Effects Tutorial Win32 Sample

<https://github.com/walbourn/directx-sdk-samples>

This is the DirectX SDK's Direct3D 11 tutorial updated to use the Windows 10 SDK without any dependencies on legacy DirectX SDK content. This sample is a Win32 desktop DirectX 11.0 application for Windows 10, Windows 8.1, Windows 8, and Windows 7.

**This is based on the legacy DirectX SDK (June 2010) Win32 desktop tutorial. This is not intended for use with Windows Store apps, Windows RT, or universal Windows apps.**

# Description

The original Direct3D 10 tutorial series covered Win32 basics in Tutorial01 - Tutorial07, DXUT in Tutorial08 - Tutorial10, and then Effects 10 in Tutorial11 - Tutorial14. This is Tutorial11 - Tutorial14 updated to use the [DXUT for Direct3D 11](http://go.microsoft.com/fwlink/?LinkId=320437) framework and [Effects 11 library](http://go.microsoft.com/fwlink/p/?LinkId=271568) for Win32 desktop applications.

## Tutorial11



This tutorial will emphasize on the capabilities of the vertex shader. It is meant to show the users the possibilities of allowing manipulation of vertices.

The outcome of this tutorial will be a wave effect imposed on the character model from the previous tutorial. This effect is done entirely by the GPU with no CPU interaction with the data.

### Vertex Shader

While previous tutorials already have shown the use of the vertex shader, this tutorial is meant to emphasize this stage in the pipeline. Vertices contain various types of information such as coordinates, normals, texture coordinates, materials, colors, and even custom data. For example, if you look under the ParticlesGS sample, you will see that each vertex passed into the pipeline will also contain a variable called Timer, which can vary the appearance of the particle over time.

The purpose of the vertex shader is to offload intensive calculations from the CPU, onto the GPU. This frees up the CPU to accomplish other tasks within the application. For example, games often offload the graphics processing to the GPU, while the AI and physics is done on the CPU itself.

This tutorial is meant to show the possibilities of the vertex shader, rather than to teach specific techniques. It is encouraged that you try new effects and experiment with the results. For specific methods to achieve effects, refer to the samples included with the SDK. However, do note that they are more complex in nature and apply much more advanced concepts.

### Creating the Wave Effect

The wave effect on the model is created by modulating the x position of each point by a sine wave. The sine wave is controlled by the y position, thus creating a wave along the model. However, to make the wave move and appear animated, the sine wave is also modulated by a time variable.

Finally, there is a variable to control the amount of displacement by the vertex shader, and that is the waviness variable specified. This variable is passed into the shader and controlled by a GUI slider.

    output.Pos.x += sin( output.Pos.y\*0.1f + Time )\*Waviness;

After this manipulation is done, the vertex shader will still have to prepare the vertices for display. Thus, there is still the transformation of the world, view, and projection matrices. However, we chose to do the world transformation before the wave effect, as it would be easier to determine the effects on the screen, since the x and y axis are apparent.

    output.Pos = mul( output.Pos, View );  
    output.Pos = mul( output.Pos, Projection );

## Tutorial12



This tutorial focuses on the pixel shader and its capabilities. There are many things that can be accomplished with the pixel shader, and some of the more common functions are listed. The pixel shader will apply an environment map to the object.

The outcome of the tutorial is that the model will become shiny, and it will reflect its surrounding environment.

### Pixel Shaders

Pixel shaders are used to manipulate the final color of a pixel before it reaches the output merger. Each pixel that is rendered will be passed through this shader before output. Once the pixel has passed through the pixel shader, the only operations that can be performed are those performed by the output merger, such as alpha blending, depth testing, stencil testing, and so on.

The pixel shader evolved from the texture mapping found in early hardware. Instead of just a simple texture lookup, it is possible to compute the final color from multiple sources, as well as alter it according to the vertex data. For general applications, however, a pixel shader does multiple lookups on different textures.

### Environment Map

On surfaces which are reflective, it is not possible to just do a texture lookup on fixed texture coordinates to determine the final color. This is because whenever the object or viewer moves, the reflection alters. Thus, we must update reflection every time something moves.

An effective method to trick the observer to believe it's a dynamic reflection of the environment is to generate a special texture that can wrap around the object. This texture is referred as a cube map. A cube map is effectively placing an object in the middle of a cube, with each face of the cube being a texture. An environment map is a cube map which has the textures which correspond to the view of the environment on that face. If the environment is static, these environment maps can be pre-generated. If the environment is dynamic, the cubemap needs to be updated on-the-fly. See the CubeMapGS sample for an illustration of how to do this.

From this environment map, we can calculate what a camera will see as the reflection. First we can find the direction the camera is viewing the object, and from that, reflect it off the normal of each pixel and perform a lookup based on that reflected vector.

