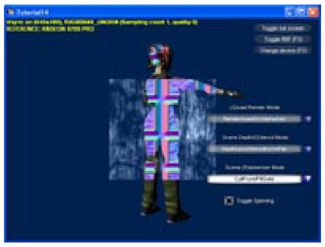
Tutorial 14: State Management

⊟ Collapse All



Summary

This tutorial will explore a very important, but often overlooked aspect of Direct3D 10 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.

Source

(SDK root)\Samples\C++\Direct3D10\Tutorials\Tutorial14

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 (ID3D10BlendState 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?

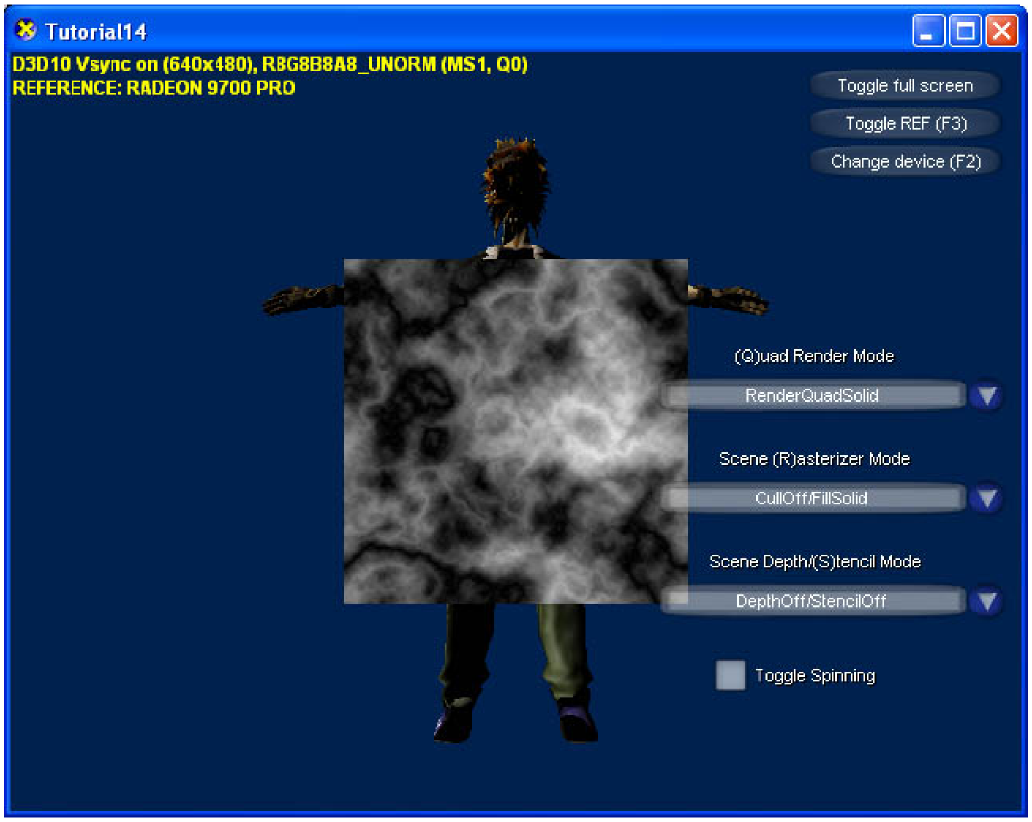
This is exactly what Blend States give you. To demonstrate this, our scene consists of a model and a quad.

```
// Load the mesh
V_RETURN( g_Mesh.Create( pd3dDevice, L"Tiny\\tiny.x", (D3D10_INPUT_ELEMENT_DESC*)layout, 3 ) );
```

