Tutorial 1: Direct3D 11 Basics

Summary

In this first tutorial, we will go through the elements necessary to create a minimal Direct3D 11 application. Every Direct3D 11 application must have these elements to function properly. The elements include setting up a window and a device object, and then displaying a color on the window.

Setting Up The Direct3D 11 Device

The first steps of creating the window and message loop are identical in Direct3D 9, Direct3D 10, and Direct3D 11. See Direct3D 10 Tutorial 00: Win32 Basics for an introduction to this process. Now that we have a window that is displaying, we can continue to set up a Direct3D 11 device. Setup is necessary if we are going to render any 3D scene. The first thing to do is to create three objects: a device, an immediate context, and a swap chain. The immediate context is a new object in Direct3D 11.

In Direct3D 10, the device object was used to perform both rendering and resource creation. In Direct3D 11, the immediate context is used by the application to perform rendering onto a buffer, and the device contains methods to create resources.

The swap chain is responsible for taking the buffer to which the device renders, and displaying the content on the actual monitor screen. The swap chain contains two or more buffers, mainly the front and the back. These are textures to which the device renders in order to display on the monitor. The front buffer is what is being presented currently to the user. This buffer is read-only and cannot be modified. The back buffer is the render target to which the device will draw. Once it finishes the drawing operation, the swap chain will present the backbuffer by swapping the two buffers. The back buffer becomes the front buffer, and vice versa.

To create the swap chain, we fill out a DXGI_SWAPCHAIN_DESC structure that describes the swap chain we are about to create. A few fields are worth mentioning. **BackBufferUsage** is a flag that tells the application how the back buffer will be used. In this case, we want to render to the back buffer, so we'll set **BackBufferUsage** to DXGI_USAGE_RENDER_TARGET_OUTPUT. The **OutputWindow** field represents the window that the swap chain will use to present images on the screen. SampleDesc is used to enable multi-sampling. Since this tutorial does not use multi-sampling, SampleDesc's Count is set to 1 and Quality to 0 to have multi-sampling disabled.

Once the description has been filled out, we can call the **D3D11CreateDeviceAndSwapChain** function to create both the device and the swap chain for us. The following is the code to create a device and a swap chain:

```
DXGI_SWAP_CHAIN_DESC sd;
ZeroMemory( &sd, sizeof(sd) );
sd.BufferCount = 1;
sd.BufferDesc.Width = 640;
sd.BufferDesc.Height = 480;
sd.BufferDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
sd.BufferDesc.RefreshRate.Numerator = 60;
sd.BufferDesc.RefreshRate.Denominator = 1;
sd.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;
sd.OutputWindow = g_hWnd;
sd.SampleDesc.Count = 1;
sd.SampleDesc.Quality = 0;
sd.Windowed = TRUE;

D3D11CreateDeviceAndSwapChain( NULL, D3D_DRIVER_TYPE_HARDWARE, NULL, 0, featureLevels,
                               numFeatureLevels, D3D11_SDK_VERSION, &sd, &g_pSwapChain,
                               &g_pd3dDevice, NULL, &g_pImmediateContext )
```

The next thing we need to do is to create a render target view. A render target view is a type of resource view in Direct3D 11. A resource view allows a resource to be bound to the graphics pipeline at a specific stage. Think of resource views as typecast in C. A chunk of raw memory in C can be cast to any data type. We can cast that chunk of memory to an array of integers, an array of floats, a structure, an array of structures, and so on. The raw memory itself is not very useful to us if we don't know its type. Direct3D 11 resource views act in a similar way. For instance, a 2D texture, analogous to the raw memory chunk, is the raw underlying resource. Once we have that resource we can create different resource views to bind that texture to different stages in the graphics pipeline with different formats: as a render target to which to render, as a depth stencil buffer that will receive depth information, or as a texture resource. Where C typecasts allow a memory chunk to be used in a different manner, so do Direct3D 11 resource views.

We need to create a render target view because we would like to bind the back buffer of our swap chain as a render target. This enables Direct3D 11 to render onto it. We first call **GetBuffer()** to get the back buffer object. Optionally, we can fill in a D3D11_RENDERTARGETVIEW_DESC structure that describes the render target view to be created. This description is normally the second parameter to **CreateRenderTargetView**. However, for these tutorials, the default render target view will suffice. The default render target view can be obtained by passing NULL as the second parameter. Once we have created the render target view, we can call **OMSetRenderTargets()** on the immediate context to bind it to the pipeline. This ensures the output that the pipeline renders gets written to the back buffer. The code to create and set the render target view is as follows:

```
ID3D11Texture2D *pBackBuffer;
g_pSwapChain->GetBuffer( 0, __uuidof( ID3D11Texture2D ), (LPVOID*)&pBackBuffer );
g_pd3dDevice->CreateRenderTargetView( pBackBuffer, NULL, &g_pRenderTargetView );
pBackBuffer->Release();
g_pImmediateContext->OMSetRenderTargets( 1, &g_pRenderTargetView, NULL );
```

The last thing we need to set up before Direct3D 11 can render is initialize the viewport. The viewport maps clip space coordinates, where X and Y range from -1 to 1 and Z ranges from 0 to 1, to render target space, sometimes known as pixel space. In Direct3D 9, if the application does not set up a viewport, a default viewport is set up to be the same size as the render target. In Direct3D 11, no viewport is set by default. Therefore, we must do so before we can see anything on the screen. Since we would like to use the entire render target for the output, we set the top left point to (0, 0) and width and height to be identical to the render target's size. To do this, use the following code:

```
D3D11_VIEWPORT vp;
vp.Width = (FLOAT)width;
vp.Height = (FLOAT)height;
vp.MinDepth = 0.0f;
vp.MaxDepth = 1.0f;
vp.TopLeftX = 0;
vp.TopLeftY = 0;
g_pImmediateContext->RSSetViewports( 1, &vp );
```

Modifying the Message Loop

We have set up the window and Direct3D 11 device, and we are ready to render. However, there is still a problem with our message loop: it uses **GetMessage()** to get messages. The problem with **GetMessage()** is that if there is no message in the queue for the application window, **GetMessage()** blocks and does not return until a message is available. Thus, instead of doing something like rendering, our application is waiting within **GetMessage()** when the message queue is empty. We can solve this problem by using **PeekMessage()** instead of **GetMessage()**. **PeekMessage()** can retrieve a message like **GetMessage()** does, but when there is no message waiting, **PeekMessage()** returns immediately instead of blocking. We can then take this time to do some rendering. The modified message loop, which uses **PeekMessage()**, looks like this:

```
MSG msg = {0};
while( WM_QUIT != msg.message )
{
    if( PeekMessage( &msg, NULL, 0, 0, PM_REMOVE ) )
    {
        TranslateMessage( &msg );
        DispatchMessage( &msg );
    }
    else
    {
        Render();  // Do some rendering
    }
}
```

The Rendering Code

Rendering is done in the **Render()** function. In this tutorial, we will render the simplest scene possible, which is to fill the screen with a single color. In Direct3D 11, an easy way to fill the render target with a single color is to use the immediate context's **ClearRenderTargetView()** method. First, we define an array of four floats that describe the color with which we would like to fill the screen. Then, we pass it to **ClearRenderTargetView()**. In this example, we choose a shade of blue. Once we fill our back buffer, we call the swap chain's **Present()** method to complete the rendering. **Present()** is responsible for displaying the swap chain's back buffer content onto the screen so that the user can see it. The **Render()** function looks like this:

```
 void Render()
{
    // Clear the backbuffer
    float ClearColor[4] = { 0.0f, 0.125f, 0.6f, 1.0f }; // RGBA
    g_pd3dDevice->ClearRenderTargetView( g_pRenderTargetView, ClearColor );

    g_pSwapChain->Present( 0, 0 );
}
```

Tutorial 2: Rendering a Triangle

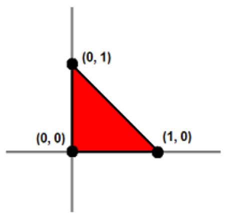
Summary

In the previous tutorial, we built a minimal Direct3D 11 application that outputs a single color to the window. In this tutorial, we will extend the application to render a single triangle on the screen. We will go through the process to set up the data structures associated with a triangle.
The outcome of this tutorial is a window with a triangle rendered to the center of the window.

Elements of a Triangle

A triangle is defined by its three points, also called vertices. A set of three vertices with unique positions define a unique triangle. In order for a GPU to render a triangle, we must tell it about the position of the triangle's three vertices. For a 2D example, let's say we wish to render a triangle such as that in figure 1. We would pass three vertices with the positions (0, 0) (0, 1) and (1, 0) to the GPU, and then the GPU has enough information to render the triangle that we want.

Figure 1. A triangle in 2D defined by its three vertices



So now we know that we must pass three positions to the GPU in order to render a triangle. How do we pass this information to the GPU? In Direct3D 11, vertex information such as position is stored in a buffer resource. A buffer that is used to store vertex information is called, not surprisingly, a vertex buffer. We must create a vertex buffer large enough for three vertices and fill it with the vertex positions. In Direct3D 11, the application must specify a buffer size in bytes when creating a buffer resource. We know the buffer has to be large enough for three vertices, but how many bytes does each vertex need? To answer that question requires an understanding of vertex layout.

