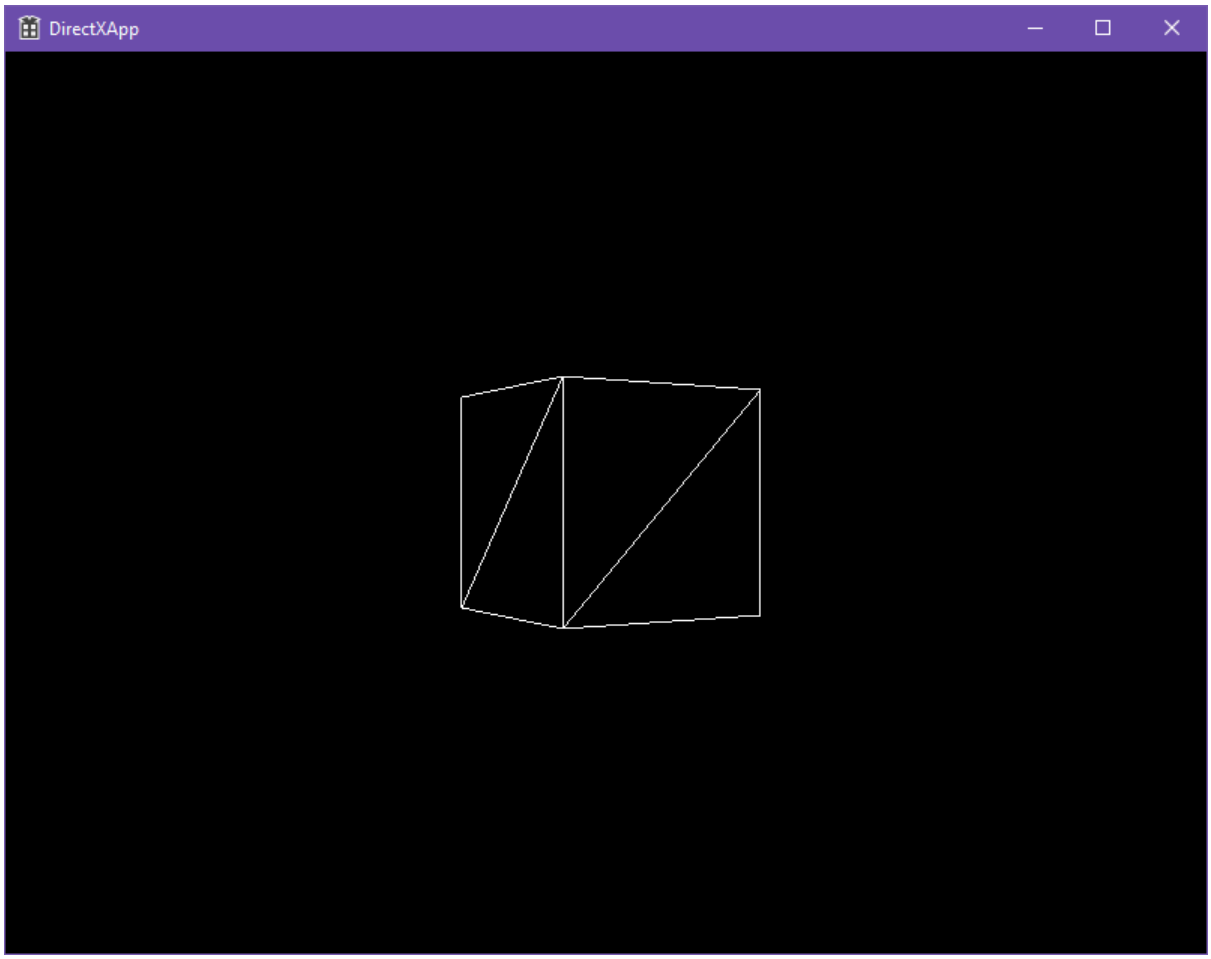
Graphics Part II: Code Tutorial for Week 2

Dr P. Perakis

Preparation

The starting point for all of the exercises this week should be the example provided for you in `DirectX_Cube_Week2.zip`. (Exercise_02_0)

This rotates the wireframe cube around the Y axis as you can see below.



