth());
       viewPort.Height = static_cast<float>(GetWindowHeight());
       viewPort.MinDepth = 0.0f;
       viewPort.MaxDepth = 1.0f;
       viewPort.TopLeftX = 0;
       viewPort.TopLeftY = 0;
       _deviceContext->RSSetViewports(1, &viewPort);
}

bool DirectXApp::GetDeviceAndSwapChain()
{
       UINT createDeviceFlags = 0;

       // We are going to only accept a hardware driver or a WARP
       // driver
       D3D_DRIVER_TYPE driverTypes[] =
       {
             D3D_DRIVER_TYPE_HARDWARE,
             D3D_DRIVER_TYPE_WARP
       };
       unsigned int totalDriverTypes = ARRAYSIZE(driverTypes);

       D3D_FEATURE_LEVEL featureLevels[] =
       {
             D3D_FEATURE_LEVEL_11_0
       };
       unsigned int totalFeatureLevels = ARRAYSIZE(featureLevels);

       DXGI_SWAP_CHAIN_DESC swapChainDesc = { 0 };
       swapChainDesc.BufferCount = 1;
       swapChainDesc.BufferDesc.Width = GetWindowWidth();
       swapChainDesc.BufferDesc.Height = GetWindowHeight();
       swapChainDesc.BufferDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
       // Set the refresh rate to 0 and let DXGI determine the best option (refer to
       // DXGI best practices)
       swapChainDesc.BufferDesc.RefreshRate.Numerator = 0;
       swapChainDesc.BufferDesc.RefreshRate.Denominator = 0;
       swapChainDesc.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;
       swapChainDesc.OutputWindow = GetHWnd();
       // Start out windowed
       swapChainDesc.Windowed = true;
       // Enable multi-sampling to give smoother lines (set to 1 if performance
       // becomes an issue)
       swapChainDesc.SampleDesc.Count = 4;
       swapChainDesc.SampleDesc.Quality = 0;

       // Loop through the driver types to determine which one is available to us
       D3D_DRIVER_TYPE driverType = D3D_DRIVER_TYPE_UNKNOWN;

       for (unsigned int driver = 0; driver < totalDriverTypes &&
                                 driverType == D3D_DRIVER_TYPE_UNKNOWN; driver++)
       {
             if (SUCCEEDED(D3D11CreateDeviceAndSwapChain(0,
                                                   createDeviceFlags,
                                                   featureLevels,
                                                   totalFeatureLevels,
                                                   D3D11_SDK_VERSION,
                                                   &swapChainDesc,
                                                   _swapChain.GetAddressOf(),
                                                   _device.GetAddressOf(),
                                                   _deviceContext.GetAddressOf()
                                                   )))
```

```
        {
            driverType = driverTypes[driver];
        }
    }
    if (driverType == D3D_DRIVER_TYPE_UNKNOWN)
    {
        // Unable to find a suitable device driver
        return false;
    }
    return true;
}

void DirectXApp::BuildGeometryBuffers()
{
    // This method uses the arrays defined in Geometry.h
    //
    // Setup the structure that specifies how big the vertex
    // buffer should be
    D3D11_BUFFER_DESC vertexBufferDescriptor = { 0 };
    vertexBufferDescriptor.Usage = D3D11_USAGE_IMMUTABLE;
    vertexBufferDescriptor.ByteWidth = sizeof(Vertex) * ARRAYSIZE(vertices);
    vertexBufferDescriptor.BindFlags = D3D11_BIND_VERTEX_BUFFER;
    vertexBufferDescriptor.CPUAccessFlags = 0;
    vertexBufferDescriptor.MiscFlags = 0;
    vertexBufferDescriptor.StructureByteStride = 0;

    // Now set up a structure that tells DirectX where to get the
    // data for the vertices from
    D3D11_SUBRESOURCE_DATA vertexInitialisationData = { 0 };
    vertexInitialisationData.pSysMem = &vertices;

    // and create the vertex buffer
    ThrowIfFailed(_device->CreateBuffer(&vertexBufferDescriptor, &vertexInitialisationData,
                        _vertexBuffer.GetAddressOf()));

    // Setup the structure that specifies how big the index
    // buffer should be
    D3D11_BUFFER_DESC indexBufferDescriptor = { 0 };
    indexBufferDescriptor.Usage = D3D11_USAGE_IMMUTABLE;
    indexBufferDescriptor.ByteWidth = sizeof(UINT) * ARRAYSIZE(indices);
    indexBufferDescriptor.BindFlags = D3D11_BIND_INDEX_BUFFER;
    indexBufferDescriptor.CPUAccessFlags = 0;
    indexBufferDescriptor.MiscFlags = 0;
    indexBufferDescriptor.StructureByteStride = 0;