Input Layout

A vertex has a position. More often than not, it also has other attributes as well, such as a normal, one or more colors, texture coordinates (used for texture mapping), and so on. Vertex layout defines how these attributes lie in memory: what data type each attribute uses, what size each attribute has, and the order of the attributes in memory. Because the attributes usually have different types, similar to the fields in a C structure, a vertex is usually represented by a structure. The size of the vertex is conveniently obtained from the size of the structure.
In this tutorial, we are only working with the position of the vertices. Therefore, we define our vertex structure with a single field of the type XMFLOAT3. This type is a vector of three floating-points components, which is typically the data type used for position in 3D.

```
struct SimpleVertex
{
    XMFLOAT3 Pos;  // Position
};
```

We now have a structure that represents our vertex. That takes care of storing vertex information in system memory in our application. However, when we feed the GPU the vertex buffer containing our vertices, we are just feeding it a chunk of memory. The GPU must also know about the vertex layout in order to extract correct attributes out from the buffer. To accomplish this requires the use of an input layout.

In Direct3D 11, an input layout is a Direct3D object that describes the structure of vertices in a way that can be understood by the GPU. Each vertex attribute can be described with the D3D11_INPUT_ELEMENT_DESC structure. An application defines an array of one or more D3D11_INPUT_ELEMENT_DESC, then uses that array to create the input layout object which describes the vertex as a whole. We will now look at the fields of D3D11_INPUT_ELEMENT_DESC in detail.

SemanticName	Semantic name is a string containing a word that describes the nature or purpose (or semantics) of this element. The word can be in any form that a C identifier can, and can be anything that we choose. For instance, a good semantic name for the vertex's position is POSITION. Semantic names are not case-sensitive.
SemanticIndex	Semantic index supplements semantic name. A vertex may have multiple attributes of the same nature. For example, it may have 2 sets of texture coordinates or 2 sets of colors. Instead of using semantic names that have numbers appended, such as "COLOR0" and "COLOR1", the two elements can share a single semantic name, "COLOR", with different semantic indices 0 and 1.
Format	Format defines the data type to be used for this element. For instance, a format of DXGI_FORMAT_R32G32B32_FLOAT has three 32-bit floating point numbers, making the element 12-byte long. A format of DXGI_FORMAT_R16G16B16A16_UINT has four 16-bit unsigned integers, making the element 8 bytes long.
InputSlot	As mentioned previously, a Direct3D 11 application passes vertex data to the GPU via the use of vertex buffer. In Direct3D 11, multiple vertex buffers can be fed to the GPU simultaneously, 16 to be exact. Each vertex buffer is bound to an input slot number ranging from 0 to 15. The InputSlot field tells the GPU which vertex buffer it should fetch for this element.
AlignedByteOffset	A vertex is stored in a vertex buffer, which is simply a chunk of memory. The AlignedByteOffset field tells the GPU the memory location to start fetching the data for this element.
InputSlotClass	This field usually has the value D3D11_INPUT_PER_VERTEX_DATA. When an application uses instancing, it can set an input layout's InputSlotClass to D3D11_INPUT_PER_INSTANCE_DATA to work with vertex buffer containing instance data. Instancing is an advanced Direct3D topic and will not be discussed here. For our tutorial, we will use D3D11_INPUT_PER_VERTEX_DATA exclusively.
InstanceDataStepRate	This field is used for instancing. Since we are not using instancing, this field is not used and must be set to 0.

Now we can define our D3D11_INPUT_ELEMENT_DESC array and create the input layout:

```
// Define the input layout
D3D11_INPUT_ELEMENT_DESC layout[] =
{
    { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0, D3D11_INPUT_PER_VERTEX_DATA, 0 },
};
UINT numElements = ARRAYSIZE(layout);
```

Vertex Layout

In the next tutorial, we will explain the technique object and the associated shaders. For now, we will just concentrate on creating the Direct3D 11 vertex layout object for the technique. However, we will learn that the vertex shaders are tightly coupled with this vertex layout. The reason is that creating a vertex layout object requires the vertex shader's input signature.

We use the ID3DBlob object returned from D3DX11CompileFromFile to retrieve the binary data that represents the input signature of the vertex shader. Once we have this data, we can call **ID3D11Device::CreateInputLayout()** to create a vertex layout object, and **ID3D11DeviceContext::IASetInputLayout()** to set it as the active vertex layout. The code to do all of that is shown below:

```
// Create the input layout
g_pd3dDevice->CreateInputLayout( layout, numElements, pVSBlob->GetBufferPointer(),
                                 pVSBlob->GetBufferSize(), &g_pVertexLayout );
```

```
// Set the input layout
g_pImmediateContext->IASetInputLayout( g_pVertexLayout );
```

Creating Vertex Buffer

One thing that we will also need to do during initialization is to create the vertex buffer that holds the vertex data. To create a vertex buffer in Direct3D 11, we fill in two structures, D3D11_BUFFER_DESC and D3D11_SUBRESOURCE_DATA, and then call **ID3D11Device::CreateBuffer()**. D3D11_BUFFER_DESC describes the vertex buffer object to be created, and D3D11_SUBRESOURCE_DATA describes the actual data that will be copied to the vertex buffer during creation. The creation and initialization of the vertex buffer is done at once so that we don't need to initialize the buffer later. The data that will be copied to the vertex buffer is vertices, an array of threeSimpleVertex structures. The coordinates in the vertices array are chosen so that we see a triangle in the middle of our application window when rendered with our shaders. After the vertex buffer is created, we can call **ID3D11DeviceContext::IASetVertexBuffers()** to bind it to the device. The complete code is shown here:

```
SimpleVertex vertices[] =
{
    XMFLOAT3( 0.0f, 0.5f, 0.5f ),
    XMFLOAT3( 0.5f, -0.5f, 0.5f ),
    XMFLOAT3( -0.5f, -0.5f, 0.5f ),
};

D3D11_BUFFER_DESC bd;
ZeroMemory( &bd, sizeof(bd) );
bd.Usage = D3D11_USAGE_DEFAULT;
bd.ByteWidth = sizeof( SimpleVertex ) * 3;
bd.BindFlags = D3D11_BIND_VERTEX_BUFFER;
bd.CPUAccessFlags = 0;
bd.MiscFlags = 0;
D3D11_SUBRESOURCE_DATA InitData;
ZeroMemory( &InitData, sizeof(InitData) );
InitData.pSysMem = vertices;
g_pd3dDevice->CreateBuffer( &bd, &InitData, &g_pVertexBuffer );

// Set vertex buffer
UINT stride = sizeof( SimpleVertex );
UINT offset = 0;
g_pImmediateContext->IASetVertexBuffers( 0, 1, &g_pVertexBuffer, &stride, &offset );
```

Primitive Topology

Primitive topology refers to how the GPU obtains the three vertices it requires to render a triangle. We discussed above that in order to render a single triangle, the application needs to send three vertices to the GPU. Therefore, the vertex buffer has three vertices in it. What if we want to render two triangles? One way is to send 6 vertices to the GPU. The first three vertices define the first triangle and the second three vertices define the second triangle. This topology is called a triangle list. Triangle lists have the advantage of being easy to understand, but in certain cases they are very inefficient. Such cases occur when successively rendered triangles share vertices. For instance, figure 3a left shows a square made up of two triangles: A B C and C B D. (By convention, triangles are typically defined by listing their vertices in clockwise order.) If we send these two triangles to the GPU using a triangle list, our vertex buffer would like this:

```
A B C C B D
```

Notice that B and C appear twice in the vertex buffer because they are shared by both triangles.

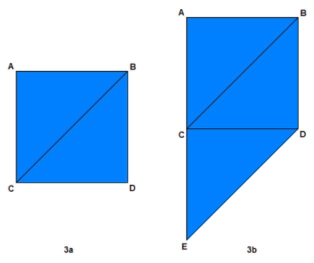


Figure 3a contains a square made up of two triangles; figure 3b contains a pentagonal shape made up of three triangles.
We can make the vertex buffer smaller if we can tell the GPU that when rendering the second triangle, instead of fetching all three vertices from the vertex buffer, use 2 of the vertices from the previous triangle and fetch only 1 vertex from the vertex buffer. As it turns out, this is supported by Direct3D, and the topology is called triangle strip. When rendering a triangle strip, the very first triangle is defined by the first three vertices in the vertex buffer. The next triangle is defined by the last two vertices of the previous triangle plus the next vertex in the vertex buffer. Taking the square in figure 3a as an example, using triangle strip, the vertex buffer would look like:

```
A B C D
```

The first three vertices, A B C, define the first triangle. The second triangle is defined by B and C, the last two vertices of the first triangle, plus D. Thus, by using the triangle strip topology, the vertex buffer size has gone from 6 vertices to 4 vertices. Similarly, for three triangles such as those in figure 3b, using triangle list would require a vertex buffer such as:

```
A B C C B D C D E
```

Using triangle strip, the size of the vertex buffer is dramatically reduced:

```
A B C D E
```

You may have noticed that in the triangle strip example, the second triangle is defined as B C D. These three vertices do not form a clockwise order. This is a natural phenomenon from using triangle strips. To overcome this, the GPU automatically swaps the order of the two vertices coming from the previous triangle. It only does this for the second triangle, fourth triangle, sixth triangle, eighth triangle, and so on. This ensures that every triangle is defined by vertices in the correct winding order (clockwise, in this case). Besides triangle list and triangle strip, Direct3D 11 supports many other types of primitive topology. We will not discuss them in this tutorial.
In our code, we have one triangle, so it doesn't really matter what we specify. However, we must specify something, so we chose a triangle list.

```
// Set primitive topology
g_pImmediateContext->IASetPrimitiveTopology( D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST );
```

Rendering the Triangle

The final item missing is the code that does the actual rendering of the triangle. We have created two shaders to for rendering, the vertex shader and pixel shader. The vertex shader is responsible for transforming the individual vertices of the triangles to their correct locations. And the pixel shader is responsible for calculating the final output color for each pixel of the triangle. This is covered in more detail in the next tutorial. To use these shaders we must call **ID3D11DeviceContext::VSSetShader()** and **ID3D11DeviceContext::PSSetShader()** respectively. The last thing that we do is call **ID3D11DeviceContext::Draw()**, which commands the GPU to render using the current vertex buffer, vertex layout, and primitive topology. The first parameter to **Draw()** is the number of vertices to send to the GPU, and the second parameter is the index of the first vertex to begin sending. Because we are rendering one triangle and we are rendering from the beginning of the vertex buffer, we use 3 and 0 for the two parameters, respectively. The entire triangle-rendering code looks like the following:

```
// Render a triangle
g_pImmediateContext->VSSetShader( g_pVertexShader, NULL, 0 );
g_pImmediateContext->PSSetShader( g_pPixelShader, NULL, 0 );
g_pImmediateContext->Draw( 3, 0 );
```

Tutorial 3: Shaders and Effect System

Summary

In the previous tutorial, we set up a vertex buffer and passed one triangle to the GPU. Now, we will actually step through the graphics pipeline and look at how each stage works. The concept of shaders and the effect system will be explained.