Introduction

Last week, you saw the rendering of a wireframe model and experimented with performing some transformations on it. This week, we will start looking at making the model more solid.

Backface Culling

Although the cube is being rendered as a wireframe model, it still appears to be somewhat solid because you are not seeing all of the polygons in the model. You only see the polygons you would expect to see, not the ones that are hidden behind them. This is because DirectX is performing *backface culling*, that is, not displaying the polygons that should not be visible from the position of the camera.

To see what backface culling is doing, you can turn it off and see the result. We can do this by changing a value in the `D3D11_RASTERIZER_DESC` structure created in the `BuildRasteriserState` method and

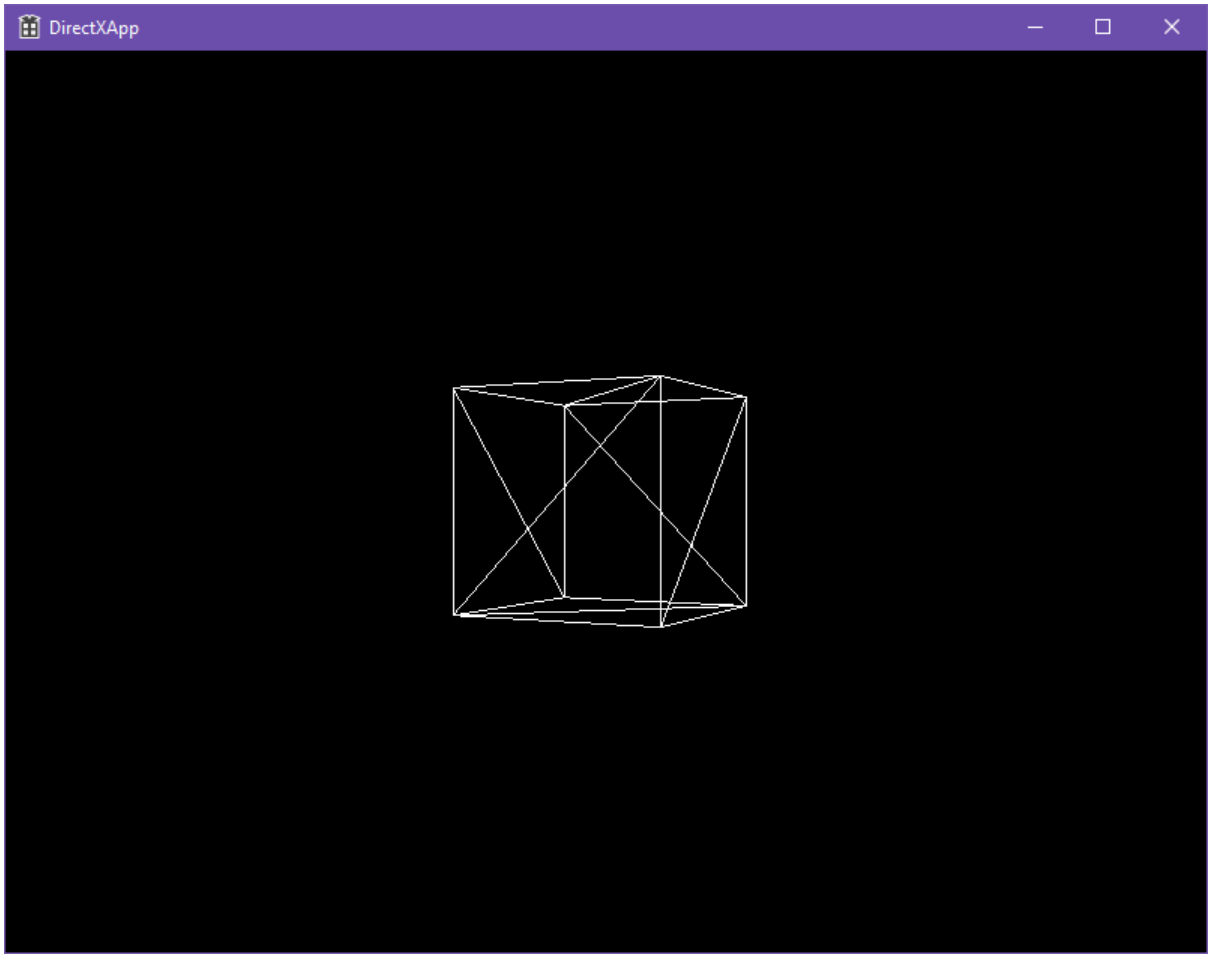
passed to the `CreateRasterizerState` function. Inside `BuildRasteriserState()` find the following line:

```
rasteriserDesc.CullMode = D3D11_CULL_BACK;
```