    // Now set up a structure that tells DirectX where to get the
    // data for the indices from
    D3D11_SUBRESOURCE_DATA indexInitialisationData;
    indexInitialisationData.pSysMem = &indices;

    // and create the index buffer
    ThrowIfFailed(_device->CreateBuffer(&indexBufferDescriptor, &indexInitialisationData,
                _indexBuffer.GetAddressOf()));
}

void DirectXApp::BuildShaders()
{
    DWORD shaderCompileFlags = 0;
#if defined( _DEBUG )
    shaderCompileFlags = D3DCOMPILE_DEBUG | D3DCOMPILE_SKIP_OPTIMIZATION;
#endif

    ComPtr<ID3DBlob> compilationMessages = nullptr;
    //Compile vertex shader
    HRESULT hr = D3DCompileFromFile(ShaderFileName,
        nullptr, D3D_COMPILE_STANDARD_FILE_INCLUDE,
        VertexShaderName, "vs_5_0",
        shaderCompileFlags, 0,
        _vertexShaderByteCode.GetAddressOf(),
        compilationMessages.GetAddressOf());

    if (compilationMessages.Get() != nullptr)
    {
        // If there were any compilation messages, display them
        MessageBoxA(0, (char*)compilationMessages->GetBufferPointer(), 0, 0);
    }
    // Even if there are no compiler messages, check to make sure there were no other
    // errors.
    ThrowIfFailed(hr);
    ThrowIfFailed(_device->CreateVertexShader(_vertexShaderByteCode->GetBufferPointer(),
            _vertexShaderByteCode->GetBufferSize(), NULL,
            _vertexShader.GetAddressOf()));

    // Compile pixel shader
    hr = D3DCompileFromFile(ShaderFileName,
        nullptr, D3D_COMPILE_STANDARD_FILE_INCLUDE,
        PixelShaderName, "ps_5_0",
        shaderCompileFlags, 0,
        _pixelShaderByteCode.GetAddressOf(),
        compilationMessages.GetAddressOf());

    if (compilationMessages.Get() != nullptr)
    {
        // If there were any compilation messages, display them
        MessageBoxA(0, (char*)compilationMessages->GetBufferPointer(), 0, 0);
    }
    ThrowIfFailed(hr);
    ThrowIfFailed(_device->CreatePixelShader(_pixelShaderByteCode->GetBufferPointer(),
            _pixelShaderByteCode->GetBufferSize(), NULL, _pixelShader.GetAddressOf()));
}

void DirectXApp::BuildVertexLayout()
{
    // Create the vertex input layout. This tells DirectX the format
    // of each of the vertices we are sending to it. The vertexDesc array is
    // defined in Geometry.h

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

void DirectXApp::BuildConstantBuffer()
{
    D3D11_BUFFER_DESC bufferDesc;
    ZeroMemory(&bufferDesc, sizeof(bufferDesc));
    bufferDesc.Usage = D3D11_USAGE_DEFAULT;
    bufferDesc.ByteWidth = sizeof(CBuffer);
    bufferDesc.BindFlags = D3D11_BIND_CONSTANT_BUFFER;

    ThrowIfFailed(_device->CreateBuffer(&bufferDesc, NULL, _constantBuffer.GetAddressOf()));
}

void DirectXApp::BuildRasteriserState()
{   // Set default and wireframe rasteriser states
    D3D11_RASTERIZER_DESC rasteriserDesc;
    rasteriserDesc.CullMode = D3D11_CULL_BACK;
    rasteriserDesc.FrontCounterClockwise = false;
    rasteriserDesc.DepthBias = 0;
    rasteriserDesc.SlopeScaledDepthBias = 0.0f;
    rasteriserDesc.DepthBiasClamp = 0.0f;
    rasteriserDesc.DepthClipEnable = true;
    rasteriserDesc.ScissorEnable = false;
    rasteriserDesc.MultisampleEnable = false;
    rasteriserDesc.AntialiasedLineEnable = false;
    rasteriserDesc.FillMode = D3D11_FILL_SOLID;
    ThrowIfFailed(_device->CreateRasterizerState(&rasteriserDesc,
                              _rasteriserState.GetAddressOf()));
}

void DirectXApp::BuildTexture()
{
    // 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(),
        TextureName,
        nullptr,
        _texture.GetAddressOf()
    ));
}