Note that this tutorial shares the same source code as the previous one, but will emphasize a different section.

The Graphics Pipeline

In the previous tutorial, we set up the vertex buffer, and then we associated a vertex layout with a vertex shader. Now, we will explain what a shader is and how it works. To fully understand the individual shaders, we will take a step back and look at the whole graphical pipeline.

In Tutorial 2, when we called **VSSetShader()** and **PSSetShader()**, we actually bound our shader to a stage in the pipeline. Then, when we called **Draw**, we start processing the vertex data passed into the graphics pipeline. The following sections describe in detail what happens after the **Draw** command.

Shaders

In Direct3D 11, shaders reside in different stages of the graphics pipeline. They are short programs that, executed by the GPU, take certain input data, process that data, and then output the result to the next stage of the pipeline. Direct3D 11 supports three basic types of shaders: vertex shader, geometry shader, and pixel shader. A vertex shader takes a vertex as input. It is run once for every vertex passed to the GPU via vertex buffers. A geometry shader takes a primitive as input, and is run once for every primitive passed to the GPU. A primitive is a point, a line, or a triangle. A pixel shader takes a pixel (or sometimes called a fragment) as input, and is run once for each pixel of a primitive that we wish to render. Together, vertex, geometry, and pixel shaders are where the meat of the action occurs. When rendering with Direct3D 11, the GPU must have a valid vertex shader and pixel shader. Geometry shader is an advanced feature in Direct3D 11 and is optional, so we will not discuss geometry shaders in this tutorial. In Direct3D 11 there are also hull and domain shaders for tessellation and compute shaders for compute. For more information about these, see the other samples.

Vertex Shaders

Vertex shaders are short programs that are executed by the GPU on vertices. Think of vertex shaders as C functions that take each vertex as input, process the input, and then output the modified vertex. After the application passes vertex data to the GPU in the form of a vertex buffer, the GPU iterates through the vertices in the vertex buffer, and executes the active vertex shader once for each vertex, passing the vertex's data to the vertex shader as input parameters.

While a vertex shader can be used to carry out many tasks, the most important job of a vertex shader is transformation. Transformation is the process of converting vectors from one coordinate system to another. For example, a triangle in a 3D scene may have its vertices at the positions (0, 0, 0) (1, 0, 0) (0, 1, 0). When the triangle is drawn on a 2D texture buffer, the GPU has to know the 2D coordinates of the points on the buffer that the vertices should be drawn at. It is transformation that helps us accomplish this. Transformation will be discussed in detail in the next tutorial. For this tutorial, we will be using a simple vertex shader that does nothing except passing the input data through as output.

In the Direct3D 11 tutorials, we will write our shaders in High-Level Shading Language (HLSL). Recall that our vertex data has a 3D position element, and the vertex shader will do no processing on the input at all. The resulting vertex shader looks like the following:

```
float4 VS( float4 Pos : POSITION ) : SV_POSITION
{
    return Pos;
}
```

This vertex shader looks a lot like a C function. HLSL uses C-like syntax to make learning easier for C/C++ programmers. We can see that this vertex shader, named VS, takes a parameter of float4 type and returns a float4 value. In HLSL, a float4 is a 4-component vector where each component is a floating-point number. The colons define the semantics of the parameter as well as the return value. As mentioned above, the semantics in HLSL describe the nature of the data. In our shader above, we choose POSITION as the semantics of the Pos input parameter because this parameter will contain the vertex position. The return value's semantics, SV_POSITION, is a pre-defined semantics with special meaning. This semantics tells the graphics pipeline that the data associated with the semantics defines the clip-space position. This position is needed by the GPU in order to drawn pixels on the screen. (We will discuss clip-space in the next tutorial.) In our shader, we take the input position data and output the exact same data back to the pipeline.

Pixel Shaders

Modern computer monitors are commonly raster display, which means the screen is actually a two-dimensional grid of small dots called pixels. Each pixel contains a color independent of other pixels. When we render a triangle on the screen, we don't really render a triangle as one entity. Rather, we light up the group of pixels that are covered by the triangle's area. Figure 2 shows an illustration of this.

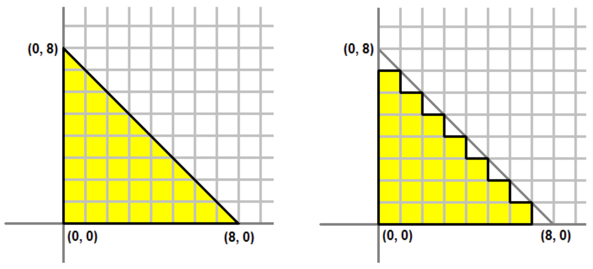


Figure 2. Left: What we would like to draw. Right: What is actually on the screen.

The process of converting a triangle defined by three vertices to a bunch of pixels covered by the triangle is called rasterization. The GPU first determines what pixels are covered by the triangle being rendered. Then it invokes the active pixel shader for each of these pixels. A pixel shader's primary purpose is to compute the color that each pixel should have. The shader takes certain input about the pixel being colored, computes the pixel's color, then outputs that color back to the pipeline. The input that it takes comes from the active geometry shader, or, if a geometry shader is not present, such as the case in this tutorial, the input comes directly from the vertex shader.
The vertex shader we created above outputs a float4 with the semantics SV_POSITION. This will be the input of our pixel shader. Since pixel shaders output color values, the output of our pixel shader will be a float4. We give the output the semantics SV_TARGET to signify outputting to the render target format. The pixel shader looks like the following:

```
float4 PS( float4 Pos : SV_POSITION ) : SV_Target
{
    return float4( 1.0f, 1.0f, 0.0f, 1.0f );    // Yellow, with Alpha = 1
}
```

Creating the Shaders

In the application code, we will need to create a vertex shader and a pixel shader object. These objects represent our shaders, and are created by calling **D3DX11CompileFromFile()**. The code is demonstrated below:

```
 // Create the vertex shader
D3DX11CompileFromFile( "Tutorial03.fx", NULL, NULL, "VS", "vs_4_0",
             D3DCOMPILE_ENABLE_STRICTNESS, NULL, NULL, &pVSBlob, &pErrorBlob, NULL );

// Create the pixel shader
D3DX11CompileFromFile( "Tutorial03.fx", NULL, NULL, "PS", "ps_4_0",
             D3DCOMPILE_ENABLE_STRICTNESS, NULL, NULL, &pPSBlob, &pErrorBlob, NULL );
```

Putting It Together

After walking through the graphics pipeline, we can start to understand the process of rendering the triangle we created at the beginning of Tutorial 2. Creating Direct3D applications requires two distinct steps. The first would be creating the source data in vertex data, as we've done in Tutorial 2. The second stage would be to create the shaders which would transform that data for rendering, which we showed in this tutorial.

Tutorial 4: 3D Spaces

Summary

In the previous tutorial, we successfully rendered a triangle in the center of our application window. We haven't paid much attention to the vertex positions that we have picked in our vertex buffer. In this tutorial, we will delve into the details of 3D positions and transformation.
The outcome of this tutorial will be a 3D object rendered to screen. Whereas previous tutorials focused on rendering a 2D object onto a 3D world, here we show a 3D object.

3D Spaces

In the previous tutorial, the vertices of the triangle were placed strategically to perfectly align themselves on the screen. However, this will not always be the case. Thus, we need a system to denote objects in 3D space and a system to display them.
In the real world, objects exist in 3D space. This means that to place an object in a particular position in the world, we would need to use a coordinate system and define three coordinates that correspond to the position. In computer graphics, 3D spaces are most commonly in Cartesian coordinate system. In this coordinate system, three axes, X, Y, and Z, perpendicular to each other, dictate the coordinate that each point in the space has. This coordinate system is further divided into left-handed and right-handed systems. In a left-handed system, when X axis points to the right and Y axis points to up, Z axis points forward. In a right-handed system, with the same X and Y axes, Z axis points backward.

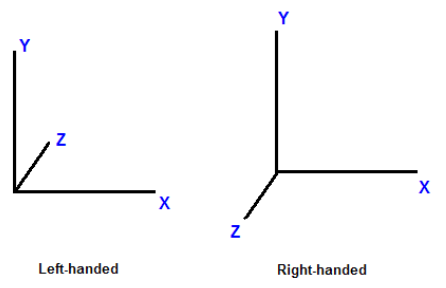


Figure 1. Left-handed versus right-handed coordinate systems

Now that we have talked about the coordinate system, consider 3D spaces. A point has different coordinates in different spaces. As an example in 1D, suppose we have a ruler and we note the point, P, at the 5-inch mark of the ruler. Now, if we move the ruler 1 inch to the right, the same point lies on the 4-inch mark. By moving the ruler, the frame of reference has changed. Therefore, while the point hasn't moved, it has a new coordinate.