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

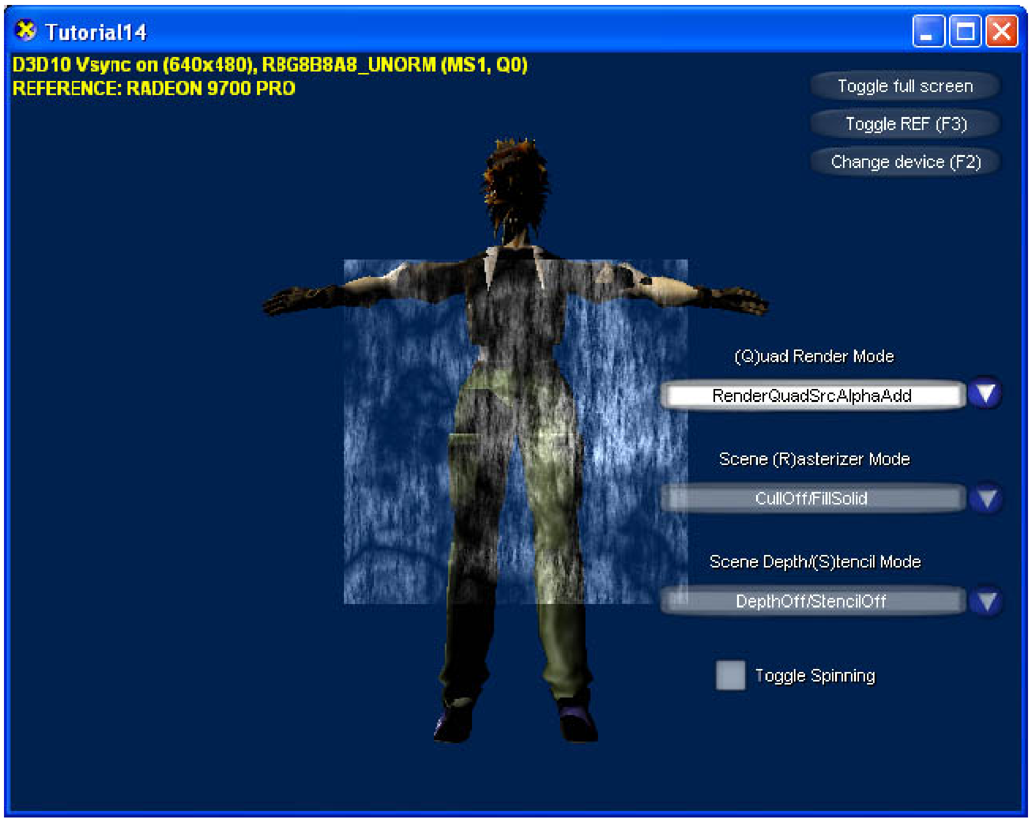
```
// Create a screen quad
const D3D10_INPUT_ELEMENT_DESC quadlayout[] =
{
    { L"POSITION", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0, 0, D3D10_INPUT_PER_VERTEX_DATA, 0 },
    { L"TEXCOORD0", 0, DXGI_FORMAT_R32G32_FLOAT, 0, 16, D3D10_INPUT_PER_VERTEX_DATA, 0 },
};

g_pTechniqueQuad[0]->GetPassByIndex( 0 )->GetDesc( &PassDesc );
V_RETURN( pd3dDevice->CreateInputLayout( quadlayout, 2, PassDesc.pIAInputSignature, &g_pQuadLayout ) );

...

D3D10_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.



D3D10 Vsync on (640x480), R8G8B8A8_UNORM (MS1, Q0)
REFERENCE: RADEON 9700 PRO

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.



D3D10 Vsync on (640x480), R8G8B8A8_UNORM (MS1, Q0)
REFERENCE: RADEON 9700 PRO

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.

```
    technique10 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.

```
technique10 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;
};
```

