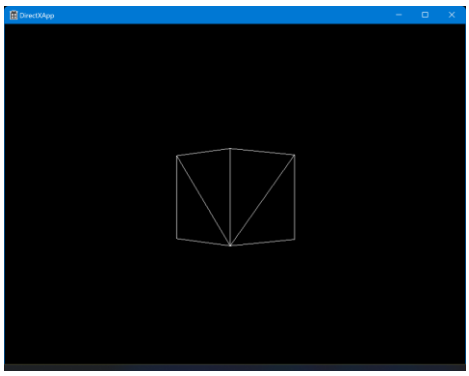
Graphics – Tutorial for Part 2, Week 2

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 a making the model more solid. There is only so far we can go this week until we look at the impact of lighting, but it will allow the introduction of some more new concepts.

The starting point for the work this week can be found in the DirectX_Cube_Week2.zip file. This rotates the wireframe cube around the Y axis as you can see below.



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.

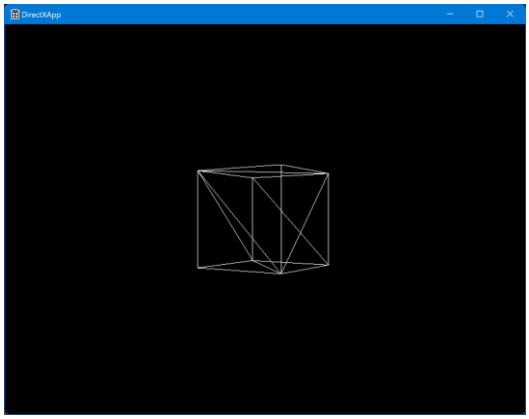
To see what backface culling is doing, you can turn it off and see the result. We can do this by changing a value in the D3D11_RASTERIZER_DESC structure created in the BuildRasteriserState method and passed to the CreateRasterizerState function. Find the following line:

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

and change it to:

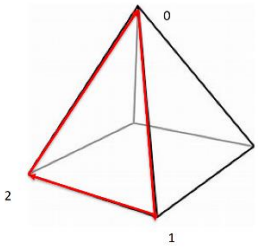
```
rasteriserDesc.CullMode = D3D11_CULL_NONE;
```

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

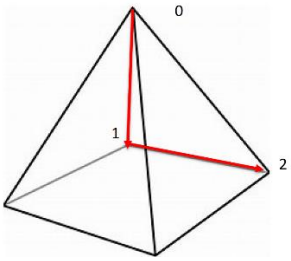


As you can see, the polygon now appears to be transparent, that is, you can see all of the polygons. This is very rarely what we want, so set the value of the CullMode back to D3D11_CULL_BACK.

So how does DirectX know which polygons to display and which ones not to display? The answer lies in the *winding ord*er of the indexes for the polygon. The default winding order for DirectX is clockwise. This is illustrated in the diagram below. The indices for the vertices in the polygon should be put in the list in the order 0, 1, 2. If you put them in the list in the order 0, 2, 1, this would be counter-clockwise winding. In DirectX, any polygon that has clockwise order is displayed



If the object is now rotated through 180 degrees, that polygon will no longer be visible since the winding order is now counter-clockwise.



You can see the impact of this by changing the winding order of one of the polygons in the cube. Change the order of the 3 indices for the first polygon in the indices array (in Geometry.h) from 0, 1, 2 to 0, 2, 1. Now rebuild and run the program. You will notice that one of the polygons is now not being displayed when it should be and is being displayed when it should not be.

Put the winding order for that polygon back to the correct order before going to the next stage.

Doing Solid Shading

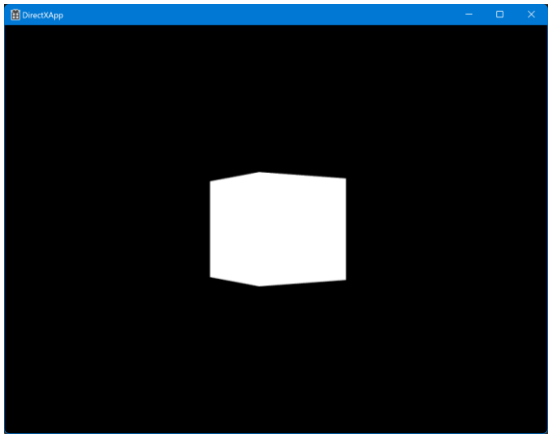
Now let's make this a solid white cube. We can do this by changing a different value in the D3D11_RASTERIZER_DESC structure created in the BuildRasteriserState method. Find the following line:

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

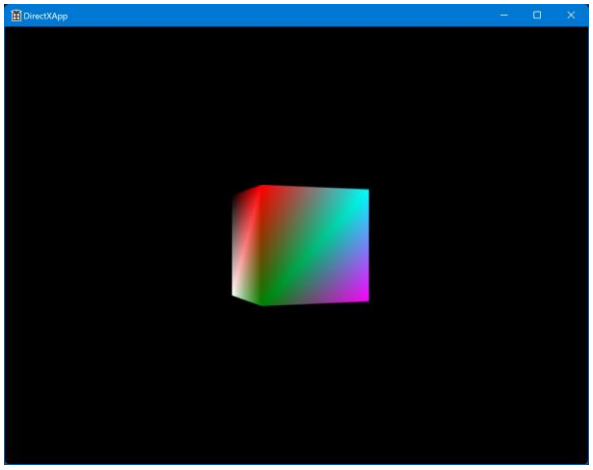
and change it to:

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

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 is a different colour. We will end up with something like the following:



You can see that the vertices are all 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. 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 }
};
```

It is worthwhile taking a brief moment to describe what the lines in this array mean and look specifically at certain fields.

For each entry, the first value is the *semantic* name for the field, that is, it defines what it means. These map on to the semantic names specified in the shader. So the first entry in the array specified that the first iield in the Vertex structure represents the position of the vertex, the second entry indicates that it specifies a colour (note the American spelling of Color for the semantic names) and so on. There are only a few semantic names defined and details can be found at https://learn.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl-semantics

The third field specifies the format of the data in the entry in the vertex structure. The rather cumbersome constant DXGI_FORMAT_R32G32B32A32_FLOAT really just indicates that this is four 32-bit floating point values. Details of the different format specifiers can be found at https://learn.microsoft.com/en-us/windows/win32/api/dxgiformat/ne-dxgiformat-dxgi_format. In practice, you just tend to use a few of these though.

Finally, for now, the fifth field specifies the offset of the field from the beginning of the structure (in bytes). So you can see for the position that this is 0. For subsequent entries, we could work out the offset in bytes ourselves. However, the easier route is to just specify the constant value D3D11_APPEND_ALIGNED_ELEMENT and let DirectX figure it out for itself.

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 as four separate float values, but it is easier to use some defined constants that are in the DirectX header files. For example,

```
Vector4(Colors::White)
```

is the same as specifying:

```
Vector4(1.0f, 1.0f, 1.0f, 1.0f)
```

Use the following colours for the eight vertices:

Colors::White
Colors::Black
Colors::Red
Colors::Green
Colors::Blue
Colors::Yellow
Colors::Cyan
Colors::Magenta

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:

```
vout.Colour = vin.Colour;
```

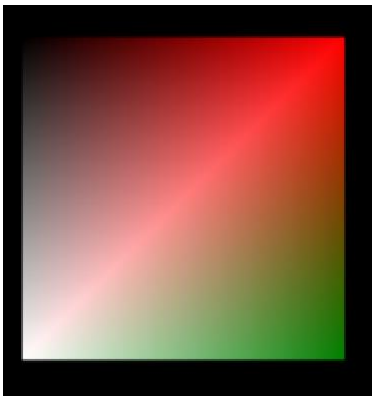
4. Finally, change the pixel shader so that it returns the colour from the input structure.

```
return pin.Colour;
```

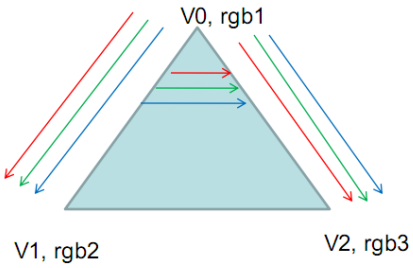
Note. Make sure you save your shader.hlsl file every time you change it. Although C++ files are automatically saved when you build your solution, the shader is not built until run-time and is not automatically saved when you make changes.

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.,



The image above is the front face of the cube, You can see that one polygon contains the white, black and red vertices and the other polygon contains the red, white and green vertices. Between the vertices in each polygon, the colours are being blended evenly across the polygon to give the gradient that you see. Each of the red, green and blue components of the colour are being linearly interpolated between the vertices as shown below:



When the pixel shader is called to render the pixel, it is the interpolated colour that is being passed in to the pixel shader in the input structure, not the colour at one of the vertices. Since our shader simply returns that colour, that is the colour that is used when the pixel is rendered.

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 polygons 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.

The end result should look something like the following (depending on what colours you choose for the vertices in the pyramid):