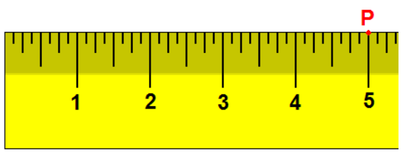


Figure 2. Spaces illustration in 1D

In 3D, a space is typically defined by an origin and three unique axes from the origin: X, Y and Z. There are several spaces commonly used in computer graphics: object space, world space, view space, projection space, and screen space.

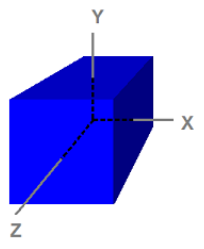


Figure 3. A cube defined in object space

Object Space

Notice that the cube is centered on the origin. Object space, also called model space, refers to the space used by artists when they create the 3D models. Usually, artists create models that are centered around the origin so that it is easier to perform transformations such as rotations to the models, as we will see when we discuss transformation. The eight vertices have the following coordinates:

```
(-1,  1, -1)
( 1,  1, -1)
(-1, -1, -1)
( 1, -1, -1)
(-1,  1,  1)
( 1,  1,  1)
(-1, -1,  1)
( 1, -1,  1)
```

Because object space is what artists typically use when they design and create models, the models that are stored on disk are also in object space. An application can create a vertex buffer to represent such a model and initialize the buffer with the model data. Therefore, the vertices in the vertex buffer will usually be in object space as well. This also means that the vertex shader receives input vertex data in object space.

World Space

World space is a space shared by every object in the scene. It is used to define spatial relationship between objects that we wish to render. To visualize world space, we could imagine that we are standing in the south-western corner of a rectangular room facing north. We define the corner that our feet are standing at to be the origin, (0, 0, 0). The X axis goes to our right; the Y axis goes up; and the Z axis goes forward, the same direction as we are facing. When we do this, every position in the room can be identified with a set of XYZ coordinates. For instance, there may be a chair 5 feet in front and 2 feet to the right of us. There may be a light on the 8-foot-high ceiling directly on top of the chair. We can then refer to the position of the chair as (2, 0, 5) and the position of the light as (2, 8, 5). As we see, world space is so-called because they tell us where objects are in relation to each other in the world.

View Space

View space, sometimes called camera space, is similar to world space in that it is typically used for the entire scene. However, in view space, the origin is at the viewer or camera. The view direction (where the viewer is looking) defines the positive Z axis. An "up" direction defined by the application becomes the positive Y axis as shown below.

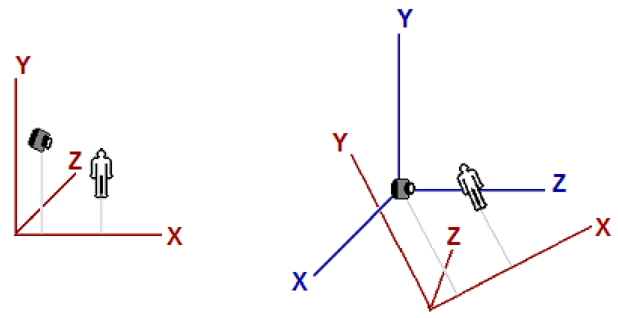


Figure 4. The same object in world space (left) and in view space (right)

The left image shows a scene that consists of a human-like object and a viewer (camera) looking at the object. The origin and axes that are used by world space are shown in red. The right image shows the view space in relation to world space. The view space axes are shown in blue. For clearer illustration, the view space does not have the same orientation as the world space in the left image to readers. Note that in view space, the viewer is looking in the Z direction.

Projection Space

Projection space refers to the space after applying projection transformation from view space. In this space, visible content has X and Y coordinates ranging from -1 to 1, and Z coordinate ranging from 0 to 1.

Screen Space

Screen space is often used to refer to locations in the frame buffer. Because frame buffer is usually a 2D texture, screen space is a 2D space. The top-left corner is the origin with coordinates (0, 0). The positive X goes to right and positive Y goes down. For a buffer that is w pixels wide and h pixels high, the most lower-right pixel has the coordinates (w - 1, h - 1).

Space-to-space Transformation

Transformation is most commonly used to convert vertices from one space to another. In 3D computer graphics, there are logically three such transformations in the pipeline: world, view, and projection transformation. Individual transformation operations such as translation, rotation, and scaling are covered in the next tutorial.

World Transformation

World transformation, as the name suggests, converts vertices from object space to world space. It usually consists of one or more scaling, rotation, and translation, based on the size, orientation, and position we would like to give to the object. Every object in the scene has its own world transformation matrix. This is because each object has its own size, orientation, and position.

View Transformation

After vertices are converted to world space, view transformation converts those vertices from world space to view space. Recall from earlier discussion that view space is what the world appears from the viewer's (or camera's) perspective. In view space, the viewer is located at origin looking out along the positive Z axis.
It is worth noting that although view space is the world from the viewer's frame of reference, view transformation matrix is applied to vertices, not the viewer. Therefore, the view matrix must perform the opposite transformation that we apply to our viewer or camera. For example, if we want to move the camera 5 units towards the -Z direction, we would need to compute a view matrix that translates vertices for 5 units along the +Z direction. Although the camera has moved backward, the vertices, from the camera's point of view, have moved forward. In XNA Math a convenient API call **XMMatrixLookAtLH()** is often used to compute a view matrix. We would simply need to tell it where the viewer is, where it's looking at, and the direction representing the viewer's top, also called the up-vector, to obtain a corresponding view matrix.

Projection Transformation

Projection transformation converts vertices from 3D spaces such as world and view spaces to projection space. In projection space, X and Y coordinates of a vertex are obtained from the X/Z and Y/Z ratios of this vertex in 3D space.

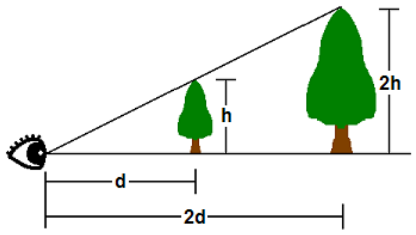


Figure 5. Projection

In 3D space, things appear in perspective. That is, the closer an object is, the larger it appears. As shown, the tip of a tree that is h units tall at d units away from the viewer's eye will appear at the same point as the tip of another tree 2h units tall and 2d units away. Therefore, where a vertex appears on a 2D screen is directly related to its X/Z and Y/Z ratios.
One of the parameters that defines a 3D space is called the field-of-view (FOV). FOV denotes which objects are visible from a particular position, while looking in a particular direction. Humans have a FOV that is forward-looking (we can't see what is behind us), and we can't see objects that are too close or too far away. In computer graphics, the FOV is contained in a view frustum. The view frustum is defined by 6 planes in 3D. Two of these planes are parallel to the XY plane. These are called the near-Z and far-Z planes. The other four planes are defined by the viewer's horizontal and vertical field of view. The wider the FOV is, the wider the frustum volume is, and the more objects the viewer sees.
The GPU filters out objects that are outside the view frustum so that it does not have to spend time rendering something that will not be displayed. This process is called clipping. The view frustum is a 4-sided pyramid with its top cut off. Clipping against this volume is complicated because to clip against one view frustum plane, the GPU must

compare every vertex to the plane's equation. Instead, the GPU generally performs projection transformation first, and then clips against the view frustum volume. The effect of projection transformation on the view frustum is that the pyramid shaped view frustum becomes a box in projection space. This is because, as mentioned previously, in projection space the X and Y coordinates are based on the X/Z and Y/Z in 3D space. Therefore, point a and point b will have the same X and Y coordinates in projection space, which is why the view frustum becomes a box.

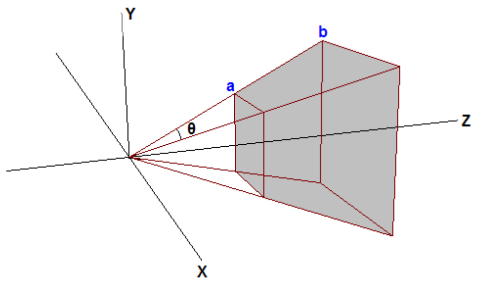


Figure 6. View Frustum

Suppose that the tips of the two trees lie exactly on the top view frustum edge. Further suppose that d = 2h. The Y coordinate along the top edge in projection space will then be 0.5 (because h/d = 0.5). Therefore, any Y values post-projection that are greater than 0.5 will be clipped by the GPU. The problem here is that 0.5 is determined by the vertical field of view chosen by the program, and different FOV values result in different values that the GPU has to clip against. To make the process more convenient, 3D programs generally scale the projected X and Y values of vertices so that the visible X and Y values range from -1 to 1. In other words, anything with X or Y coordinate that's outside the [-1 1] range will be clipped out. To make this clipping scheme work, the projection matrix must scale the X and Y coordinates of projected vertices by the inverse of h/d, or d/h. d/h is also the cotangent of half of FOV. With scaling, the top of the view frustum becomes h/d * d/h = 1. Anything greater than 1 will be clipped by the GPU. This is what we want. A similar tweak is generally done for the Z coordinate in projection space as well. We would like the near and far Z planes to be at 0 and 1 in projection space, respectively. When Z = near-Z value in 3D space, Z should be 0 in projection space; when Z = far-Z in 3D space, Z should be 1 in projection space. After this is done, any Z values outside [0 1] will be clipped out by the GPU.
In Direct3D 11, the easiest way to obtain a projection matrix is to call the **XMMatrixPerspectiveFovLH()** method. We simply supply 4 parameters—FOVy, Aspect, Zn, and Zf—and get back a matrix that does everything necessary as mentioned above. FOVy is the field of view in Y direction. Aspect is the aspect ratio, which is ratio of view space width to height. From FOVy and Aspect, FOVx can be computed. This aspect ratio is usually obtained from the ratio of the render target width to height. Zn and Zf are the near and far Z values in view space, respectively.