void DirectXApp::BuildVertexNormals()
{
    // Calculate vertex normals
    int vertexContributingCount[ARRAYSIZE(vertices)];
    for (int i = 0; i < ARRAYSIZE(vertices); i++)
    {
        vertexContributingCount[i] = 0;
    }
    int polygonCount = ARRAYSIZE(indices) / 3;
    for (int i = 0; i < polygonCount; i++)
    {
        int index0 = indices[i * 3];
        int index1 = indices[i * 3 + 1];
        int index2 = indices[i * 3 + 2];
        Vector3 u = vertices[index1].Position - vertices[index0].Position;
        Vector3 v = vertices[index2].Position - vertices[index0].Position;
        Vector3 normal = u.Cross(v);
        vertices[index0].Normal += normal;
        vertexContributingCount[index0]++;
        vertices[index1].Normal += normal;
        vertexContributingCount[index1]++;
        vertices[index2].Normal += normal;
        vertexContributingCount[index2]++;
    }
    // Now divide the vertex normals by the contributing counts and normalise
    for (int i = 0; i < ARRAYSIZE(vertices); i++)
    {
        vertices[i].Normal /= (float)vertexContributingCount[i];
        vertices[i].Normal.Normalize();
    }
}
```

DirectXApp.h

```
#pragma once
#include <vector>
#include "Framework.h"
#include "DirectXCore.h"
#include "SimpleMath.h"

using namespace SimpleMath;

class DirectXApp : public Framework
{
public:
      DirectXApp();

      bool Initialise();
      void Update();
      void Render();
      void OnResize(WPARAM wParam);
      void Shutdown();

private:
      ComPtr<ID3D11Device>            _device;
      ComPtr<ID3D11DeviceContext>     _deviceContext;
      ComPtr<IDXGISwapChain>          _swapChain;
      ComPtr<ID3D11Texture2D>         _depthStencilBuffer;
      ComPtr<ID3D11RenderTargetView>  _renderTargetView;
      ComPtr<ID3D11DepthStencilView>  _depthStencilView;

      ComPtr<ID3D11ShaderResourceView> _texture;;

      D3D11_VIEWPORT                  _screenViewport{ 0 };

      ComPtr<ID3D11Buffer>            _vertexBuffer;
      ComPtr<ID3D11Buffer>            _indexBuffer;

      ComPtr<ID3DBlob>                _vertexShaderByteCode = nullptr;
      ComPtr<ID3DBlob>                _pixelShaderByteCode = nullptr;
      ComPtr<ID3D11VertexShader>      _vertexShader;
      ComPtr<ID3D11PixelShader>       _pixelShader;
      ComPtr<ID3D11InputLayout>       _layout;
      ComPtr<ID3D11Buffer>            _constantBuffer;

      ComPtr<ID3D11RasterizerState>  _rasteriserState;

      Vector3                         _eyePosition;
      Vector3                         _focalPointPosition;
      Vector3                         _upVector;

      Matrix                          _worldTransformation;
      Matrix                          _viewTransformation;
      Matrix                          _projectionTransformation;

      int                             _rotationAngle{ 0 };

      bool GetDeviceAndSwapChain();
      void BuildGeometryBuffers();
      void BuildShaders();
      void BuildVertexLayout();
      void BuildConstantBuffer();
      void BuildRasteriserState();
      void BuildTexture();
      void BuildVertexNormals();
};
```

Geometry.h

```
#pragma once

constexpr auto ShaderFileName   = L"shader.hlsl";
constexpr auto VertexShaderName = "VS";
constexpr auto PixelShaderName  = "PS";
constexpr auto TextureName      = L"Woodbox.bmp";

// Format of the constant buffer. This must match the format of the
// cbuffer structure in the shader

struct CBuffer
{
       Matrix       WorldViewProjection;
       Matrix       World;
       Vector4      AmbientLightColour;
       Vector4      DirectionalLightColour;
       Vector4      DirectionalLightVector;
};

// Structure of a single vertex.  This must match the
// structure of the input vertex in the shader

struct Vertex
{
       Vector3            Position;
       Vector3            Normal;
       Vector2            TextureCoordinate;
};

// The description of the vertex that is passed to CreateInputLayout.  This must
// match the format of the vertex above and the format of the input vertex in the shader

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, D3D11_APPEND_ALIGNED_ELEMENT,
                    D3D11_INPUT_PER_VERTEX_DATA, 0 },
       { "TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT, 0, D3D11_APPEND_ALIGNED_ELEMENT,
                    D3D11_INPUT_PER_VERTEX_DATA, 0 }
};

// This example uses hard-coded vertices and indices for a cube. Usually, you will load the
verticesa and indices from a model file.
// We will see this later in the module.