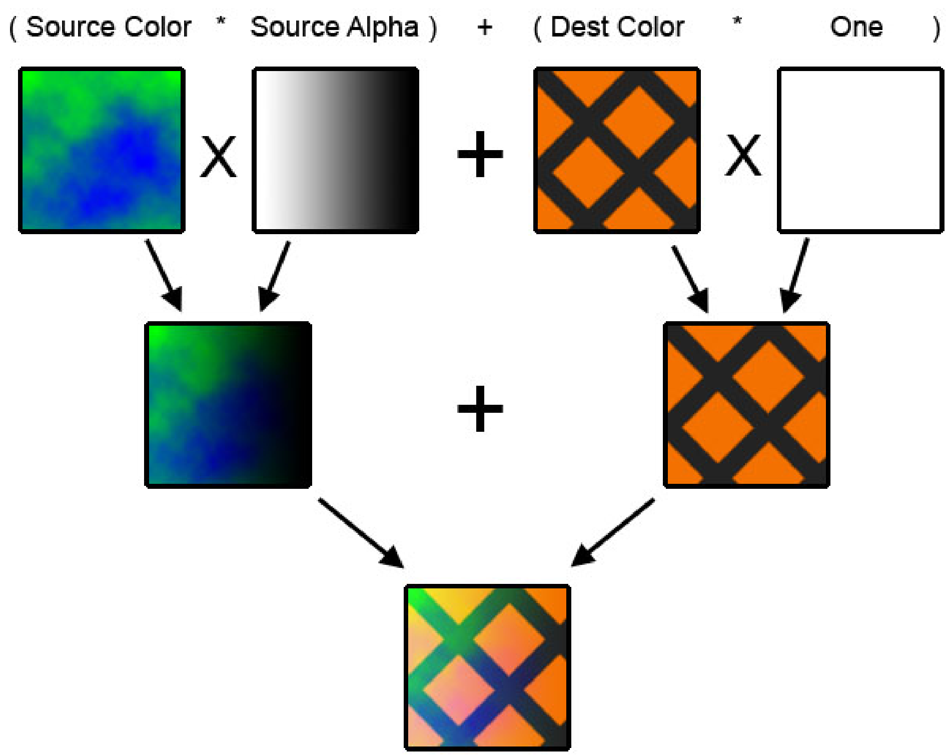
This blend state is a little more complex than the last one. Blending is enabled on the first (0) render target. For a thorough description of all Blend State parameters, refer to the documentation for **D3D10_BLEND_DESC**. Here is a quick overview of what this function is telling Direct3D 10 to do. 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 (ID3D10RasterizerState 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 10 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.

```
D3D10_FILL_MODE fill[MAX_RASTERIZER_MODES] =
{
    D3D10_FILL_SOLID,
    D3D10_FILL_SOLID,
    D3D10_FILL_SOLID,
    D3D10_FILL_WIREFRAME,
    D3D10_FILL_WIREFRAME,
    D3D10_FILL_WIREFRAME
};
D3D10_CULL_MODE cull[MAX_RASTERIZER_MODES] =
{
    D3D10_CULL_NONE,
    D3D10_CULL_FRONT,
    D3D10_CULL_BACK,
    D3D10_CULL_NONE,
    D3D10_CULL_FRONT,
    D3D10_CULL_BACK
};

for( UINT i=0; i<MAX_RASTERIZER_MODES; i++ )
{
    D3D10_RASTERIZER_DESC rasterizerState;
    rasterizerState.FillMode = fill[i];
    rasterizerState.CullMode = cull[i];
```

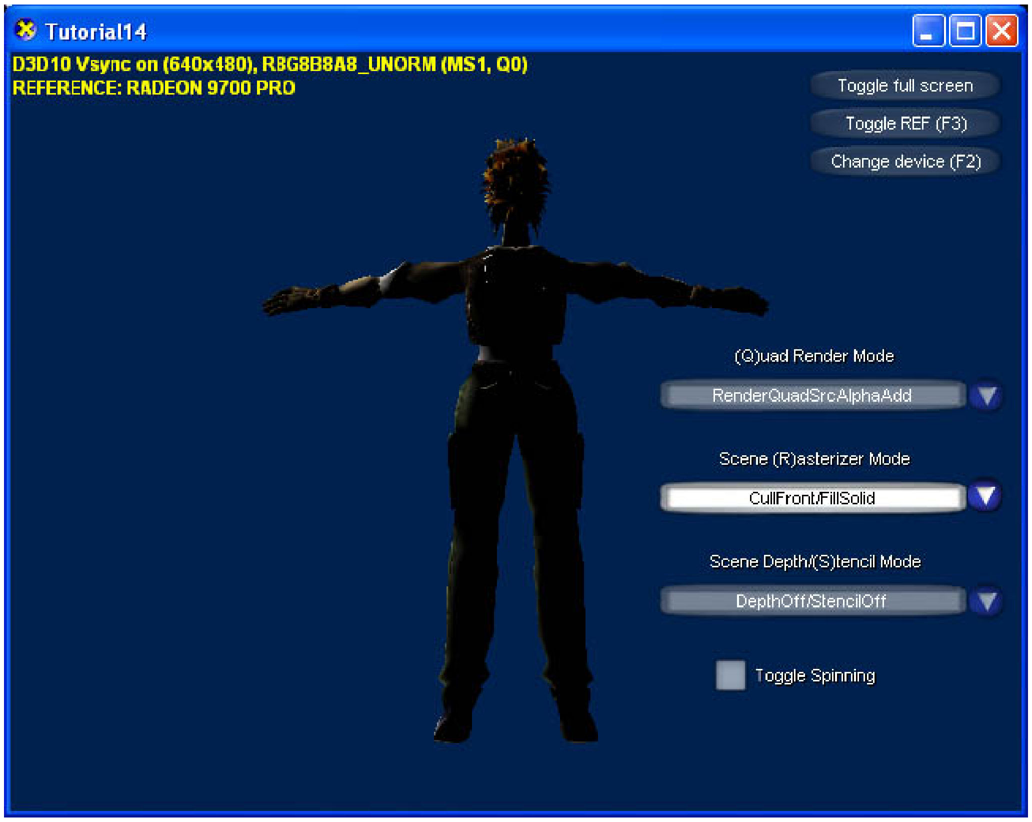
```
        rasterizerState.FrontCounterClockwise = true;
        rasterizerState.DepthBias = false;
        rasterizerState.DepthBiasClamp = 0;
        rasterizerState.SlopeScaledDepthBias = 0;
        rasterizerState.DepthClipEnable = true;
        rasterizerState.ScissorEnable = false;
        rasterizerState.MultisampleEnable = 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 D3D10_RASTERIZER_DESC structure with the necessary information and then calling ID3D10Device::CreateRasterizerState to get a pointer to an ID3D10RasterizerState object. The loop creates one state for each type of Rasterizer State we want to show off. Notice that the first state, which uses D3D10_FILL_SOLID as the fill mode and D3D10_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 OnD3D10FrameRender.