Using Transformation

In the previous tutorial, we wrote a program that renders a single triangle to screen. When we create the vertex buffer, the vertex positions that we use are directly in projection space so that we don't have to perform any transformation. Now that we have an understanding of 3D space and transformation, we are going to modify the program so that the vertex buffer is defined in object space, as it should be. Then, we will modify our vertex shader to transform the vertices from object space to projection space.

Modifying the Vertex Buffer

Since we started representing things in three dimensions, we have changed the flat triangle from the previous tutorial to a cube. This will allow us to demonstrate these concepts much clearer.

```
SimpleVertex vertices[] =
    {
        { XMFLOAT3( -1.0f,  1.0f, -1.0f ), XMFLOAT4( 0.0f, 0.0f, 1.0f, 1.0f ) },
        { XMFLOAT3(  1.0f,  1.0f, -1.0f ), XMFLOAT4( 0.0f, 1.0f, 0.0f, 1.0f ) },
        { XMFLOAT3(  1.0f,  1.0f,  1.0f ), XMFLOAT4( 0.0f, 1.0f, 1.0f, 1.0f ) },
        { XMFLOAT3( -1.0f,  1.0f,  1.0f ), XMFLOAT4( 1.0f, 0.0f, 0.0f, 1.0f ) },
        { XMFLOAT3( -1.0f, -1.0f, -1.0f ), XMFLOAT4( 1.0f, 0.0f, 1.0f, 1.0f ) },
        { XMFLOAT3(  1.0f, -1.0f, -1.0f ), XMFLOAT4( 1.0f, 1.0f, 0.0f, 1.0f ) },
        { XMFLOAT3(  1.0f, -1.0f,  1.0f ), XMFLOAT4( 1.0f, 1.0f, 1.0f, 1.0f ) },
        { XMFLOAT3( -1.0f, -1.0f,  1.0f ), XMFLOAT4( 0.0f, 0.0f, 0.0f, 1.0f ) },
    };
```

If you notice, all we did was specify the eight points on the cube, but we didn't actually describe the individual triangles. If we passed this in as-is, the output would not be what we expect. We will need to specify the triangles that form the cube through these eight points.
On a cube, many triangles will be sharing the same vertex and it would be a waste of space to redefine the same points over and over again. As such, there is a method to specify just the eight points, and then let Direct3D know which points to pick for a triangle. This is done through an index buffer. An index buffer will contain a list, which will refer to the index of vertices in the buffer, to specify which points to use in each triangle. The code below shows which points make up each of our triangles.

```
    // index buffer
    WORD indices[] =
    {
        3,1,0,
        2,1,3,

        0,5,4,
        1,5,0,

        3,4,7,
        0,4,3,

        1,6,5,
        2,6,1,

        2,7,6,
        3,7,2,

        6,4,5,
        7,4,6,
    };
```

As you can see, the first triangle is defined by points 3, 1, and 0. This means that the first triangle has vertices at: (-1.0f, 1.0f, 1.0f),(1.0f, 1.0f, -1.0f), and (-1.0f, 1.0f, -1.0f), respectively. There are six faces on the cube, and each face is comprised of two triangles. Thus, you see 12 total triangles defined here.
Since each vertex is explicitly listed, and no two triangles are sharing edges (at least, in the way it has been defined), this is considered a triangle list. In total, for 12 triangles in a triangle list, we will require a total of 36 vertices.
The creation of the index buffer is very similar to the vertex buffer, where we specified parameters such as size and type in a structure, and called CreateBuffer. The type is D3D11_BIND_INDEX_BUFFER, and since we declared our array using DWORD, we will use sizeof(DWORD).

```
    D3D11_BUFFER_DESC bd;
    ZeroMemory( &bd, sizeof(bd) );
    bd.Usage = D3D11_USAGE_DEFAULT;
    // 36 vertices needed for 12 triangles in a triangle list
    bd.ByteWidth = sizeof( WORD ) * 36;
    bd.BindFlags = D3D11_BIND_INDEX_BUFFER;
    bd.CPUAccessFlags = 0;
    bd.MiscFlags = 0;
    InitData.pSysMem = indices;
    g_pd3dDevice->CreateBuffer( &bd, &InitData, &g_pIndexBuffer );
```

Once we created this buffer, we will need to set it so that Direct3D knows to refer to this index buffer when generating the triangles. We specify the pointer to the buffer, the format, and the offset in the buffer to start referencing from.

```
    // Set index buffer
    g_pImmediateContext->IASetIndexBuffer( g_pIndexBuffer, DXGI_FORMAT_R16_UINT, 0 );
```

Modifying the Vertex Shader

In our vertex shader from the previous tutorial, we take the input vertex position and output the same position without any modification. We can do this because the input vertex position is already defined in projection space. Now, because the input vertex position is defined in object space, we must transform it before outputting from the vertex

shader. We do this with three steps: transform from object to world space, transform from world to view space, and transform from view to projection space. The first thing that we need to do is declare three constant buffer variables. Constant buffers are used to store data that the application needs to pass to shaders. Before rendering, the application usually writes important data to constant buffers, and then during rendering the data can be read from within the shaders. In an FX file, constant buffer variables are declared like global variables in a C++ struct. The three variables that we will use are the world, view, and projection transformation matrices of the HLSL type "matrix."
Once we have declared the matrices that we will need, we update our vertex shader to transform the input position by using the matrices. A vector is transformed by multiplying the vector by a matrix. In HLSL, this is done using the mul() intrinsic function. Our variable declaration and new vertex shader are shown below:

```
cbuffer ConstantBuffer : register( b0 )
{
    matrix World;
    matrix View;
    matrix Projection;
}

//
// Vertex Shader
//
VS_OUTPUT VS( float4 Pos : POSITION, float4 Color : COLOR )
{
    VS_OUTPUT output = (VS_OUTPUT)0;
    output.Pos = mul( Pos, World );
    output.Pos = mul( output.Pos, View );
    output.Pos = mul( output.Pos, Projection );
    output.Color = Color;
    return output;
}
```

In the vertex shader, each mul() applies one transformation to the input position. The world, view, and projection transformations are applied in that order sequentially. This is necessary because vector and matrix multiplication is not commutative.

Setting up the Matrices

We have updated our vertex shader to transform using matrices, but we also need to define three matrices in our program. These three matrices will store the transformation to be used when we render. Before rendering, we copy the values of these matrices to the shader constant buffer. Then, when we initiate the rendering by calling Draw(), our vertex shader reads the matrices stored in the constant buffer. In addition to the matrices, we also need an ID3D11Buffer object that represents the constant buffer. Therefore, our global variables will have the following addition:

```
ID3D11Buffer* g_pConstantBuffer = NULL;
XMMATRIX g_World;
XMMATRIX g_View;
XMMATRIX g_Projection;
```

To create the ID3D11Buffer object, we use ID3D11Device::CreateBuffer() and specify D3D11_BIND_CONSTANT_BUFFER.

```
D3D11_BUFFER_DESC bd;
ZeroMemory( &bd, sizeof(bd) );
bd.Usage = D3D11_USAGE_DEFAULT;
bd.ByteWidth = sizeof(ConstantBuffer);
bd.BindFlags = D3D11_BIND_CONSTANT_BUFFER;
bd.CPUAccessFlags = 0;
g_pd3dDevice->CreateBuffer( &bd, NULL, &g_pConstantBuffer );
```

The next thing that we need to do is come up with three matrices that we will use to do the transformation. We want the triangle to be sitting on origin, parallel to the XY plane. This is exactly how it is stored in the vertex buffer in object space. Therefore, the world transformation needs to do nothing, and we initialize the world matrix to an identity matrix. We would like to set up our camera so that it is situated at [0 1 -5], looking at the point [0 1 0]. We can call

XMMatrixLookAtLH() to conveniently compute a view matrix for us using the up vector [0 1 0] since we would like the +Y direction to always stay at top. Finally, to come up with a projection matrix, we call XMMatrixPerspectiveFovLH(), with a 90 degree vertical field of view (pi/2), an aspect ratio of 640/480 which is from our back buffer size, and near and far Z at 0.1 and 110, respectively. This means that anything closer than 0.1 or further than 110 will not be visible on the screen. These three matrices are stored in the global variables g_World, g_View, and g_Projection.
Updating Constant Buffers
We have the matrices, and now we must write them to the constant buffer when rendering so that the GPU can read them. To update the buffer, we can use the ID3D11DeviceContext::UpdateSubresource() API and pass it a pointer to the matrices stored in the same order as the shader's constant buffer. To help do this, we will create a structure that has the same layout as the constant buffer in the shader. Also, because matrices are arranged differently in memory in C++ and HLSL, we must transpose the matrices before updating them.

```
    ConstantBuffer cb;
    cb.mWorld = XMMatrixTranspose( g_World );
    cb.mView = XMMatrixTranspose( g_View );
    cb.mProjection = XMMatrixTranspose( g_Projection );
    g_pImmediateContext->UpdateSubresource( g_pConstantBuffer, 0, NULL, &cb, 0, 0 );
```

Tutorial 5: 3D Transformation

Summary

In the previous tutorial, we rendered a cube from model space to the screen. In this tutorial, we will extend the concept of transformations and demonstrate simple animation that can be achieved with these transformations.
The outcome of this tutorial will be an object that orbits around another. It would be useful to demonstrate the transformations and how they can be combined to achieve the desired effect. Future tutorials will be building on this foundation as we introduce new concepts.