### Setting up the Environment Map

The setup of the environment map is not the emphasis of this tutorial. The procedure to follow is very similar to that of a normal texture map. Please refer to Tutorial 7, Texture Mapping and Constant Buffers, for an explanation of how to properly initialize a texture map and its associated resource view.

    // Load the Environment Map  
    ID3D11Resource \*pResource = nullptr;  
    V\_RETURN( DXUTFindDXSDKMediaFileCch( str, MAX\_PATH, L"Lobby\\LobbyCube.dds" ) );  
    V\_RETURN( DXUTCreateTextureFromFile( pd3dDevice, str, NULL, NULL, &pResource ) );  
    if(pResource)  
    {  
        g\_pEnvMap = (ID3D11Texture2D\*)pResource;  
        pResource->Release();  
        D3D11\_TEXTURECUBE\_DESC desc;  
        g\_pEnvMap->GetDesc( &desc );  
        D3D11\_SHADER\_RESOURCE\_VIEW\_DESC SRVDesc;  
        ZeroMemory( &SRVDesc, sizeof(SRVDesc) );  
        SRVDesc.Format = desc.Format;  
        SRVDesc.ViewDimension = D3D11\_SRV\_DIMENSION\_TEXTURECUBE;  
        SRVDesc.TextureCube.MipLevels = desc.MipLevels;  
        SRVDesc.TextureCube.MostDetailedMip = 0;  
        V\_RETURN(pd3dDevice->CreateShaderResourceView( g\_pEnvMap, &SRVDesc, &g\_pEnvMapSRV ));  
    }

    // Set the Environment Map  
    g\_pEnvMapVariable->SetResource( g\_pEnvMapSRV );

### Implementing the Lookup

We will step through the simple algorithm described above and perform a proper lookup of an environment map. The calculations are done in the vertex shader and then interpolated to the pixel shader for the lookup. It is better to compute it in the vertex shader and interpolate to the pixel shader since there are fewer computations necessary.

To compute the reflected vector for the lookup, we require two pieces of information. First is the normal of the pixel in question, second is the direction of the eye.

Since the operation is done in view space, we must transform the pixel's normal to proper viewing space.

    float3 viewNorm = mul( input.Norm, (float3x3)View );

Next we need to find the direction of the camera. Note, however, that we are already in view space, and thus, the camera's direction is that of the Z-axis (0,0,-1.0), since we are staring directly at the object.

Now that we have our two pieces of information, we can calculate the reflection of the vector with the reflect command. The variable ViewR is used to store the resulting reflection for processing in the pixel shader.

    output.ViewR = reflect( viewNorm, float3(0,0,-1.0) );

Once the correct vector has been calculated in the vertex shader, we can process the environment map in the pixel shader. This is done by calling a built-in function to sample the environment map and return the color value from that texture.

    // Load the environment map texture  
    float4 cReflect = g\_txEnvMap.Sample( samLinearClamp, viewR );

Since it is all normalized (and texture coordinates range from 0 to 1), the x and y coordinates will correspond directly to the texture coordinates that it needs to look up.

As a bonus, we included the code to do the original flat texture lookup, and computed that as the diffuse term. To play with the amount of reflection, you can modulate cReflect by a scaling factor against cDiffuse and experiment with the results. You can have a very reflective character or a somewhat dull character.

## Tutorial13



This tutorial will explore a part of the graphics pipeline that has not been touched in the previous tutorials. We will be touching upon some basic geometry shader functionality.

The outcome of this tutorial is that the model will have a second layer extruded from the model. Note that the original model is still preserved at the center.

### Geometry Shader

The benefit of the geometry shader (GS) is that it allows manipulation of meshes on a per-primitive basis. Instead of running a computation on each vertex individually, there is the option to operate on a per-primitive basis. That is, vertices can be passed in as a single vertex, a line segment (two vertices), or as a triangle (three vertices).

By allowing manipulation on a per-primitive level, new ideas can be approached, and there is more access to data to allow for that. In the tutorial, you will see that we have calculated the normal for the face. By knowing the position of all three vertices, we can find the face normal.

In addition to allowing access to whole primitives, the geometry shader can create new primitives on the fly. Previously in Direct3D, the graphics pipeline was only able to manipulate existing content, and it could amplify or deamplify data. The geometry shader in Direct3D can read in a single primitive (with optional edge-adjacent primitives) and emit zero, one, or multiple primitives based on that.

It is possible to emit a different type of geometry than the input source. For instance, it is possible to read in individual vertices, and generate multiple triangles based on those. This allows a wide range of new ideas to be executed on the graphics pipeline without CPU intervention.

The geometry shader exists between the vertex and the pixel shaders in the graphics pipeline. Since new geometry can potentially be created by the geometry shader, we must ensure that they are also properly transformed to projection space before we pass them off to the pixel shader (PS). This can either be done by the vertex shader (VS) before it enters the geometry shader, or it can be done within the geometry shader itself.