and change it to:

```
rasteriserDesc.CullMode = D3D11_CULL_NONE;
```

Rebuild the solution and run it. You should now see something like the following:



As you can see, the polygon now appears to be transparent, that is, you can see all of the polygons. This is very rarely what we want, so set the value of the `CullMode` back to `D3D11_CULL_BACK`.

Doing Solid Shading

Now let's make this a solid white cube. We can do this by changing a different value in the `D3D11_RASTERIZER_DESC` structure created in the `BuildRasteriserState` method. Inside `BuildRasteriserState()` find the following line:

```
rasteriserDesc.FillMode = D3D11_FILL_WIREFRAME;
```

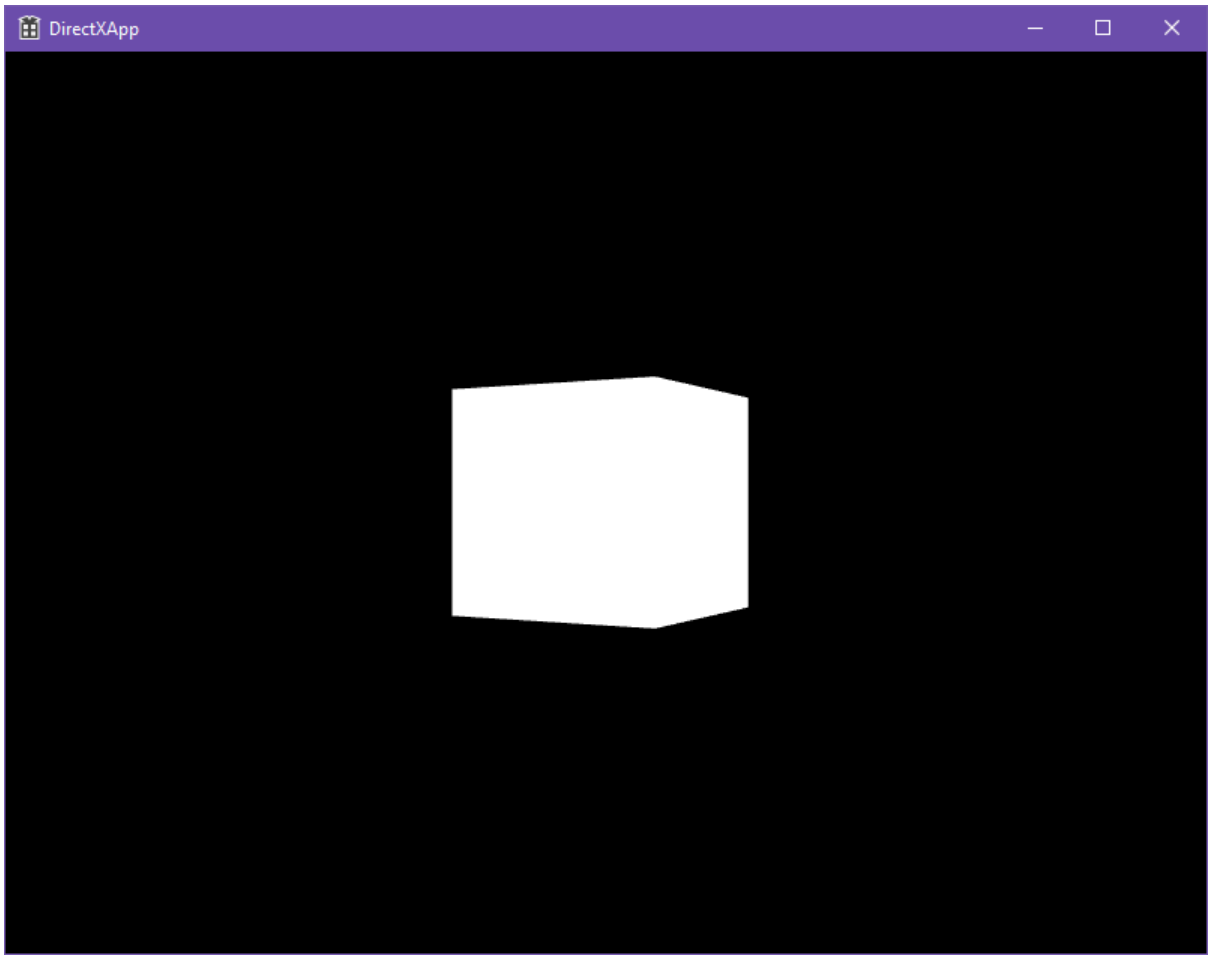
and change it to:

```
rasteriserDesc.FillMode = D3D11_FILL_SOLID;
```