Transformation

In 3D graphics, transformation is often used to operate on vertices and vectors. It is also used to convert them in one space to another. Transformation is performed by way of multiplication with a matrix. There are typically three types of primitive transformation that can be performed on vertices: translation (where it lies in space relative to the origin), rotation (its direction in relation to the x, y, z frame), and scaling (its distance from origin). In addition to those, projection transformation is used to go from view space to projection space. The XNA Math library contains APIs that can conveniently construct a matrix for many purposes such as translation, rotation, scaling, world-to-view transformation, view-to-projection transformation, and so on. An application can then use these matrices to transform vertices in its scene. A basic understanding of matrix transformations is required. We will briefly look at some examples below.

Translation

Translation refers to moving or displacing for a certain distance in space. In 3D, the matrix used for translation has the form

```
    1  0  0  0
    0  1  0  0
    0  0  1  0
    a  b  c  1
```

where (a, b, c) is the vector that defines the direction and distance to move. For example, to move a vertex -5 unit along the X axis (negative X direction), we can multiply it with this matrix:

```
    1  0  0  0
    0  1  0  0
    0  0  1  0
   -5  0  0  1
```

If we apply this to a cube object centered at origin, the result is that the box is moved 5 units towards the negative X axis, as figure 5 shows, after translation is applied.

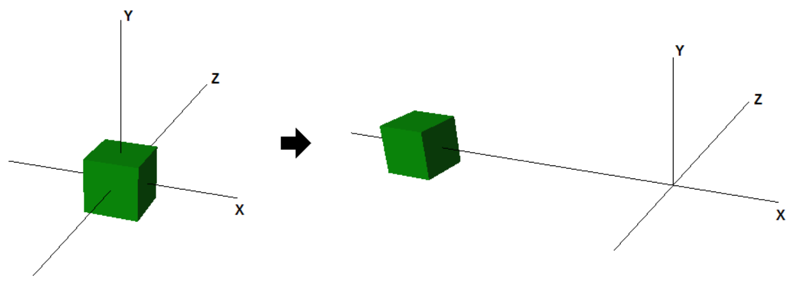


Figure 1. The effect of translation

In 3D, a space is typically defined by an origin and three unique axes from the origin: X, Y and Z. There are several spaces commonly used in computer graphics: object space, world space, view space, projection space, and screen space.

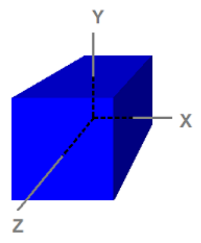


Figure 2. A cube defined in object space

Rotation

Rotation refers to rotating vertices about an axis going through the origin. Three such axes are the X, Y, and Z axes in the space. An example in 2D would be rotating the vector [1 0] 90 degrees counter-clockwise. The result from the rotation is the vector [0 1]. The matrix used for rotating ΐ degrees clockwise about the Y axis looks like this:

```
cosβ  0  -sinβ   0
 0    1    0     0
sinβ  0   cosβ   0
 0    0    0     1
```

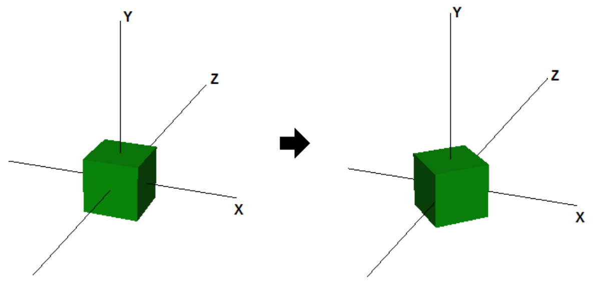


Figure 3. The effect of rotation about the Y axis

Scaling

Scaling refers to enlarging or shrinking the size of vector components along axis directions. For example, a vector can be scaled up along all directions or scaled down along the X axis only. To scale, we usually apply the scaling matrix below:

```
p  0  0  0
0  q  0  0
0  0  r  0
0  0  0  1
```

where p, q, and r are the scaling factor along the X, Y, and Z direction, respectively. The figure below shows the effect of scaling by 2 along the X axis and scaling by 0.5 along the Y axis.

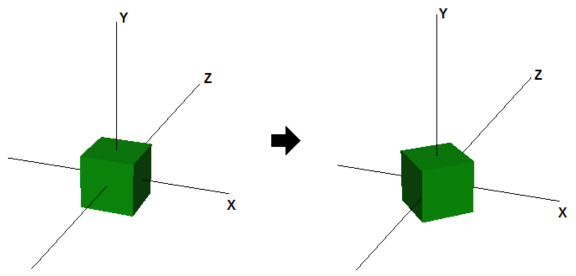


Figure 4. The effect of Scaling

Multiple Transformations

To apply multiple transformations to a vector, we can simply multiply the vector by the first transformation matrix, then multiply the resulting vector by the second transformation matrix, and so on. Because vector and matrix multiplication is associative, we can also multiply all of the matrices first, then multiply the vector by the product matrix and obtain an identical result. The figure below shows how the cube would end up if we combine a rotation and a translation transformation together.

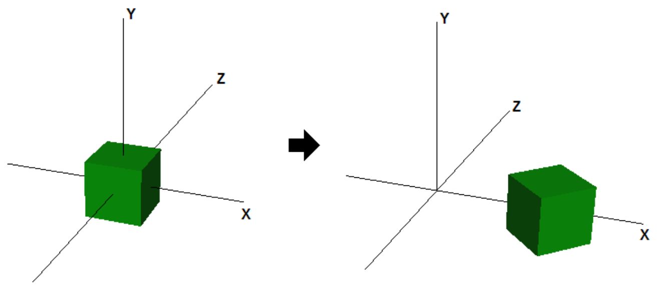


Figure 5. The effect of rotation and translation

Creating the Orbit

In this tutorial, we will be transforming two cubes. The first one will rotate in place, while the second one will rotate around the first, while spinning on its own axis. The two cubes will have its own world transformation matrix associated with it, and this matrix will be reapplied to it in every frame rendered.

There are functions within XNA Math that will assist in the creation of the rotation, translation, and scaling matrices.

- Rotations performed around the X, Y. and Z axis are accomplished with the functions XMMatrixRotationX, XMMatrixRotationY, and XMMatrixRotationZ, respectively. They create basic rotation matrices that rotate around one of the primary axis. Complex rotations around other axis can be done by multiplying together several of them.
- Translations can be performed by invoking the XMMatrixTranslation function. This function will create a matrix that will translate points specified by the parameters.
- Scaling is done with XMMatrixScaling. It scales only along the primary axes. If scaling along arbitrary axis is desired, then the scaling matrix can be multiplied with an appropriate rotation matrix to achieve the effect.

The first cube will be spinning in place and act as the center for the orbit. The cube has a rotation along the Y axis applied to the associated world matrix. This is done by calling the XMMatrixRotationY function shown in the following code. The cube is rotated by a set amount each frame. Since the cubes are suppose to continuously rotate, the value which the rotation matrix is based on gets incremented with every frame.

```
    // 1st Cube: Rotate around the origin
    g_World1 = XMMatrixRotationY( t );
```

The second cube will be orbiting around the first one. To demonstrate multiple transformations, a scaling factor, and its own axis spin will be added. The formula used is shown right below the code (in comments). First the cube will be scale down to 30% size, and then it will be rotated along its spin axis (the Z axis in this case). To simulate the orbit, it will get translated away from the origin, and then rotated along the Y axis. The desired effect can be achieved by using four separate matrices with its individual transformation (mScale, mSpin, mTranslate, mOrbit), then multiplied together.

```
    // 2nd Cube:  Rotate around origin
    XMMATRIX mSpin = XMMatrixRotationZ( -t );
    XMMATRIX mOrbit = XMMatrixRotationY( -t * 2.0f );
    XMMATRIX mTranslate = XMMatrixTranslation( -4.0f, 0.0f, 0.0f );
    XMMATRIX mScale = XMMatrixScaling( 0.3f, 0.3f, 0.3f );
    g_World2 = mScale * mSpin * mTranslate * mOrbit;
```

An important point to note is that these operations are not commutative. The order in which the transformations are applied matter. Experiment with the order of transformation and observe the results.
Since all the transformation functions will create a new matrix from the parameters, the amount at which they rotate has to be incremented. This is done by updating the "time" variable.

```
    // Update our time
    t += XM_PI * 0.0125f;
```

Before the rendering calls are made, the constant buffer must be updated for the shaders. Note that the world matrix is unique to each cube, and thus, changes for every object that gets passed into it.

```
    //
    // Update variables for the first cube
    //
    ConstantBuffer cb1;
    cb1.mWorld = XMMatrixTranspose( g_World1 );
    cb1.mView = XMMatrixTranspose( g_View );
    cb1.mProjection = XMMatrixTranspose( g_Projection );
    g_pImmediateContext->UpdateSubresource( g_pConstantBuffer, 0, NULL, &cb1, 0, 0 );

    //
    // Render the first cube
    //
    g_pImmediateContext->VSSetShader( g_pVertexShader, NULL, 0 );
    g_pImmediateContext->VSSetConstantBuffers( 0, 1, &g_pConstantBuffer );
    g_pImmediateContext->PSSetShader( g_pPixelShader, NULL, 0 );
    g_pImmediateContext->DrawIndexed( 36, 0, 0 );

    //
    // Update variables for the second cube
    //
    ConstantBuffer cb2;
    cb2.mWorld = XMMatrixTranspose( g_World2 );
    cb2.mView = XMMatrixTranspose( g_View );
    cb2.mProjection = XMMatrixTranspose( g_Projection );
    g_pImmediateContext->UpdateSubresource( g_pConstantBuffer, 0, NULL, &cb2, 0, 0 );

    //
    // Render the second cube
    //
    g_pImmediateContext->DrawIndexed( 36, 0, 0 );
```