Finally, the output of the GS can be rerouted to a buffer. You can read in a bunch of triangles, generate new geometry, and store them in a new buffer. However, the concept of stream output is beyond the scope of tutorials, and it is best shown in the samples found in the SDK.

### Formatting a Geometry Shader

Unlike the VS and the PS, the geometry shader does not necessarily produce a static number of outputs per input. As such, the format to declare the shader is different from the other two.

The first parameter is maxvertexcount. This parameter describes the maximum number of vertices that can be output each time that the shader is run. This is followed by the name of the geometry shader, which has been aptly named GS.

The function name is followed by the parameters passed into the function. The first contains the keyword triangle, which specifies that the GS will operate on triangles as input. Following that is the vertex format, and the identifier (with the number signifying the size of the array, 3 for triangles, 2 for line segments). The second parameter is the output format and stream. TriangleStream signifies that the output will be in triangles (triangle strips to be exact), then the format is specified in the angled brackets. Finally, the identifier for the stream is denoted.

    [maxvertexcount(12)]  
    void GS( triangle GSPS\_INPUT input[3], inout TriangleStream<GSPS\_INPUT> TriStream )

If vertices start being emitted to a TriangleStream, it will assume that they are all linked together as a strip. To end a strip, call RestartStrip within the stream. To create a triangle list, you have to make sure that you call RestartStrip after every triangle.

### Exploding the Model

In this tutorial, we cover the basic functions of the geometry shader. To illustrate this, we will create an explosion effect on our model. This effect is created by extruding each vertex in the direction of that triangle's normal.

Note that a similar effect has been implemented in previous tutorials, whereby we extrude each vertex by its normal, controlled by the puffiness slider. This tutorial demonstrates the usage of the full triangle's information to generate the face normal. The difference of using the face normal is that you will see gaps between the exploded triangles. Because vertices on different triangles are shared, they will actually be passed twice into the geometry shader. Moreover, since each time it will extrude it in the normal of the triangle, as opposed to the vertex, the two final vertices may end up in different positions.

#### Calculating the face normal

To calculate the normal for any plane, we first need two vectors that reside on the plane. Since we are given a triangle, we can subtract any two vertices of the triangle to get the relative vectors. Once we have the vectors, we take the cross product to get the normal. We must also normalize the normal, since we will be scaling it later.

    //  
    // Calculate the face normal  
    //  
    float3 faceEdgeA = input[1].Pos - input[0].Pos;  
    float3 faceEdgeB = input[2].Pos - input[0].Pos;  
    float3 faceNormal = normalize( cross(faceEdgeA, faceEdgeB) );

Once we have the face normal, we can extrude each point of the triangle in that direction. To do so, we use a loop, which will step through three times and operate on each vertex. The position of the vertex is extruded by the normal, multiplied by a factor. Then, since the vertex shader has not transformed the vertices to proper projection space, we must also do that in the geometry shader. Finally, once we package the rest of the data, we can append this new vertex to our TriangleStream.

    for( int v=0; v<3; v++ )

    {

        output.Pos = input[v].Pos + float4(faceNormal\*Explode,0);

        output.Pos = mul( output.Pos, View );

        output.Pos = mul( output.Pos, Projection );

        output.Norm = input[v].Norm;

        output.Tex = input[v].Tex;  
          
        TriStream.Append( output );

    }

Once the three vertices have been emitted, we can cut the strip and restart. In this tutorial, we want to extrude each triangle separately, so we end up with a triangle list.

    TriStream.RestartStrip();

This new triangle stream is then sent to the pixel shader, which will operate on this data and draw it to the render target.

## Tutorial14



This tutorial will explore a very important, but often overlooked aspect of Direct3D 11 programming, state. While not as glamorous or as flashy as shaders, state changes are indispensibly important when it comes to graphics programming. In fact, the same shader can have drastically different visual results based solely upon the state of the device at the time of rendering.

In this tutorial we will explore 3 main types of state objects, BlendStates, DepthStencilStates, and RasterizerStates. At the end of the tutorial, you should have a much better understanding of the way in which states interact to produce different representations of the same scene.

### Blend States

Have you ever heard the term Alpha Blending? Alpha Blending involves modifying the color of a pixel being drawn to a certain screen location using the color of the pixel that already exists in that location. Blend States (ID3D11BlendState when used directly from the API and BlendState when used with FX) allow the developer to specify this interaction between new and old pixels. With the Blend State set to default, pixels just overwrite any pixels that already exist at the given screen coordinates during rasterization. Any information about the previous pixel that was drawn there is lost.

