# Tutorial 2: Rendering Vertices

Applications written in Direct3D use vertices to draw geometric shapes. Each three-dimensional (3D) scene includes one or more of these geometric shapes. The Vertices sample project creates the simplest shape, a triangle, and renders it to the display.

This tutorial shows how to use vertices to create a triangle with the following steps:

## Steps

- Step 1 - Defining a Custom Vertex Type
- Step 2 - Setting Up the Vertex Buffer
- Step 3 - Rendering the Display

**Note**

The path of the Vertices sample project is:

(*SDK root*)\Samples\C++\Direct3D\Tutorials\Tut02_Vertices

The sample code in the Vertices project is nearly identical to the sample code in the CreateDevice project. The Rendering Vertices tutorial focuses only on the code unique to vertices and does not cover initializing Direct3D, handling Windows messages, rendering, or shutting down.

## Step 1 - Defining a Custom Vertex Type

The Vertices sample project renders a 2D triangle by using three vertices. This introduces the concept of the vertex buffer, which is a Direct3D object that is used to store and render vertices. Vertices can be defined in many ways by specifying a custom vertex structure and corresponding custom flexible vertex format (FVF). The format of the vertices in the Vertices sample project is shown in the following code fragment:

```
struct CUSTOMVERTEX
{
    FLOAT x, y, z, rhw; // The transformed position for the vertex.
    DWORD color;        // The vertex color.
};
```

The structure above specifies the format of the custom vertex type. The next step is to define the FVF that describes the contents of the vertices in the vertex buffer. The following code fragment defines an FVF that corresponds with the custom vertex type created above.

```
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZRHW|D3DFVF_DIFFUSE)
```

**D3DFVF** describes what type of custom vertex is being used. The sample code above uses the D3DFVF_XYZRHW and D3DFVF_DIFFUSE flags, which tell the vertex buffer that the custom vertex type has a transformed point followed by a color component.

Now that the custom vector format and FVF are specified, the next step is to fill the vertex buffer with vertices, as described in Step 2 - Setting Up the Vertex Buffer.

**Note**

The vertices in the Vertices sample project are transformed. In other words, they are already in 2D window coordinates. This means that the point (0,0) is at the top left corner and the positive x-axis is right and the positive y-axis is down. These vertices are also lit, meaning that they are not using Direct3D lighting but are supplying their own color.

## Step 2 - Setting Up the Vertex Buffer

Now that the custom vertex format is defined, it is time to initialize the vertices. The Vertices sample project does this by calling the application-defined function InitVB after creating the required Direct3D objects. The following code fragment initializes the values for three custom vertices.

```
CUSTOMVERTEX vertices[] =
{
    { 150.0f,  50.0f, 0.5f, 1.0f, 0xffff0000, }, // x, y, z, rhw, color
    { 250.0f, 250.0f, 0.5f, 1.0f, 0xff00ff00, },
    {  50.0f, 250.0f, 0.5f, 1.0f, 0xff00ffff, },
};
```

The preceding code fragment fills three vertices with the points of a triangle and specifies which color each vertex will emit. The first point is at (150, 50) and emits the color red (0xffff0000). The second point is at (250, 250) and emits the color green (0xff00ff00). The third point is at (50, 250) and emits the color blue-green (0xff00ffff). Each of these points has a depth value of 0.5 and an RHW of 1.0.

The next step is to call **IDirect3DDevice9::CreateVertexBuffer** to create a vertex buffer as shown in the following code fragment.

```
if( FAILED( g_pd3dDevice->CreateVertexBuffer( 3*sizeof(CUSTOMVERTEX),
         0 /*Usage*/, D3DFVF_CUSTOMVERTEX, D3DPOOL_DEFAULT, &g_pVB, NULL ) ) )
    return E_FAIL;
```

The first two parameters of **IDirect3DDevice9::CreateVertexBuffer** tell Direct3D the desired size and usage for the new vertex buffer. The next two parameters specify the vector format and memory location for the new buffer. The vector format here is D3DFVF_CUSTOMVERTEX, which is the flexible vertex format (FVF) that the sample code specified earlier. The D3DPOOL_DEFAULT flag tells Direct3D to create the vertex buffer in the memory allocation that is most appropriate for this buffer. The final parameter is the address of the vertex buffer to create.

