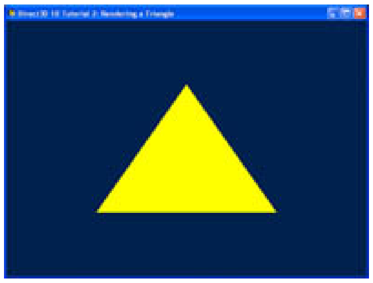
Tutorial 2: Rendering a Triangle

⊟ Collapse All



Summary

In the previous tutorial, we built a minimal Direct3D 10 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 setup the data structures associated with a triangle.

The outcome of this tutorial is a window with a triangle rendered to the center of it.

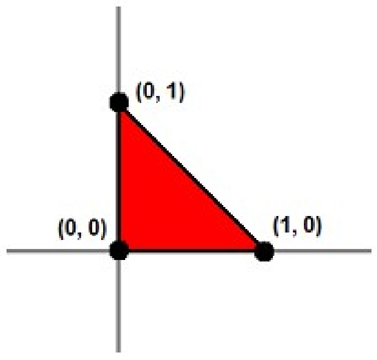
Source

(SDK root)\Samples\C++\Direct3D10\Tutorials\Tutorial02

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 10, 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 10, 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 D3DXVECTOR3. This type is a vector of 3 floating-points components, which is typically the data type used for position in 3D.

```
struct SimpleVertex
{
    D3DXVECTOR3 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 10, 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 D3D10_INPUT_ELEMENT_DESC structure. An application defines an array of one or more D3D10_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 D3D10_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 10 application passes vertex data to the GPU via the use of vertex buffer. In Direct3D 10, 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 D3D10_INPUT_PER_VERTEX_DATA. When an application uses instancing, it can set an input layout's InputSlotClass to D3D10_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 D3D10_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 D3D10_INPUT_ELEMENT_DESC array and create the input layout:

```
// Define the input layout
D3D10_INPUT_ELEMENT_DESC layout[] =
{
    { L"POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0, D3D10_INPUT_PER_VERTEX_DATA, 0 },
};
UINT numElements = sizeof(layout)/sizeof(layout[0]);
```

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 10 vertex layout object for the technique. However, we will learn that the technique and 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 first call the technique's GetPassByIndex() method to obtain an effect pass object that represents the first pass of the technique. Then, we call the pass object's GetDesc() method to obtain a pass description structure. Within this structure is a field named pIAInputSignature that points to the binary data that represents the input signature of the vertex shader used in this pass. Once we have this data, we can call

ID3D10Device::CreateInputLayout() to create a vertex layout object, and ID3D10Device::IASetInputLayout() to set it as the active vertex layout. The code to do all of that is shown below:

```
// Create the input layout
D3D10_PASS_DESC PassDesc;
g_pTechnique->GetPassByIndex( 0 )->GetDesc( &PassDesc );
if( FAILED( g_pd3dDevice->CreateInputLayout( layout, numElements, PassDesc.pIAInputSignature,
        PassDesc.IAInputSignatureSize, &g_pVertexLayout ) ) )
    return FALSE;
// Set the input layout
g_pd3dDevice->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 10, we fill in two structures, D3D10_BUFFER_DESC and D3D10_SUBRESOURCE_DATA, and then call ID3D10Device::CreateBuffer(). D3D10_BUFFER_DESC describes the vertex buffer object to be created, and D3D10_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 3 SimpleVertex 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 ID3D10Device::IASetVertexBuffers() to bind it to the device. The complete code is shown here:

```
// Create vertex buffer
SimpleVertex vertices[] =
{
    D3DXVECTOR3( 0.0f, 0.5f, 0.5f ),
    D3DXVECTOR3( 0.5f, -0.5f, 0.5f ),
    D3DXVECTOR3( -0.5f, -0.5f, 0.5f ),
};
D3D10_BUFFER_DESC bd;
bd.Usage = D3D10_USAGE_DEFAULT;
bd.ByteWidth = sizeof( SimpleVertex ) * 3;
bd.BindFlags = D3D10_BIND_VERTEX_BUFFER;
bd.CPUAccessFlags = 0;
bd.MiscFlags = 0;
D3D10_SUBRESOURCE_DATA InitData;
InitData.pSysMem = vertices;
if( FAILED( g_pd3dDevice->CreateBuffer( &bd, &InitData, &g_pVertexBuffer ) ) )
    return FALSE;

// Set vertex buffer
UINT stride = sizeof( SimpleVertex );
UINT offset = 0;
g_pd3dDevice->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 3 vertices define the first triangle and the second 3 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.

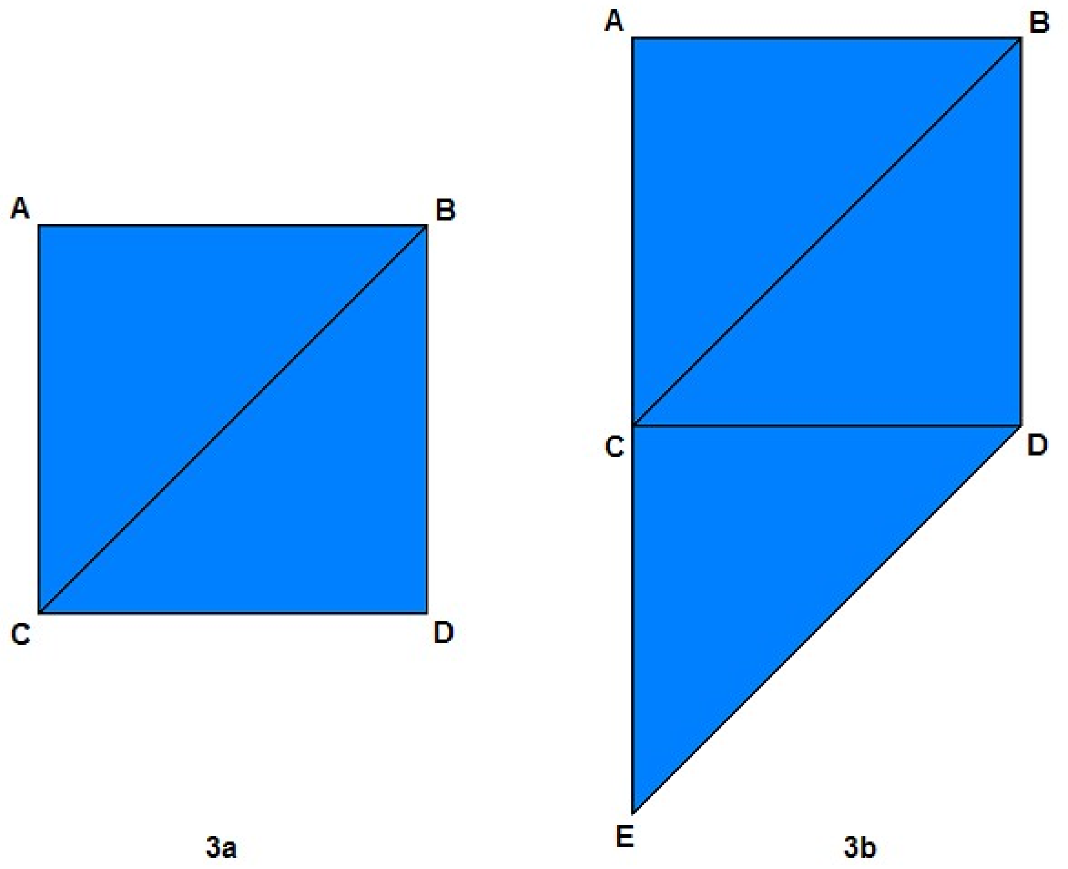


Figure 3a contains a square made up of two triangles; figure 3b contains a pentagonal shape made up of three triangles.

We can make the vertex buffer smaller if we can tell the GPU that when rendering the second triangle, instead of fetching all 3 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 3 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 3 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 10 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_pd3dDevice->IASetPrimitiveTopology( D3D10_PRIMITIVE_TOPOLOGY_TRIANGLELIST );
```

Rendering the Triangle

The final item missing is the code that does the actual rendering of the triangle. As mentioned earlier, this tutorial will use the effect system. We start by calling ID3D10EffectTechnique::GetDesc() on the technique object obtained earlier to receive a D3D10FX_TECHNIQUE_DESC structure which describes the technique. One of the members of D3D10FX_TECHNIQUE_DESC, Passes, indicates the number of passes the technique contains. To correctly render using this technique, the application should loop the same number of times as there are passes. Within the loop, we must first call the technique's GetPassByIndex() method to obtain the pass object, then call its Apply() method to have the effect system bind the associated shaders and render states to the graphics pipeline. The next thing that we do is call ID3D10Device::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
D3D10_TECHNIQUE_DESC techDesc;
g_pTechnique->GetDesc( &techDesc );
for( UINT p = 0; p < techDesc.Passes; ++p )
{
    g_pTechnique->GetPassByIndex( p )->Apply(0);
    g_pd3dDevice->Draw( 3, 0 );
}
```