To give an example, pretend that your application is drawing the view of a city from inside a taxi cab. You've drawn thousands of buildings, sidewalks, telephone poles, trash cans, and other objects that make up a city. You have even drawn the interior of the cab. Now, as your last step, you want to draw the windows of the cab to make it look like as though you're actually peering through glass. Unfortunately, the pixels drawn during rasterization of the glass mesh completely overwrite the pixels already at those locations on the screen. This includes your beautiful city scene along with the sidewalks, telephone poles, and all of your other city objects. Wouldn't it be great if we could include the information that was already on the screen when drawing the glass windows of the cab? Wouldn't it also be great if we could do this with minimal changes in our current pixel shader code?

This is exactly what Blend States give you. To demonstrate this, our scene consists of a model and a quad.

    // Load the mesh  
    V\_RETURN( g\_Mesh.Create( pd3dDevice, L"Tiny\\tiny.x", (D3D11\_INPUT\_ELEMENT\_DESC\*)layout, 3 ) );

Note that our quad has a different vertex format and therefore needs a different input layout description

    // Create a screen quad  
    const D3D11\_INPUT\_ELEMENT\_DESC quadlayout[] =  
    {  
        { L"POSITION", 0, DXGI\_FORMAT\_R32G32B32A32\_FLOAT, 0, 0, D3D11\_INPUT\_PER\_VERTEX\_DATA, 0 },  
        { L"TEXCOORD0", 0, DXGI\_FORMAT\_R32G32\_FLOAT, 0, 16, D3D11\_INPUT\_PER\_VERTEX\_DATA, 0 },  
    };

    g\_pTechniqueQuad[0]->GetPassByIndex( 0 )->GetDesc( &PassDesc );  
    V\_RETURN( pd3dDevice->CreateInputLayout( quadlayout, 2, PassDesc.pIAInputSignature, &g\_pQuadLayout ) );

    ...

    D3D11\_SUBRESOURCE\_DATA InitData;  
    InitData.pSysMem = svQuad;  
    InitData.SysMemPitch = 0;  
    InitData.SysMemSlicePitch = 0;  
    V\_RETURN( pd3dDevice->CreateBuffer( &vbdesc, &InitData, &g\_pScreenQuadVB ) );

When we render our scene we see the model of a person, with a quad drawn over the top.

Imagine that this person represents our city and the quad is really a pane of glass. When we launch the application, the situation is exactly as we described it above, we can't see through the glass. However, by selecting a different Quad Render Mode from the drop down box, we can suddenly see through the quad. It's as if it's a plane of glass.

In fact, all that the application is doing when we select a different Quad Render Mode is changing which technique in the FX file we're using to render the quad. For example, RenderQuadSolid in the combo box refers to the following technique in the FX file.

    technique11 RenderQuadSolid  
    {  
        pass P0  
        {  
            SetVertexShader( CompileShader( vs\_4\_0, QuadVS() ) );  
            SetGeometryShader( NULL );  
            SetPixelShader( CompileShader( ps\_4\_0, QuadPS() ) );       
             
            SetBlendState( NoBlending, float4( 0.0f, 0.0f, 0.0f, 0.0f ), 0xFFFFFFFF );  
        }  
    }

The SetVertexShader, SetGeometryShader, and SetPixelShader functions should all look familiar at this point. If not, please review tutorials 10 through 12. The last line is what we're focusing on for now. In an FX file, this is how you set the Blend State. NoBlending actually refers to a state structure defined at the top of the FX file. The structure disables blending for the first (0) render target.

    BlendState NoBlending  
    {  
        BlendEnable[0] = FALSE;  
    };

As we said before, this Blend State does nothing interesting. Because alpha blending is disabled, the pixels of the quad simply overwrite the existing pixels placed on the screen when rendering the person. However, things get more interesting when we select RenderQuadSrcAlphaAdd from the Quad Render Mode combo box. This changes the quad to be rendered with the RenderQuadSrcAlphaAdd technique.

    technique11 RenderQuadSrcAlphaAdd  
    {  
        pass P0  
        {  
            SetVertexShader( CompileShader( vs\_4\_0, QuadVS() ) );  
            SetGeometryShader( NULL );  
            SetPixelShader( CompileShader( ps\_4\_0, QuadPS() ) );     
               
            SetBlendState( SrcAlphaBlendingAdd, float4( 0.0f, 0.0f, 0.0f, 0.0f ), 0xFFFFFFFF );  
        }  
    }

A close look at this technique reveals two differences between it and the RenderQuadSolid technique. The first is the name. The second, and vitally important, difference is the Blend State passed into SetBlendState. Instead of passing in the NoBlending Blend State, we're passing in the SrcAlphBlendingAdd Blend State which is defined at the top of the FX file.