The final code for `BuildRasteriserState()` is: (Exercise_02_1)

```
void DirectXApp::BuildRasteriserState()
{	// Set default and wireframe rasteriser states
	D3D11_RASTERIZER_DESC rasteriserDesc;
	// The following tells the rasteriser to apply backfece culling.  For not doing
so, set it to D3D11_CULL_NONE
	rasteriserDesc.CullMode = D3D11_CULL_BACK;
	//rasteriserDesc.CullMode = D3D11_CULL_NONE;
	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;
	// The following tells the rasteriser to draw a wireframe model.  For solid
models, set it to D3D11_FILL_SOLID
	//rasteriserDesc.FillMode = D3D11_FILL_WIREFRAME;
	rasteriserDesc.FillMode = D3D11_FILL_SOLID;
	ThrowIfFailed(_device->CreateRasterizerState(&rasteriserDesc,
_rasteriserState.GetAddressOf()));
}
```

If you rebuild your solution and run the resulting code, you should see something like the following:



Although the cube is now rendered as a solid object, you cannot see any definition of the cube.

We will revisit this when we look at the impact of lighting on an object next week, but for now we will change the cube so that each vertex has a different colour.

Colouring the Cube

We can now make the vertices to all have different colours and colours are blended between the vertices. To make this change, we need to update the structures in `Geometry.h` and also update the shader. (Exercise_02_2)

The input vertices will be changed to include the colour, the vertex shader will be changed to copy the colour from the input vertex to the output vertex and the pixel shader will be changed to return the colour supplied in the input parameter.

Changes to Geometry.h

1. Change the Vertex structure to add an additional field that provides the colour for the vertex.

Your Vertex structure should now look like the following:

```
struct Vertex
{
	Vector4			Position;
	Vector4			Colour;
};
```

We are specifying the colour as a Vector4, that is, four floating point values that represent the red, green, blue and alpha (opacity) values.

2. Now change the vertexDesc array to add the colour. This array tells DirectX how the input vertex structure is defined and how it should map on to the input vertex structure defined in the shader.

Change it to be as follows:

```
D3D11_INPUT_ELEMENT_DESC vertexDesc[] =
{
	{ "POSITION", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0, 0,
	D3D11_INPUT_PER_VERTEX_DATA, 0 },
	{ "COLOR", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0, D3D11_APPEND_ALIGNED_ELEMENT,
	D3D11_INPUT_PER_VERTEX_DATA, 0 }
};
```

The important thing to note is that this array must always match the format of the vertex structures in your C++ code and in the shader exactly.