The Depth Buffer
There is one other important addition to this tutorial, and that is the depth buffer. Without it, the smaller orbiting cube would still be drawn on top of the larger center cube when it went around the back of the latter. The depth buffer allows Direct3D to keep track of the depth of every pixel drawn to the screen. The default behavior of the depth buffer in Direct3D 11 is to check every pixel being drawn to the screen against the value stored in the depth buffer for that screen-space pixel. If the depth of the pixel being rendered is less than or equal to the value already in the depth buffer, the pixel is drawn and the value in the depth buffer is updated to the depth of the newly drawn pixel. On the other hand, if the pixel being draw has a depth greater than the value already in the depth buffer, the pixel is discarded and the depth value in the depth buffer remains unchanged.
The following code in the sample creates a depth buffer (a DepthStencil texture). It also creates a DepthStencilView of the depth buffer so that Direct3D 11 knows to use it as a Depth Stencil texture.

```
    // Create depth stencil texture
    D3D11_TEXTURE2D_DESC descDepth;
    ZeroMemory( &descDepth, sizeof(descDepth) );
    descDepth.Width = width;
    descDepth.Height = height;
    descDepth.MipLevels = 1;
    descDepth.ArraySize = 1;
    descDepth.Format = DXGI_FORMAT_D24_UNORM_S8_UINT;
    descDepth.SampleDesc.Count = 1;
    descDepth.SampleDesc.Quality = 0;
    descDepth.Usage = D3D11_USAGE_DEFAULT;
    descDepth.BindFlags = D3D11_BIND_DEPTH_STENCIL;
    descDepth.CPUAccessFlags = 0;
    descDepth.MiscFlags = 0;
    hr = g_pd3dDevice->CreateTexture2D( &descDepth, NULL, &g_pDepthStencil );
    if( FAILED(hr) )
        return hr;

    // Create the depth stencil view
    D3D11_DEPTH_STENCIL_VIEW_DESC descDSV;
    ZeroMemory( &descDSV, sizeof(descDSV) );
    descDSV.Format = descDepth.Format;
    descDSV.ViewDimension = D3D11_DSV_DIMENSION_TEXTURE2D;
    descDSV.Texture2D.MipSlice = 0;
  g_pd3dDevice->CreateDepthStencilView(g_pDepthStencil,&descDSV,&g_pDepthStencilView );
```

In order to use this newly created depth stencil buffer, the tutorial must bind it to the device. This is done by passing the depth stencil view to the third parameter of the OMSetRenderTargets function.

```
 g_pImmediateContext->OMSetRenderTargets(1,&g_pRenderTargetView, g_pDepthStencilView );
```

As with the render target, we must also clear the depth buffer before rendering. This ensures that depth values from previous frames do not incorrectly discard pixels in the current frame. In the code below the tutorial is actually setting the depth buffer to be the maximum amount (1.0).

```
    //
    // Clear the depth buffer to 1.0 (max depth)
    //
```

Tutorial 6: Lighting

Summary

In the previous tutorials, the world looks boring because all the objects are lit in the same way. This tutorial will introduce the concept of simple lighting and how it can be applied. The technique used will be lambertian lighting.
The outcome of this tutorial will modify the previous example to include a light source. This light source will be attached to the cube in orbit. The effects of the light can be seen on the center cube.

Lighting

In this tutorial, the most basic type of lighting will be introduced: lambertian lighting. Lambertian lighting has uniform intensity irrespective of the distance away from the light. When the light hits the surface, the amount of light reflected is calculated by the angle of incidence the light has on the surface. When a light is shined directly on a surface, it is shown to reflect all the light back, with maximum intensity. However, as the angle of the light is increased, the intensity of the light will fade away.
To calculate the intensity that a light has on a surface, the angle between the light direction and the normal of the surface has to be calculated. The normal for a surface is defined as a vector that is perpendicular to the surface. The calculation of the angle can be done with a simple dot product, which will return the projection of the light direction vector onto the normal. The wider the angle, the smaller the projection will be. Thus, this gives us the correct function to modulate the diffused light with.

The light source used in this tutorial is an approximation of directional lighting. The vector which describes the light source determines the direction of the light. Since it's an approximation, no matter where an object is, the direction in which the light shines towards it is the same. An example of this light source is the sun. The sun is always seen to be shining in the same direction for all objects in a scene. In addition, the intensity of the light on individual objects is not taken into consideration.
Other types of light include point lights, which radiate uniform light from their centers, and spot lights, which are directional but not uniform across all objects.

Initializing the Lights

In this tutorial, there will be two light sources. One will be statically placed above and behind the cube, and another one will be orbiting the center cube. Note that the orbiting cube in the previous tutorial has been replaced with this light source. Since lighting is computed by the shaders, the variables would have to be declared and then bound to the variables within the technique. In this sample, we just require the direction of the light source, as well as its color value. The first light is grey and not moving, while the second one is an orbiting red light.

```
XMFLOAT4 vLightDirs[2] =
{
    XMFLOAT4( -0.577f, 0.577f, -0.577f, 1.0f ),
    XMFLOAT4( 0.0f, 0.0f, -1.0f, 1.0f ),
};
XMFLOAT4 vLightColors[2] =
{
    XMFLOAT4( 0.5f, 0.5f, 0.5f, 1.0f ),
    XMFLOAT4( 0.5f, 0.0f, 0.0f, 1.0f )
};
```

The orbiting light is rotated just like the cube in the last tutorial. The rotation matrix applied will change the direction of the light, to show the effect that it is always shining towards the center. Note that function XMVector3Transform is used to multiply a matrix with a vector. In the previous tutorial, we multiplied just the transformation matrices into the world matrix, then passed into the shader for transformation. However, for simplicity's sake in this case, we're actually doing the world transform of the light in the CPU.

```
XMMATRIX mRotate = XMMatrixRotationY( -2.0f * t );
XMVECTOR vLightDir = XMLoadFloat4( &vLightDirs[1] );
vLightDir = XMVector3Transform( vLightDir, mRotate );
XMStoreFloat4( &vLightDirs[1], vLightDir );
```

The lights' direction and color are both passed into the shader just like the matrices. The associated variable is called to set, and the parameter is passed in.

```
ConstantBuffer cb1;
cb1.mWorld = XMMatrixTranspose( g_World );
cb1.mView = XMMatrixTranspose( g_View );
cb1.mProjection = XMMatrixTranspose( g_Projection );
cb1.vLightDir[0] = vLightDirs[0];
cb1.vLightDir[1] = vLightDirs[1];
cb1.vLightColor[0] = vLightColors[0];
cb1.vLightColor[1] = vLightColors[1];
cb1.vOutputColor = XMFLOAT4(0, 0, 0, 0);
g_pImmediateContext->UpdateSubresource( g_pConstantBuffer, 0, NULL, &cb1, 0, 0 );
```

Rendering the Lights in the Pixel Shader

Once we have all the data set up and the shader properly fed with data, we can compute the lambertian lighting term on each pixel from the light sources. We'll be using the dot product rule discussed previously.
Once we've taken the dot product of the light versus the normal, it can then be multiplied with the color of the light to calculate the effect of that light. That value is passed through the saturate function, which converts the range to [0, 1]. Finally, the results from the two separate lights are summed together to create the final pixel color.
Consider that the material of the surface itself is not factored into this light calculation. The final color of the surface is a result of the light's colors.

```
    float4 PS( PS_INPUT input) : SV_Target
    {
        float4 finalColor = 0;

        for(int i=0; i<2; i++)
        {
      finalColor += saturate( dot( (float3)vLightDir[i],input.Norm) * vLightColor[i] );
        }
        return finalColor;
    }
```

Once through the pixel shader, the pixels will be modulated by the lights, and you can see the effect of each light on the cube surface. Note that the light in this case looks flat because pixels on the same surface will have the same normal. Diffuse is a very simple and easy lighting model to compute. You can use more complex lighting models to achieve richer and more realistic materials.

Tutorial 7: Texture Mapping and Constant Buffers

Summary

In the previous tutorial, we introduced lighting to our project. Now we will build on that by adding textures to our cube. Also, we will introduce the concept of constant buffers, and explain how you can use buffers to speed up processing by minimizing bandwidth usage.
The purpose of this tutorial is to modify the center cube to have a texture mapped onto it.

Texture Mapping

Texture mapping refers to the projection of a 2D image onto 3D geometry. We can think of it as wrapping a present, by placing decorative paper over an otherwise bland box. To do this, we have to specify how the points on the surface of the geometry correspond with the 2D image.
The trick is to properly align the coordinates of the model with the texture. For complex models, it is difficult to determine the coordinates for the textures by hand. Thus, 3D modeling packages generally will export models with corresponding texture coordinates. Since our example is a cube, it is easy to determine the coordinates needed to match the texture. Texture coordinates are defined at the vertices, and are then interpolated for individual pixels on a surface.