```
    //
    // 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 OnD3D10FrameRender, that same Rasterizer State is applied to any and all drawing operations that come after until a different Rasterizer State is set.

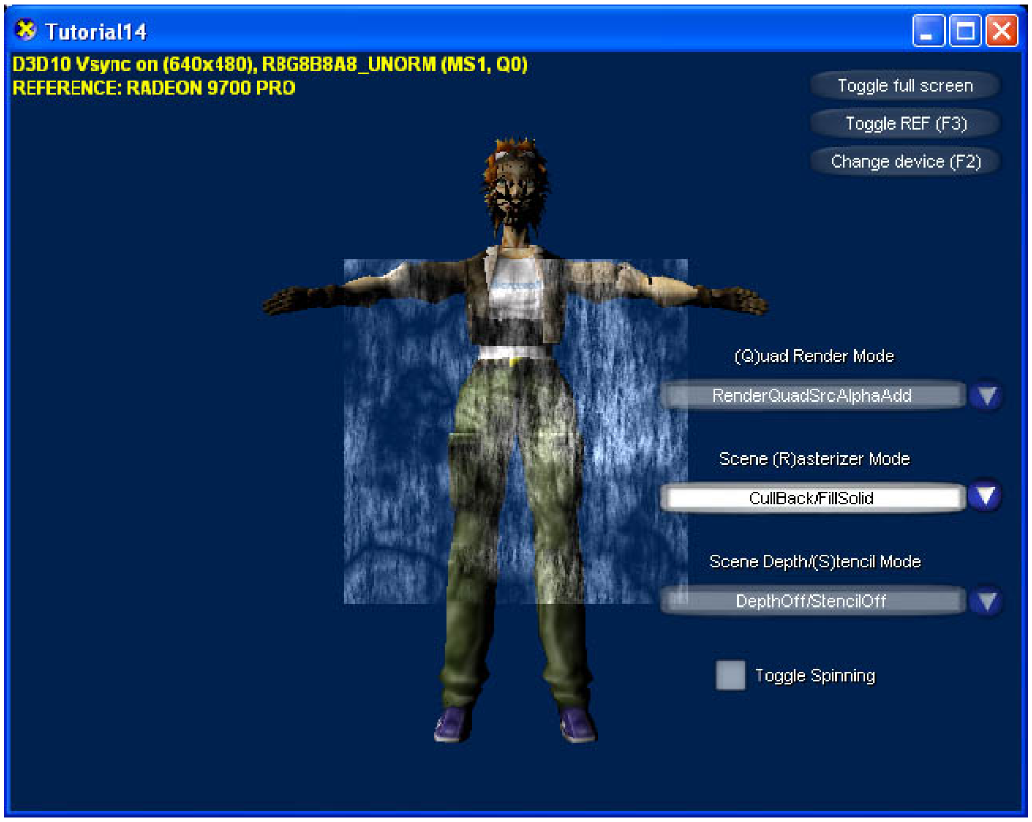
When we selected the second option from the Scene Rasterizer Mode combo box our figure got a lot worse and the quad just dissappeared.