    BlendState SrcAlphaBlendingAdd  
    {  
        BlendEnable[0] = TRUE;  
        SrcBlend = SRC\_ALPHA;  
        DestBlend = ONE;  
        BlendOp = ADD;  
        SrcBlendAlpha = ZERO;  
        DestBlendAlpha = ZERO;  
        BlendOpAlpha = ADD;  
        RenderTargetWriteMask[0] = 0x0F;  
    };

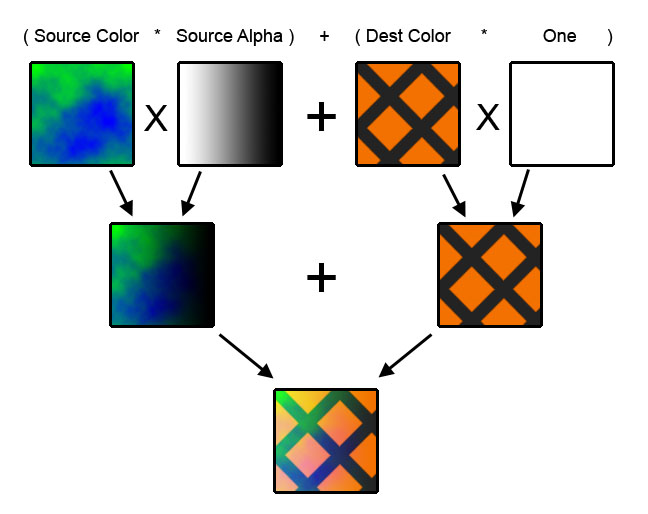
This blend state is a little more complex than the last one. Blending is enabled on the first (0) render target. For a thorough description of all Blend State parameters, refer to the documentation for D3D11\_BLEND\_DESC. Here is a quick overview of what this function is telling Direct3D 11 to do. When rendering the quad using this technique, the pixel about to be rendered to the framebuffer does not simply replace the existing pixel. Instead its color is modified using the following formula.

    outputPixel = ( SourceColor\*SourceBlendFactor ) BlendOp ( DestColor\*DestinationBlendFactor )

    SourceColor is a pixel being rendered when we draw the quad.  
    DestColor is the pixel that already exists in the framebuffer (from drawing the person)  
    BlendOp is the BlendOp from the SrcAlphaBlendingAdd blend structure  
    SourceBlendFactor is SrcBlend in the SrcAlphaBlendingAdd blend structure  
    DestinationBlendFactor is DestBlend in the SrcAlphaBlendingAdd blend structure

By using the SrcAlphaBlendingAdd structure above, we can translate the equation into the following

    OutputPixel = ( SourceColor.rgba \* SourceColor.aaaa ) ADD ( DestColor.rgba \* (1,1,1,1) )

Graphically this can be displayed by the following diagram.

Thus, the output color is dependent on the color that was already in the framebuffer at the time of drawing. Different selections in the combo box will result in the quad being draw with different techniques, and hence, different Blend States. Each Blend State changes the variables to the equation, so be sure to experiment with them all. See if you can predict what will happen just by looking at the Blend State.

### Rasterizer states

You may have noticed that no matter which Blend State was selected, the figure in the middle looked a little off. Some polygons were showing up where they shouldn't have been. Sometimes the legs looked as though you were looking at an upright tube cut in half. This brings us to our next state object, Rasterizer State.

The Rasterizer State ( ID3D11RasterizerState when used through the API and RasterizerState when used through the FX framework ) controls how the triangles are actually rendered on the screen. Unlike the previous section on Blend States, we are not going to control Rasterizer States through the FX interface. This does not mean that they cannot be used through the FX system. In fact, it's just as easy to use Rasterizer States through FX as it is to use Blend States through FX. However, we're going to take this opportunity to learn about non-FX state management using the Direct3D 11 APIs.

As the name implies, Rasterizer State controls how polygons are rasterized on the screen. When the tutorial starts up, the polygons are not culled. This means that where the polygons are pointing in your direction or pointing away from you, they all get draw equally. This is why some dark polygons are "poking" through in the figure. In order to change this behavior we must first create a Rasterizer State that defines the behavior we want. Since we're doing this through the API and not the FX interface the process is a little different than for Blend States. The following code is from the LoadRasterizerStates function.

    D3D11\_FILL\_MODE fill[MAX\_RASTERIZER\_MODES] =

    {

        D3D11\_FILL\_SOLID,

        D3D11\_FILL\_SOLID,

        D3D11\_FILL\_SOLID,

        D3D11\_FILL\_WIREFRAME,

        D3D11\_FILL\_WIREFRAME,

        D3D11\_FILL\_WIREFRAME

    };

    D3D11\_CULL\_MODE cull[MAX\_RASTERIZER\_MODES] =

    {

        D3D11\_CULL\_NONE,

        D3D11\_CULL\_FRONT,

        D3D11\_CULL\_BACK,

        D3D11\_CULL\_NONE,

        D3D11\_CULL\_FRONT,

        D3D11\_CULL\_BACK

    };

    for( UINT i=0; i<MAX\_RASTERIZER\_MODES; i++ )