Creating a Shader Resource from the Texture and Sampler State

The texture is a 2D image that is retrieved from file and used to create a shader-resource view, so that it can be read from a shader.

```
D3DX11CreateShaderResourceViewFromFile( g_pd3dDevice, L"seafloor.dds", NULL, NULL,
         &g_pTextureRV, NULL );
```

We also need to create a sampler state that controls how the shader handles filtering, MIPs, and addressing. For this tutorial we will enable simple sampler state that enables linear filtering and wrap addressing. To create the sampler state, we will use `ID3D11Device::CreateSamplerState()`.

```
    // Create the sample state
    D3D11_SAMPLER_DESC sampDesc;
    ZeroMemory( &sampDesc, sizeof(sampDesc) );
    sampDesc.Filter = D3D11_FILTER_MIN_MAG_MIP_LINEAR;
    sampDesc.AddressU = D3D11_TEXTURE_ADDRESS_WRAP;
    sampDesc.AddressV = D3D11_TEXTURE_ADDRESS_WRAP;
    sampDesc.AddressW = D3D11_TEXTURE_ADDRESS_WRAP;
    sampDesc.ComparisonFunc = D3D11_COMPARISON_NEVER;
    sampDesc.MinLOD = 0;
    sampDesc.MaxLOD = D3D11_FLOAT32_MAX;
    hr = g_pd3dDevice->CreateSamplerState( &sampDesc, &g_pSamplerLinear );
```

Defining the Coordinates

Before we can map the image onto our cube, we must first define the texture coordinates on each of the vertices of the cube. Since images can be of any size, the coordinate system used has been normalized to [0, 1]. The top left corner of the texture corresponds to (0,0) and the bottom right corner maps to (1,1).
In this example, we're having the whole texture spread across each side of the cube. This simplifies the definition of the coordinates, without confusion. However, it is entirely possible to specify the texture to stretch across all six faces, although it's more difficult to define the points, and it will appear stretched and distorted.
First, we updated the structure used to define our vertices to include the texture coordinates.

```
struct SimpleVertex
{
    XMFLOAT3 Pos;
    XMFLOAT2 Tex;
};
```

Next, we updated the input layout to the shaders to also include these coordinates.

```
// Define the input layout
D3D11_INPUT_ELEMENT_DESC layout[] =
{
  { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0, D3D11_INPUT_PER_VERTEX_DATA, 0 },
  { "TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT, 0, 12, D3D11_INPUT_PER_VERTEX_DATA, 0 },
};
```

Since the input layout changed, the corresponding vertex shader input must also be modified to match the addition.

```
struct VS_INPUT
{
    float4 Pos : POSITION;
    float2 Tex : TEXCOORD;
};
```

Finally, we are ready to include texture coordinates in our vertices we defined back in Tutorial 4. Note the second parameter input is a D3DXVECTOR2 containing the texture coordinates. Each vertex on the cube will correspond to a corner of the texture. This creates a simple mapping where each vertex gets (0,0) (0,1) (1,0) or (1,1) as the coordinate.

```
SimpleVertex vertices[] =
{
    { XMFLOAT3( -1.0f, 1.0f, -1.0f ), XMFLOAT2( 0.0f, 0.0f ) },
    { XMFLOAT3( 1.0f, 1.0f, -1.0f ), XMFLOAT2( 1.0f, 0.0f ) },
    { XMFLOAT3( 1.0f, 1.0f, 1.0f ), XMFLOAT2( 1.0f, 1.0f ) },
    { XMFLOAT3( -1.0f, 1.0f, 1.0f ), XMFLOAT2( 0.0f, 1.0f ) },

    { XMFLOAT3( -1.0f, -1.0f, -1.0f ), XMFLOAT2( 0.0f, 0.0f ) },
    { XMFLOAT3( 1.0f, -1.0f, -1.0f ), XMFLOAT2( 1.0f, 0.0f ) },
    { XMFLOAT3( 1.0f, -1.0f, 1.0f ), XMFLOAT2( 1.0f, 1.0f ) },
    { XMFLOAT3( -1.0f, -1.0f, 1.0f ), XMFLOAT2( 0.0f, 1.0f ) },

    { XMFLOAT3( -1.0f, -1.0f, 1.0f ), XMFLOAT2( 0.0f, 0.0f ) },
    { XMFLOAT3( -1.0f, -1.0f, -1.0f ), XMFLOAT2( 1.0f, 0.0f ) },
    { XMFLOAT3( -1.0f, 1.0f, -1.0f ), XMFLOAT2( 1.0f, 1.0f ) },
    { XMFLOAT3( -1.0f, 1.0f, 1.0f ), XMFLOAT2( 0.0f, 1.0f ) },

    { XMFLOAT3( 1.0f, -1.0f, 1.0f ), XMFLOAT2( 0.0f, 0.0f ) },
    { XMFLOAT3( 1.0f, -1.0f, -1.0f ), XMFLOAT2( 1.0f, 0.0f ) },
    { XMFLOAT3( 1.0f, 1.0f, -1.0f ), XMFLOAT2( 1.0f, 1.0f ) },
    { XMFLOAT3( 1.0f, 1.0f, 1.0f ), XMFLOAT2( 0.0f, 1.0f ) },

    { XMFLOAT3( -1.0f, -1.0f, -1.0f ), XMFLOAT2( 0.0f, 0.0f ) },
    { XMFLOAT3( 1.0f, -1.0f, -1.0f ), XMFLOAT2( 1.0f, 0.0f ) },
    { XMFLOAT3( 1.0f, 1.0f, -1.0f ), XMFLOAT2( 1.0f, 1.0f ) },
    { XMFLOAT3( -1.0f, 1.0f, -1.0f ), XMFLOAT2( 0.0f, 1.0f ) },
    { XMFLOAT3( -1.0f, -1.0f, 1.0f ), XMFLOAT2( 0.0f, 0.0f ) },
    { XMFLOAT3( 1.0f, -1.0f, 1.0f ), XMFLOAT2( 1.0f, 0.0f ) },
    { XMFLOAT3( 1.0f, 1.0f, 1.0f ), XMFLOAT2( 1.0f, 1.0f ) },
    { XMFLOAT3( -1.0f, 1.0f, 1.0f ), XMFLOAT2( 0.0f, 1.0f ) },
};
```

When we sample the texture, we will need to modulate it with a material color for the geometry underneath.

Bind Texture as Shader Resource

A texture and sampler state are objects like the constant buffers that we have seen in previous tutorials. Before they can be used by the shader, they need to be set with the ID3D11DeviceContext::PSSetSamplers() and ID3D11DeviceContext::PSSetShaderResources() APIs.

```
    g_pImmediateContext->PSSetShaderResources( 0, 1, &g_pTextureRV );
    g_pImmediateContext->PSSetSamplers( 0, 1, &g_pSamplerLinear );
```

There we go, now we're ready to use the texture within the shader.

Applying the Texture

To map the texture on top of the geometry, we will be calling a texture lookup function within the pixel shader. The function Sample will perform a texture lookup of a 2D texture, and then return the sampled color. The pixel shader shown below calls this function and multiplies it by the underlying mesh color (or material color), and then outputs the final color.

- txDiffuse is the object storing our texture that we passed in from the code above, when we bound the resource view g_pTextureRV to it.
- samLinear will be described below; it is the sampler specifications for the texture lookup.
- input.Tex is the coordinates of the texture that we have specified in the source.

```
// Pixel Shader
float4 PS( PS_INPUT input) : SV_Target
{
    return txDiffuse.Sample( samLinear, input.Tex ) * vMeshColor;
}
```

Another thing we must remember to do is to pass the texture coordinates through the vertex shader. If we don't, the data is lost when it gets to the pixel shader. Here, we just copy the input's coordinates to the output, and let the hardware handle the rest.

```
// Vertex Shader
PS_INPUT VS( VS_INPUT input )
{
    PS_INPUT output = (PS_INPUT)0;
    output.Pos = mul( input.Pos, World );
    output.Pos = mul( output.Pos, View );
    output.Pos = mul( output.Pos, Projection );
    output.Tex = input.Tex;

    return output;
}
```

Constant Buffers

In Direct3D 11, an application can use a constant buffer to set shader constants (shader variables). Constant buffers are declared using a syntax similar to C-style structs. Constant buffers reduce the bandwidth required to update shader constants by allowing shader constants to be grouped together and committed at the same time, rather than making individual calls to commit each constant separately.

In the previous tutorials, we used a single constant buffer to hold all of the shader constants we need. But the best way to efficiently use constant buffers is to organize shader variables into constant buffers based on their frequency of update. This allows an application to minimize the bandwidth required for updating shader constants. As an example, this tutorial groups constants into three structures: one for variables that change every frame, one for variables that change only when a window size is changed, and one for variables that are set once and then do not change.

The following constant buffers are defined in this tutorial's .fx file.

```
cbuffer cbNeverChanges
{
    matrix View;
};

cbuffer cbChangeOnResize
{
    matrix Projection;
};

cbuffer cbChangesEveryFrame
{
    matrix World;
    float4 vMeshColor;
};
```

To work with these constant buffers, you need to create a ID3D11Buffer object for each one. Then you can call **ID3D11DeviceContext::UpdateSubresource()** to update each constant buffer when needed without affecting the other constant buffers.

```
//
// Update variables that change once per frame
//
CBChangesEveryFrame cb;
cb.mWorld = XMMatrixTranspose( g_World );
cb.vMeshColor = g_vMeshColor;
g_pImmediateContext->UpdateSubresource( g_pCBChangesEveryFrame, 0, NULL, &cb, 0, 0 );
```