D3D10 Vsync on (640x480), R8G8B8A8_UNORM (MS1, Q0)
REFERENCE: RADEON 9700 PRO

(Q)uad Render Mode: RenderQuadSrcAlphaAdd
Scene (R)asterizer Mode: CullFront/FillSolid
Scene Depth/(S)tencil Mode: DepthOff/StencilOff

This is because we selected a cull mode that told Direct3D 10 not to draw triangles that are facing forward. The next Rasterizer State (D3D10_CULL_BACK and D3D10_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).



D3D10 Vsync on (640x480), R8G8B8A8_UNORM (MS1, Q0)
REFERENCE: RADEON 9700 PRO

(Q)uad Render Mode: RenderQuadSrcAlphaAdd
Scene (R)asterizer Mode: CullBack/FillSolid
Scene Depth/(S)tencil Mode: DepthOff/StencilOff

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 D3D10_CULL_BACK and the fill mode to D3D10_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 10 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 D3D10_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 10 what type of test to do. The equation boils down to the following.

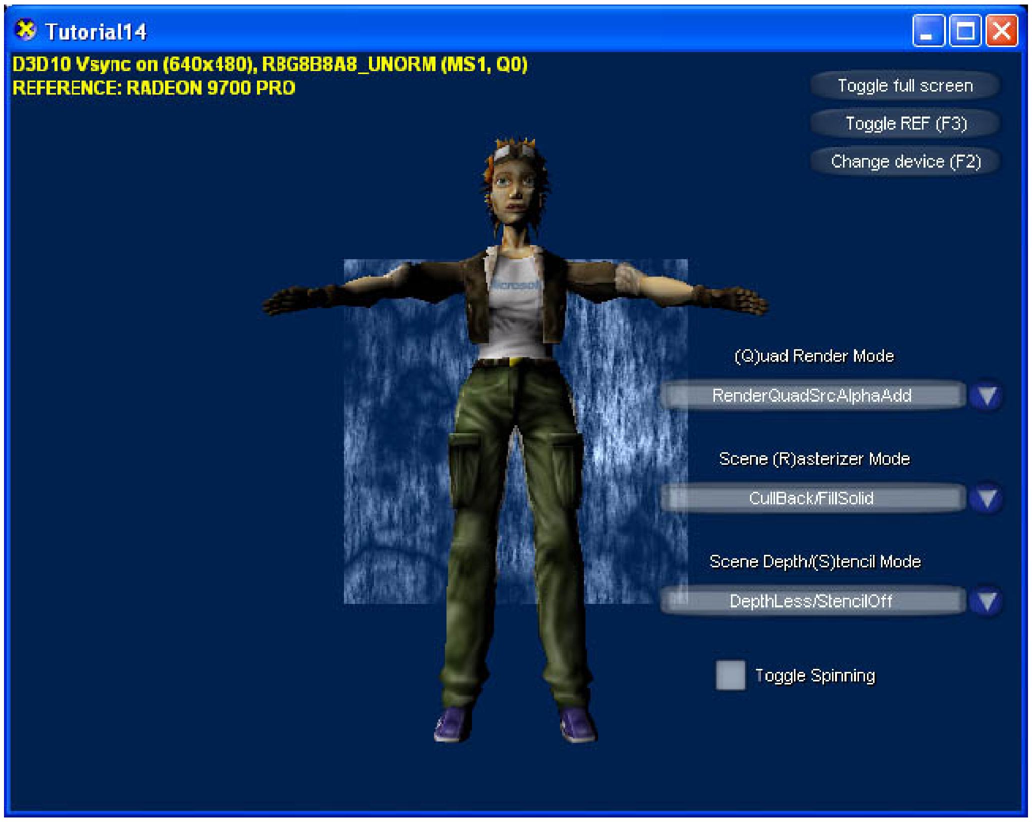
```
DrawThePixel? = DepthOfCurrentPixel D3D10_COMPARISON CurrentDepthInDepthBuffer

D3D10_COMPARISON is the comparison function in the DepthFunc parameter of the D3D10_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 D3D10_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 D3D10_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.