After creating a vertex buffer, it is filled with data from the custom vertices as shown in the following code fragment.

```
VOID* pVertices;
if( FAILED( g_pVB->Lock( 0, sizeof(vertices), (void**)&pVertices, 0 ) ) )
    return E_FAIL;

memcpy( pVertices, vertices, sizeof(vertices) );

g_pVB->Unlock();
```

The vertex buffer is first locked by calling **IDirect3DVertexBuffer9::Lock**. The first parameter is the offset into the vertex data to lock, in bytes. The second parameter is the size of the vertex data to lock, in bytes. The third parameter is the address of a BYTE pointer, filled with a pointer to vertex data. The fourth parameter tells the vertex buffer how to lock the data.

The vertices are then copied into the vertex buffer using memcpy. After the vertices are in the vertex buffer, a call is made to **IDirect3DVertexBuffer9::Unlock** to unlock the vertex buffer. This mechanism of locking and unlocking is required because the vertex buffer may be in device memory.

Now that the vertex buffer is filled with the vertices, it is time to render the display, as described in Step 3 - Rendering the Display.

## Step 3 - Rendering the Display

Now that the vertex buffer is filled with vertices, it is time to render the display. Rendering the display starts by clearing the back buffer to a blue color and then calling BeginScene.

```
g_pd3dDevice->Clear( 0, NULL, D3DCLEAR_TARGET, D3DCOLOR_XRGB(0,0,255), 1.0f, 0L );
g_pd3dDevice->BeginScene();
```

Rendering vertex data from a vertex buffer requires a few steps. First, you need to set the stream source; in this case, use stream 0. The source of the stream is specified by calling **IDirect3DDevice9::SetStreamSource**.

```
g_pd3dDevice->SetStreamSource( 0, g_pVB, 0, sizeof(CUSTOMVERTEX) );
```

The first parameter is the stream number and the second parameter is a pointer to the vertex buffer. The third parameter is the offset from the beginning of the stream to the beginning of the vertex data, which is 0 in this example. The last parameter is the number of bytes of each element in the vertex declaration.

The next step is to call **IDirect3DDevice9::SetFVF** to identify the fixed function vertex shader. Full, custom vertex shaders are an advanced topic, but in this case the vertex shader is only the flexible vertex format (FVF) code. The following code fragment sets the FVF code.

```
g_pd3dDevice->SetFVF( D3DFVF_CUSTOMVERTEX );
```

The only parameter for SetFVF is the fixed vertex function code to define the vertex data layout.

The next step is to use **IDirect3DDevice9::DrawPrimitive** to render the vertices in the vertex buffer as shown in the following code fragment.

```
g_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLELIST, 0, 1 );
```

The first parameter accepted by DrawPrimitive is a flag that tells Direct3D what type of primitives to draw. This sample uses the flag D3DPT_TRIANGLELIST to specify a list of triangles. The second parameter is the index of the first vertex to load. The third parameter tells the number of primitives to draw. Because this sample draws only one triangle, this value is set to 1.

For more information about different kinds of primitives, see **Primitives**.

The last steps are to end the scene and then present the back buffer to the front buffer. This is shown in the following code fragment.

```
g_pd3dDevice->EndScene();
g_pd3dDevice->Present( NULL, NULL, NULL, NULL );
```

After the back buffer is presented to the front buffer, the client window shows a triangle with three different colored points.

This tutorial has shown you how to use vertices to render geometric shapes. Tutorial 3: Using Matrices introduces the concept of matrices and how to use them.