    {

        D3D11\_RASTERIZER\_DESC rasterizerState;

        rasterizerState.FillMode = fill[i];

        rasterizerState.CullMode = cull[i];

        rasterizerState.FrontCounterClockwise = true;

        rasterizerState.DepthBias = false;

        rasterizerState.DepthBiasClamp = 0;

        rasterizerState.SlopeScaledDepthBias = 0;

        rasterizerState.DepthClipEnable = true;

        rasterizerState.ScissorEnable = false;

        rasterizerState.MultisampleEnable = false;

        rasterizerState.AntialiasedLineEnable = false;

        pd3dDevice->CreateRasterizerState( &rasterizerState, &g\_pRasterStates[i] );

        g\_SampleUI.GetComboBox(IDC\_SCENERASTERIZER\_MODE)->AddItem( g\_szRasterizerModes[i], (void\*)(UINT64)i );

    }

All this code is doing is filling in a D3D11\_RASTERIZER\_DESC structure with the necessary information and then calling ID3D11Device::CreateRasterizerState to get a pointer to an ID3D11RasterizerState object. The loop creates one state for each type of Rasterizer State we want to show off. Notice that the first state, which uses D3D11\_FILL\_SOLID as the fill mode and D3D11\_CULL\_NONE as the cull mode is what is causing our figure to look weird. To change this, we can select the second rasterizer state from Scene Rasterizer Mode combo box. This object can then be used to set the state before we render. This is how the currently selected Rasterizer State is set in OnD3D11FrameRender.

    //  
    // Update the Cull Mode (non-FX method)  
    //  
    pd3dDevice->RSSetState(g\_pRasterStates[ g\_eSceneRasterizerMode ]);

It must be noted any state set "persists" through all subsequent drawing operations until a different state is set or the device is destroyed. This means that when we set the rasterizer state in OnD3D11FrameRender, that same Rasterizer State is applied to any and all drawing operations that come after until a different Rasterizer State is set.

When we selected the second option from the Scene Rasterizer Mode combo box our figure got a lot worse and the quad just dissappeared.

This is because we selected a cull mode that told Direct3D not to draw triangles that are facing forward. The next Rasterizer State ( D3D11\_CULL\_BACK and D3D11\_FILL\_SOLID ) should give us the results we want ( with the exception of some stray black triangles around the head which we'll discuss in the next section ).

In addition to which triangles to draw and not draw, Rasterizer States can control many other aspects of rendering. Feel free to experiment with the different Scene Rasterizer Mode selections or to even change the code in LoadRasterizerStates to see what happens.

### Depth Stencil states

In the last section, setting the cull mode to D3D11\_CULL\_BACK and the fill mode to D3D11\_FILL\_SOLID gave a decent looking figure in the middle. However, there were still some black polygons showing up around the face. This is because those polygons were still classified as pointing to the front even though they were on the other side of the head. This is where the Depth Stencil State comes into play. This state object actually controls two different parts of the Direct3D 11 pipeline. The first is the Depth Buffer.

On a basic level the depth buffer stores values representing the distance of the pixel from the view plane. The greater the value, the greater the distance. The depth buffer is the same resolution (height and width) as the backbuffer. By itself, this doesn't seem very useful to our situation. However, the Depth Stencil State lets us modify the current pixel based upon the depth of the pixel already at that location in the framebuffer.

The Depth Stencil States are created in the LoadDepthStencilStates function. This function is similar to the LoadRasterizerStates. We will discuss some of the parameters of the D3D11\_DEPTH\_STENCIL\_DESC structure. The most important of which is the DepthEnable. This determines whether the Depth Test is even functioning. As we can see, when the app starts up, the Depth Test is disabled. This is why the front facing hair spikes on the far side of the head still show up.

Setting the Depth Stencil State is also very similar to setting the Rasterizer State with one exception. There is an extra parameter. This parameter is the StencilRef which will be explained in the next section.

    //  
    // Update the Depth Stencil States (non-FX method)  
    //  
    pd3dDevice->OMSetDepthStencilState(g\_pDepthStencilStates[ g\_eSceneDepthStencilMode ], 0);

The second Depth Stencil State in the Scene Depth/Stencil Mode combo enables the Depth Test. When you enable the Depth Test, you need to tell Direct3D what type of test to do. The equation boils down to the following.

    DrawThePixel? = DepthOfCurrentPixel D3D11\_COMPARISON CurrentDepthInDepthBuffer

D3D11\_COMPARISON is the comparison function in the DepthFunc parameter of the D3D11\_DEPTH\_STENCIL\_DESC

By substituting the value of the DepthFunc from the second Depth Stencil State, we get the following for the equation.

    DrawThePixel? = DepthOfCurrentPixel D3D11\_COMPARISON\_LESS CurrentDepthInDepthBuffer

In English, this means "Draw the current pixel only if it's depth is LESS than the depth already stored at this location." Additionally, we set the DepthWriteMask to D3D11\_DEPTH\_WRITE\_MASK\_ALL to ensure that our depth is put into the depth buffer at this location (should we pass the test) so that any subsequent pixels will have to be closer than our depth value to be drawn.