3. Now you need to update the vertices array so that each vertex in the array includes both a position and a colour. Each colour should be represented as a Vector4 which specifies the colour for that vertex. We could specify the colours using some predefined constants that are in the DirectX header files, or as four separate RGBA float values.

For example, you may update the Cube structure as follows:

```
Vertex Cube_vertices[] =
{
	{ Vector4(-1.0f, -1.0f, -1.0f, 1.0f), Vector4(1.0f, 0.0f, 0.0f, 1.0f) }, //0
	{ Vector4(-1.0f, +1.0f, -1.0f, 1.0f), Vector4(0.0f, 1.0f, 0.0f, 1.0f) }, //1
	{ Vector4(+1.0f, +1.0f, -1.0f, 1.0f), Vector4(0.0f, 0.0f, 1.0f, 1.0f) }, //2
	{ Vector4(+1.0f, -1.0f, -1.0f, 1.0f), Vector4(0.0f, 1.0f, 0.0f, 1.0f) }, //3
```

```
        { Vector4(-1.0f, -1.0f, +1.0f, 1.0f), Vector4(0.0f, 1.0f, 0.0f, 1.0f) }, //4
        { Vector4(-1.0f, +1.0f, +1.0f, 1.0f), Vector4(0.0f, 0.0f, 1.0f, 1.0f) }, //5
        { Vector4(+1.0f, +1.0f, +1.0f, 1.0f), Vector4(0.0f, 1.0f, 0.0f, 1.0f) }, //6
        { Vector4(+1.0f, -1.0f, +1.0f, 1.0f), Vector4(1.0f, 0.0f, 0.0f, 1.0f) }  //7
};
```

Changes to shader.hlsl

1. Change the `VertexIn` structure to add the colour field as follows:

```
struct VertexIn
{
        float4 InputPosition : POSITION;
        float4 Colour        : COLOR;
};
```

You can see that the names after the colons are the semantic names used in the Input layout structure earlier.

2. Change the `vertexOut` structure to also add the colour:

```
struct VertexOut
{
        float4 OutputPosition       : SV_POSITION;
        float4 Colour               : COLOR;
};
```

3. Add a line to the vertex shader to copy the input colour to the output vertex:

```
VertexOut VS(VertexIn vin)
{
        VertexOut vout;

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

        vout.Colour = vin.Colour;

    return vout;
}
```