Vertex vertices[] =
{
       { Vector3(-1.0f, -1.0f, 1.0f), Vector3(0, 0, 0), Vector2(0.0f, 0.0f) },    // side 1
       { Vector3(1.0f, -1.0f, 1.0f), Vector3(0, 0, 0), Vector2(0.0f, 1.0f)  },
       { Vector3(-1.0f, 1.0f, 1.0f), Vector3(0, 0, 0), Vector2(1.0f, 0.0f)  },
       { Vector3(1.0f, 1.0f, 1.0f), Vector3(0, 0, 0), Vector2(1.0f, 1.0f)  },

       { Vector3(-1.0f, -1.0f, -1.0f), Vector3(0, 0, 0), Vector2(0.0f, 0.0f)  },    // side 2
       { Vector3(-1.0f, 1.0f, -1.0f), Vector3(0, 0, 0), Vector2(0.0f, 1.0f)  },
       { Vector3(1.0f, -1.0f, -1.0f), Vector3(0, 0, 0), Vector2(1.0f, 0.0f)  },
       { Vector3(1.0f, 1.0f, -1.0f), Vector3(0, 0, 0), Vector2(1.0f, 1.0f)  },

       { Vector3(-1.0f, 1.0f, -1.0f), Vector3(0, 0, 0), Vector2(0.0f, 0.0f)  },    // side 3
       { Vector3(-1.0f, 1.0f, 1.0f), Vector3(0, 0, 0), Vector2(0.0f, 1.0f)  },
       { Vector3(1.0f, 1.0f, -1.0f), Vector3(0, 0, 0), Vector2(1.0f, 0.0f)  },
       { Vector3(1.0f, 1.0f, 1.0f), Vector3(0, 0, 0), Vector2(1.0f, 1.0f)  },
      { Vector3(-1.0f, -1.0f, -1.0f), Vector3(0, 0, 0), Vector2(0.0f, 0.0f)  },    // side 4
      { Vector3(1.0f, -1.0f, -1.0f), Vector3(0, 0, 0), Vector2(0.0f, 1.0f)  },
      { Vector3(-1.0f, -1.0f, 1.0f), Vector3(0, 0, 0), Vector2(1.0f, 0.0f)  },
      { Vector3(1.0f, -1.0f, 1.0f), Vector3(0, 0, 0), Vector2(1.0f, 1.0f)  },

      { Vector3(1.0f, -1.0f, -1.0f), Vector3(0, 0, 0), Vector2(0.0f, 0.0f)  },    // side 5
      { Vector3(1.0f, 1.0f, -1.0f), Vector3(0, 0, 0), Vector2(0.0f, 1.0f)  },
      { Vector3(1.0f, -1.0f, 1.0f), Vector3(0, 0, 0), Vector2(1.0f, 0.0f)  },
      { Vector3(1.0f, 1.0f, 1.0f), Vector3(0, 0, 0), Vector2(1.0f, 1.0f)  },

      { Vector3(-1.0f, -1.0f, -1.0f), Vector3(0, 0, 0), Vector2(0.0f, 0.0f)  },    // side 6
      { Vector3(-1.0f, -1.0f, 1.0f), Vector3(0, 0, 0), Vector2(0.0f, 1.0f)  },
      { Vector3(-1.0f, 1.0f, -1.0f), Vector3(0, 0, 0), Vector2(1.0f, 0.0f)  },
      { Vector3(-1.0f, 1.0f, 1.0f), Vector3(0, 0, 0), Vector2(1.0f, 1.0f)  }
};

UINT indices[] = {
                                      0, 1, 2,       // side 1
                                      2, 1, 3,
                                      4, 5, 6,       // side 2
                                      6, 5, 7,
                                      8, 9, 10,      // side 3
                                      10, 9, 11,
                                      12, 13, 14,    // side 4
                                      14, 13, 15,
                                      16, 17, 18,    // side 5
                                      18, 17, 19,
                                      20, 21, 22,    // side 6
                                      22, 21, 23,

};
```

Shader.hlsl:

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

Texture2D Texture;
SamplerState ss;

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

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 = -directionalLightVector;
	float4 adjustedNormal = normalize(mul(worldTransformation, float4(vin.Normal, 0.0f)));
	float diffuseBrightness = saturate(dot(adjustedNormal, vectorBackToLight));
	vout.Colour = saturate(ambientLightColour + diffuseBrightness * directionalLightColour);

	vout.TexCoord = vin.TexCoord;
	return vout;
}

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