This results in the figure being drawn without any black marks. The forward facing hair spikes in the rear never show through. They are either overwritten by closer triangles in the near side of the face, or they are never drawn because the near triangles have a closer depth value. You will also notice that this reveals something about the quad. All this time it looked like it was in front of the figure. Now, we realize that it is actually intersecting the figure.

Try playing around with the DepthFunc values and see what happens.

### Rendering Using the Stencil Buffer

The Stencil parameters are part of the Depth Stencil State, but, because of their complexity, they deserve their own section.

The Stencil buffer can be thought of as a buffer like the Depth Buffer. Like the depth buffer, it has the same dimensions as the Render Target. However, instead of storing depth values, the stencil buffer stores integer values. These values have meaning only to the application. The Stencil part of the Depth Stencil State determines how these values get there and what they mean to the application.

Like the Depth Buffer's Depth Test, the Stencil Buffer has a Stencil Test. This Stencil Test can compare the incoming value of the pixel to the current value in the Stencil Buffer. However, it can also augment this comparison with results from the Depth Test. Let's go through one of the Stencil Test setups to see what this means. The following information is from the fifth iteration of the loop in the LoadDepthStencilStates funciton. This reflects the 5th Depth Stencil State in the Scene Depth/Stencil Mode combo box.

    D3D11\_DEPTH\_STENCIL\_DESC dsDesc;  
    dsDesc.DepthEnable = TRUE;  
    dsDesc.DepthWriteMask = D3D11\_DEPTH\_WRITE\_MASK\_ALL;  
    dsDesc.DepthFunc = D3D11\_COMPARISON\_LESS;

    dsDesc.StencilEnable = TRUE  
    dsDesc.StencilReadMask = 0xFF;  
    dsDesc.StencilWriteMask = 0xFF;

    // Stencil operations if pixel is front-facing  
    dsDesc.FrontFace.StencilFailOp = D3D11\_STENCIL\_OP\_KEEP;  
    dsDesc.FrontFace.StencilDepthFailOp = D3D11\_STENCIL\_OP\_INCR;  
    dsDesc.FrontFace.StencilPassOp = D3D11\_STENCIL\_OP\_KEEP;  
    dsDesc.FrontFace.StencilFunc = D3D11\_COMPARISON\_ALWAYS;

    // Stencil operations if pixel is back-facing  
    dsDesc.BackFace.StencilFailOp = D3D11\_STENCIL\_OP\_KEEP;  
    dsDesc.BackFace.StencilDepthFailOp = D3D11\_STENCIL\_OP\_INC;  
    dsDesc.BackFace.StencilPassOp = D3D11\_STENCIL\_OP\_KEEP;  
    dsDesc.BackFace.StencilFunc = D3D11\_COMPARISON\_ALWAYS;

From the previous section, you should know that the Depth Test is set to let any pixel through that is closer to the plane than the previous depth value at that location. However, now, we've enabled the Stencil Test by setting StencilEnable to TRUE. We've also set the StencilReadMask and StencilWriteMask to values that ensure we read from and write to all bits of the stencil buffer.

Now comes the meat of the Stencil settings. The stencil operation can have different effects depending on whether the triangle being rasterized is front facing or back facing. (Remember that we can select to draw only front or back facing polygons using the Rasterizer State.) For simplicity, we're going to use the same operations for both front and back facing polygons, so we'll only discuss the FrontFace operations. The FrontFace.StencilFailOp is set to D3D11\_STENCILL\_OP\_KEEP. This means that if the Stencil Test fails, we keep the current value in the stencil buffer. The FrontFace.StencilDepthFailOp is set to D3D11\_STENCIL\_OP\_INC. This means that if we fail the depth test, we increment the value in the stencil buffer by 1. The FrontFace.STencilPassOp is set to D3D11\_STENCIL\_OP\_KEEP, meaning that on passing the Stencil Test we keep the current value. Finally FrontFace.StencilFunc represents the comparison used to determine if we actually pass the Stencil Test (not the Depth Test). This can be any of the D3D11\_COMPARISON values. For example, D3D11\_COMPARISON\_LESS would only pass the Stencil Test if the current stencil value was less than the incumbant stencil value in the Stencil Buffer. But where does this current test value come from? It's the second parameter to the OMSetDepthStencilState function (the StencilRef parameter).

    //  
    // Update the Depth Stencil States (non-FX method)  
    //  
    pd3dDevice->OMSetDepthStencilState(g\_pDepthStencilStates[ g\_eSceneDepthStencilMode ], 0);

For this example, the StencilRef parameter doesn't matter in the stencil comparison function. Why? Because we've set the StencilFunc to D3D11\_COMPARISON\_ALWAYS, meaning we ALWAYS pass the test.