```
D3D10_DEPTH_STENCIL_DESC dsDesc;
dsDesc.DepthEnable = TRUE;
dsDesc.DepthWriteMask = D3D10_DEPTH_WRITE_MASK_ALL;
dsDesc.DepthFunc = D3D10_COMPARISON_LESS;

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

// Stencil operations if pixel is front-facing
dsDesc.FrontFace.StencilFailOp = D3D10_STENCIL_OP_KEEP;
dsDesc.FrontFace.StencilDepthFailOp = D3D10_STENCIL_OP_INCR;
dsDesc.FrontFace.StencilPassOp = D3D10_STENCIL_OP_KEEP;
dsDesc.FrontFace.StencilFunc = D3D10_COMPARISON_ALWAYS;

// Stencil operations if pixel is back-facing
dsDesc.BackFace.StencilFailOp = D3D10_STENCIL_OP_KEEP;
dsDesc.BackFace.StencilDepthFailOp = D3D10_STENCIL_OP_INC;
dsDesc.BackFace.StencilPassOp = D3D10_STENCIL_OP_KEEP;
dsDesc.BackFace.StencilFunc = D3D10_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 D3D10_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 D3D10_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 D3D10_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 D3D10_COMPARISON values. For example, D3D10_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 D3D10_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 OnD3D10FrameRender), 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 10 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 10 APIs, we'll show how to use a Depth Stencil State in FX to view the contents of the stencil buffer.

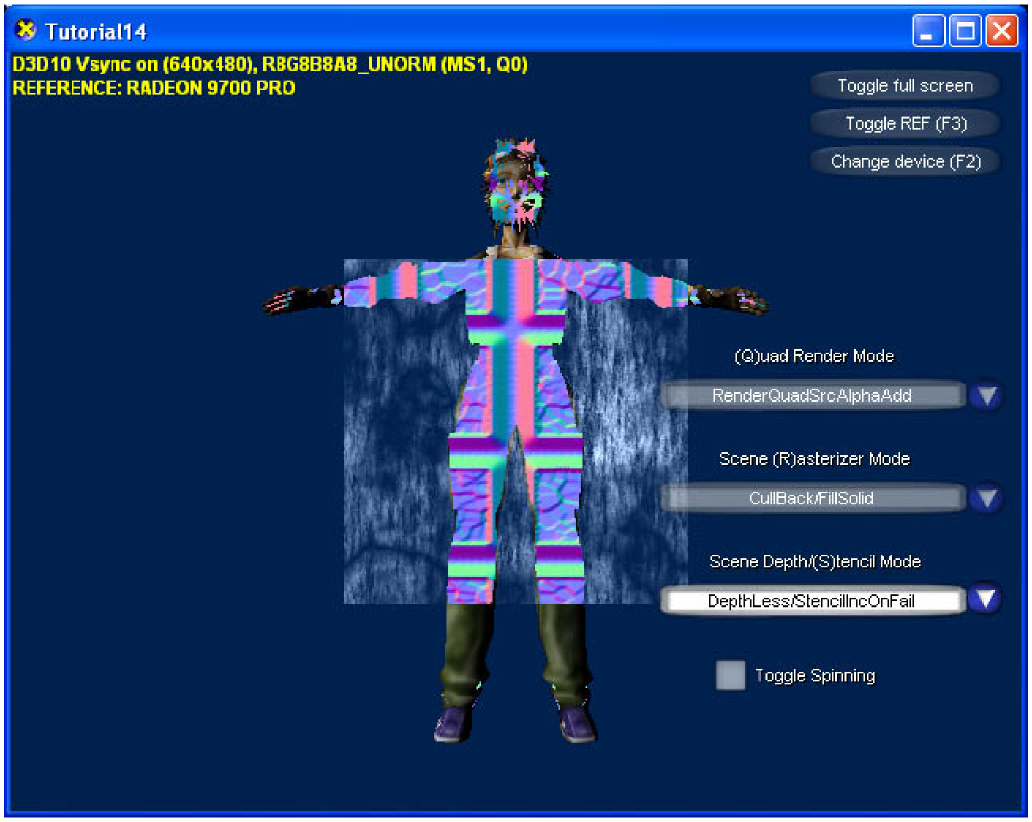
After both the figure and the quad have been drawn, we render another quad that covers the entire viewport. Because we cover the entire viewport, we can access every pixel that could possibly be touched by our previous rendering operations. The technique used to render this quad is RenderWithStencil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

```
technique10 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 D3D10_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.



D3D10 Vsync on (640x480), R8G8B8A8_UNORM (MS1, Q0)
REFERENCE: RADEON 9700 PRO

(Q)uad Render Mode: RenderQuadSrcAlphaAdd

Scene (R)asterizer Mode: CullBack/FillSolid

Scene Depth/(S)tencil Mode: DepthLess/StencilIncOnFail

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.