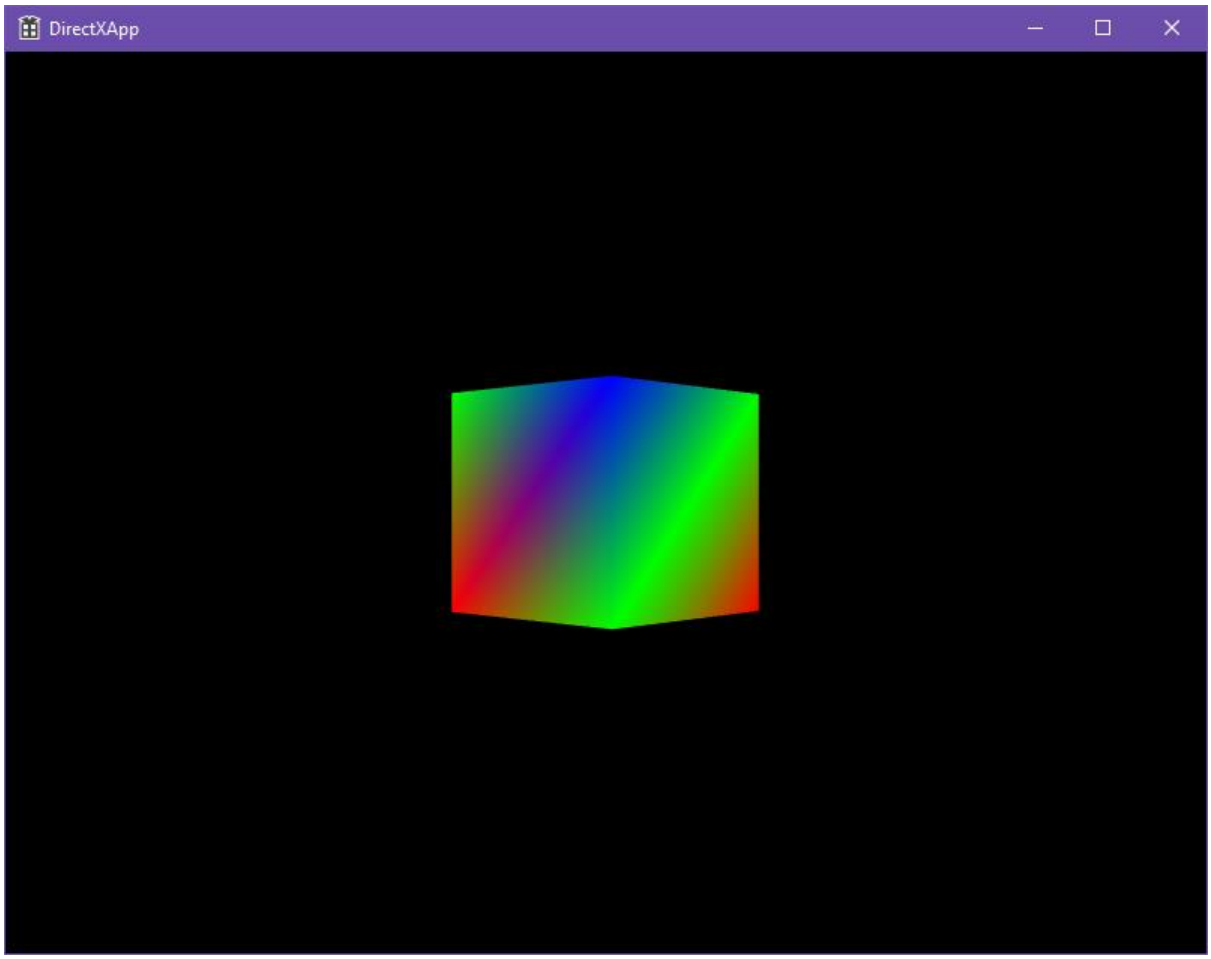
4. Finally, change the pixel shader so that it returns the colour from the input structure.

```
float4 PS(VertexOut pin) : SV_Target
{
        //return float4(1.0f, 1.0f, 1.0f, 1.0f);
        return pin.Colour;
}
```

Note. Make sure you save your `shader.hlsl` file every time you change it. Although C++ files are automatically saved when you build your solution, the shader is not built until run-time and is not automatically saved when you make changes.

Now you should be able to run the resulting solution and see a rotating cube that contains blended colours. You might be wondering at this stage about what is going on here. Colours are only specified at the vertices, but something is causing the colours between the vertices on each polygon to be

blended from the colours at the vertices. This is due to the linear interpolation being done by the rasterization stage.



Exercise

Modify the program so that instead of drawing just one cube, it draws a cube and a square-bottomed pyramid. This will take more work. A square-bottomed pyramid consists of two triangles on its base and one on each side. Just like the cube above, use a different colour for each vertex in the pyramid.

You will need to create another vertex buffer and another index buffer. You should not need to change the constant buffer or the shader. Make the pyramid rotate around the cube. (Exercise_02_3).