What you may have already determined from looking at this Depth Stencil State is that any pixel that FAILS the Depth Test will cause the value in the Stencil Buffer to increment (for that location in the buffer). If we fill the stencil buffer with zeroes (which we do at the beginning of OnD3D11FrameRender), then any non-zero value in the stencil buffer is a place where a pixel failed the depth test.

If we could actually visualize this Stencil Buffer, we could determine all of the places in the scene where we attempted to draw something, but something else was closer. In essence, we could tell where we tried to draw something behind an object already there.

Unfortunately, there is no Direct3D 11 function called ShowMeTheStencilBuffer. Fortunately, we can create an FX Technique that does the same thing. To show that the Depth Stencil State can be just as easily set through the FX system as it can be through the Direct3D 11 APIs, we'll show how to use a Depth Stencil State in FX to view the contents of the stencil buffer.

After both the figure and the quad have been drawn, we render another quad that covers the entire viewport. Because we cover the entire viewport, we can access every pixel that could possibly be touched by our previous rendering operations. The technique used to render this quad is RenderWithStencil in the FX file.

    technique11 RenderWithStencil  
    {  
        pass P0  
        {  
            SetVertexShader( CompileShader( vs\_4\_0, ScreenQuadVS() ) );  
            SetGeometryShader( NULL );  
            SetPixelShader( CompileShader( ps\_4\_0, QuadPS() ) );       
             
           SetBlendState( NoBlending, float4( 0.0f, 0.0f, 0.0f, 0.0f ), 0xFFFFFFFF );  
            SetDepthStencilState( RenderWithStencilState, 0 );  
        }  
    }

Notice that in addition to setting the Blend State, we also set the Depth Stencil State to RenderWithStencilState. The second parameter sets the StencilRef value to 0. Let's take a look at the RenderWithStencilState as defined at the top of the FX file.

    DepthStencilState RenderWithStencilState

    {

        DepthEnable = false;

        DepthWriteMask = ZERO;

        DepthFunc = Less;

        // Setup stencil states

        StencilEnable = true;

        StencilReadMask = 0xFF;

        StencilWriteMask = 0x00;

        FrontFaceStencilFunc = Not\_Equal;

        FrontFaceStencilPass = Keep;

        FrontFaceStencilFail = Zero;

        BackFaceStencilFunc = Not\_Equal;

        BackFaceStencilPass = Keep;

        BackFaceStencilFail = Zero;

    };

The first thing you should notice is that this is similar to the D3D11\_DEPTH\_STENCIL\_DESC you'd find in the LoadDepthStencilStates function in the cpp file. The values are different, but the members are the same. First, we disable the depth test. Secondly, we enable the Stencil Test, BUT we set our StencilWriteMask to 0. We only want to read the stencil value. Last we set the StencilFunc to Not\_Equal, the StencilPass to Keep, and the StencilFail to Zero for both front and back faces. Remember that the Stencil Test tests the current incoming value with the value already stored in the Stencil Buffer. In this case, we set the incoming value (the StencilRef) to 0 when we called SetDepthStencilState( RenderWithStencilState, 0 ); Thus, we can translate this state to mean that the stencil test passes where the Stencil Buffer is not equal to 0 (StencilRef). On pass we get to Keep the incoming pixel from the pixel shader and write it out to the frame buffer. On a fail, we discard the incoming pixel, and it never gets drawn. Thus, our screen covering quad is only drawn in places where the Stencil Buffer is non-zero.

Selecting different items in the Scene Depth/Stencil Mode combo box change how the Depth and Stencil buffers are filled, and thus how much of the Screen Sized quad that's drawn last actually makes it onto the screen. Some of these combinations won't produce any output. Some will produce bizarre effects. Try to figure out which ones will do what before you apply them.

# Dependencies

DXUT-based samples typically make use of runtime HLSL compilation. Build-time compilation is recommended for all production Direct3D applications, but for experimentation and samples development runtime HLSL compilation is preferred. Effects11-based samples also need the D3DCompile APIs for reflection. Therefore, the D3DCompile\*.DLL must be available in the search path when these programs are executed.

* When using the Windows 10 SDK and targeting Windows 7 or later, you can include the D3DCompile\_47 DLL side-by-side with your application copying the file from the REDIST folder.

%ProgramFiles(x86)%\Windows kits\10\Redist\D3D\ x86 or x64

# More Information

[Where is the DirectX SDK (2021 Edition)?](https://aka.ms/dxsdk)

[DXUT for Win32 Desktop Update](https://walbourn.github.io/dxut-for-win32-desktop-update/)

[Effects for Direct3D 11 Update](https://walbourn.github.io/effects-for-direct3d-11-update/)

[Games for Windows and DirectX SDK blog](https://walbourn.github.io/)