1. Declaring the Pyramid

You were given a Cube as a starting point. You have to declare accordingly a Pyramid in `Geometry.h`:

```
Vertex Pyramid_vertices[] =
{
	{ Vector4( 0.0f, +1.0f,  0.0f, 1.0f), Vector4(1.0f, 0.0f, 0.0f, 1.0f) }, //0
	{ Vector4(-1.0f, -1.0f, -1.0f, 1.0f), Vector4(0.0f, 1.0f, 0.0f, 1.0f) }, //1
	{ Vector4(+1.0f, -1.0f, -1.0f, 1.0f), Vector4(1.0f, 1.0f, 0.0f, 1.0f) }, //2
	{ Vector4(+1.0f, -1.0f, +1.0f, 1.0f), Vector4(0.0f, 0.0f, 1.0f, 1.0f) }, //3
	{ Vector4(-1.0f, -1.0f, +1.0f, 1.0f), Vector4(0.0f, 1.0f, 1.0f, 1.0f) }  //4
};

UINT Pyramid_indices[] = {
	// front face
	0, 2, 1,
	// back face
	0, 4, 3,
	// left face
	0, 1, 4,
	// right face
	0, 3, 2,
	// bottom face
	4, 1, 2,
	4, 2, 3
};
```

2. Updating the DirectXApp Class

You were given as a starting point a `DirectXApp` class. You have to modify this class in `DiractXApp.h` for keeping the parameters of the two models.

First for the two models' vertices and index structures:

```
//ComPtr<ID3D11Buffer>		_vertexBuffer;
//ComPtr<ID3D11Buffer>		_indexBuffer;
ComPtr<ID3D11Buffer>		_cube_vertexBuffer;
ComPtr<ID3D11Buffer>		_cube_indexBuffer;
ComPtr<ID3D11Buffer>		_pyramid_vertexBuffer;
ComPtr<ID3D11Buffer>		_pyramid_indexBuffer;
```

And secondly for the two models' transformations:

```
//Matrix			_worldTransformation;
Matrix				_worldTransformation1;
Matrix				_worldTransformation2;
```

3. Updating the Geometry Buffers for the Cube and Pyramid

You were given as a starting point an empty `DirectXApp::Update()` function. You have to modify this function for setting up the buffer of the two models:

```
void DirectXApp::BuildGeometryBuffers()
{
	// This method uses the arrays defined in Geometry.h
	//
	//Declare and initialize Buffers
	D3D11_BUFFER_DESC vertexBufferDescriptor = { 0 };
	vertexBufferDescriptor.Usage = D3D11_USAGE_IMMUTABLE;
	vertexBufferDescriptor.BindFlags = D3D11_BIND_VERTEX_BUFFER;
	vertexBufferDescriptor.CPUAccessFlags = 0;
	vertexBufferDescriptor.MiscFlags = 0;
	vertexBufferDescriptor.StructureByteStride = 0;

	D3D11_SUBRESOURCE_DATA vertexInitialisationData = { 0 };

	D3D11_BUFFER_DESC indexBufferDescriptor = { 0 };
	indexBufferDescriptor.Usage = D3D11_USAGE_IMMUTABLE;
	indexBufferDescriptor.BindFlags = D3D11_BIND_INDEX_BUFFER;
	indexBufferDescriptor.CPUAccessFlags = 0;
	indexBufferDescriptor.MiscFlags = 0;
	indexBufferDescriptor.StructureByteStride = 0;

	D3D11_SUBRESOURCE_DATA indexInitialisationData = { 0 };

	// Cube
	// Setup the structure that specifies how big the vertex buffer should be
	vertexBufferDescriptor.ByteWidth = sizeof(Vertex) * ARRAYSIZE(Cube_vertices);

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

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

	// Setup the structure that specifies how big the index buffer should be
	indexBufferDescriptor.ByteWidth = sizeof(UINT) * ARRAYSIZE(Cube_indices);


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

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

	// Pyramid
	// Setup the structure that specifies how big the vertex buffer should be
	vertexBufferDescriptor.ByteWidth = sizeof(Vertex) *
ARRAYSIZE(Pyramid_vertices);

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

	// and create the vertex buffer
```

```
	ThrowIfFailed(_device->CreateBuffer(&vertexBufferDescriptor,
&vertexInitialisationData, _pyramid_vertexBuffer.GetAddressOf()));

	// Setup the structure that specifies how big the index buffer should be
	indexBufferDescriptor.ByteWidth = sizeof(UINT) * ARRAYSIZE(Pyramid_indices);

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

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

4. Modifications for rendering the two models

You were given as a starting point the `DirectXApp::Render()` function. You have to modify this function accordingly, for rendering the two models separately:

```
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);


	Matrix completeTransformation;
	CBuffer constantBuffer;

	// Specify the distance between vertices and the starting point in the vertex
buffer
	UINT stride = sizeof(Vertex);
	UINT offset = 0;

	// Set rendering commons
	// Set the constant buffer. Note the layout of the constant buffer must match
that in the shader
	_deviceContext->VSSetConstantBuffers(0, 1, _constantBuffer.GetAddressOf());

	// 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());
```

```
	//Cube
	// Calculate the world x view x projection transformation
	completeTransformation = _worldTransformation1 * _viewTransformation *
_projectionTransformation;

	// Update the constant buffer.
	constantBuffer.WorldViewProjection = completeTransformation;
	_deviceContext->UpdateSubresource(_constantBuffer.Get(), 0, 0, &constantBuffer,
0, 0);

	// Now render the cube

	// Set the vertex buffer and index buffer we are going to use
	_deviceContext->IASetVertexBuffers(0, 1, _cube_vertexBuffer.GetAddressOf(),
&stride, &offset);
	_deviceContext->IASetIndexBuffer(_cube_indexBuffer.Get(), DXGI_FORMAT_R32_UINT,
0);

	// Now draw the object
	_deviceContext->DrawIndexed(ARRAYSIZE(Cube_indices), 0, 0);


	//Pyramid
	// Calculate the world x view x projection transformation
	completeTransformation = _worldTransformation2 * _viewTransformation *
_projectionTransformation;

	// Update the constant buffer.
	constantBuffer.WorldViewProjection = completeTransformation;
	_deviceContext->UpdateSubresource(_constantBuffer.Get(), 0, 0, &constantBuffer,
0, 0);


	// Now render the pyramid

	// Set the vertex buffer and index buffer we are going to use
	_deviceContext->IASetVertexBuffers(0, 1, _pyramid_vertexBuffer.GetAddressOf(),
&stride, &offset);
	_deviceContext->IASetIndexBuffer(_pyramid_indexBuffer.Get(),
DXGI_FORMAT_R32_UINT, 0);

	// Now draw the object
	_deviceContext->DrawIndexed(ARRAYSIZE(Pyramid_indices), 0, 0);

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

Note that now you have two different vertex and indices structures when calling the rendering and the draw function `_deviceContext->DrawIndexed()`. *You are also applying two different model transformations* `_worldTransformation1` *and* `_worldTransformation2` *in the* `completeTransformation` *parameter of the constant buffer.*

5. Moving the Models and setting the Camera

You were given as a starting point a `DirectXApp::Update()` function.

You have to modify this function for animating with different transformations `_worldTransformation1` and `_worldTransformation2` the two models, and also for moving the camera to a new position:

```
void DirectXApp::Update()
{
	Matrix RotationZ, RotationY, TranslationX, Scale;

	//Cube
	RotationY = Matrix::CreateRotationY(_rotationAngle * XM_PI / 180.0f);
	_worldTransformation1 = RotationY;

	//Pyramid
	RotationY = Matrix::CreateRotationY(-0.5f * _rotationAngle * XM_PI / 180.0f);
	TranslationX = Matrix::CreateTranslation(4.0f, 0.0f, 0);
	_worldTransformation2 = TranslationX * RotationY;

	_rotationAngle = (_rotationAngle + 1) % 360;

	//Camera
	int cameraAngleX = 30;
	float cameraDistance = 15.0f;
	_eyePosition.y = cameraDistance * sin(cameraAngleX * XM_PI / 180.0f);
	_eyePosition.z = -cameraDistance * cos(cameraAngleX * XM_PI / 180.0f);
}
```

Note that you have moved the camera for a better view of the scene. The camera keeps looking at the origin but moved at a position 15 units away from the origin and at an angle of +30 degrees from ground level, showing the top of the models.

6. The output of the executable

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