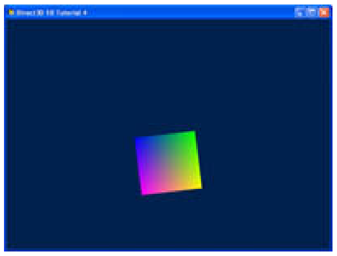
Tutorial 4: 3D Spaces

⊟ Collapse All



Summary

In the previous tutorial, we have 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.

Source

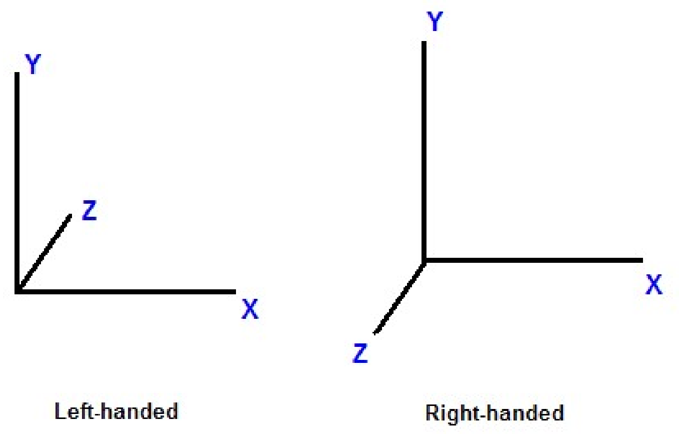
(SDK root)\Samples\C++\Direct3D10\Tutorials\Tutorial04

3D Spaces

In the previous tutorial, the vertices of the triangle were placed strategically to perfectly align itself on the screen. However, this will not always be the case, and thus, we need a system to denote objects in 3D space and a system to display them.

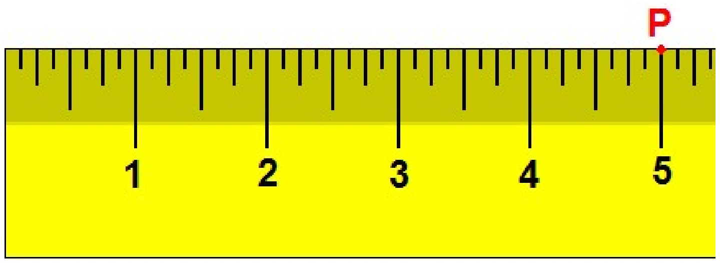
In the real world, objects exist in 3D space. This means that to place an object in a particular position in the world, we would need to use a coordinate system and define three coordinates that correspond to the position. In computer graphics, 3D spaces are most commonly in Cartesian coordinate system. 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



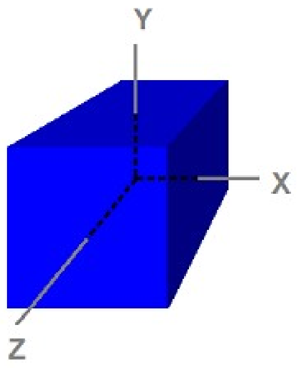
Now that we have talked about the coordinate system, consider 3D spaces. A point has different coordinates in different spaces. As an example in 1D, suppose we have a ruler and we note the point, P, at the 5-inch mark of the ruler. Now, if we move the ruler 1 inch to the right, the same point lies on the 4-inch mark. By moving the ruler, the frame of reference has changed, and 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 8 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.

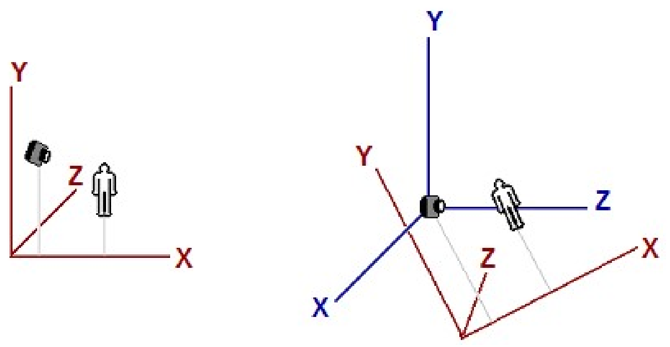
World Space

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 3 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

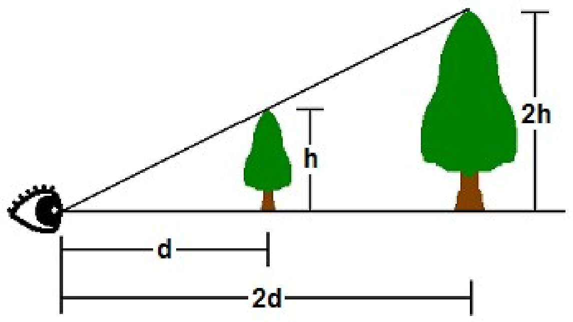
After vertices are converted to world space, view transformation converts those vertices from world space to view space. Recall from earlier discussion that view space is what the world appears from the viewer's (or camera's) perspective. In view space, the viewer is located at origin looking out along 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 Direct3D a convenient API call D3DXMatrixLookAtLH() 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

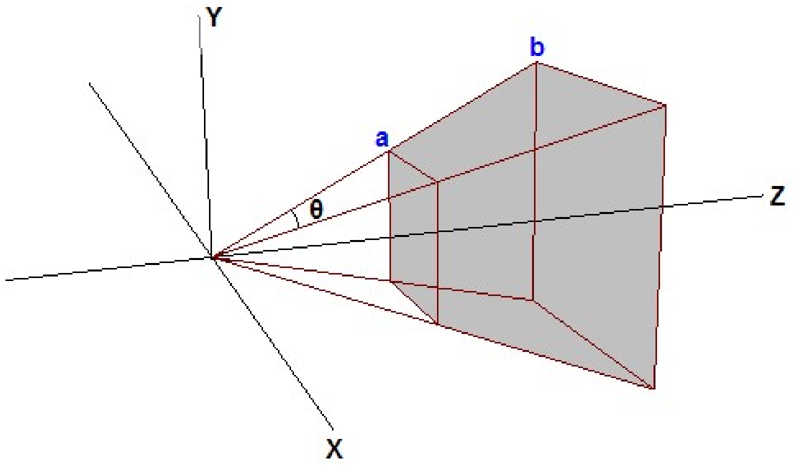


In 3D space, things appear in perspective. That is, the closer an object is, the larger it appears. As shown, the tip of a tree that is h units tall at d units away from the viewer's eye will appear at the same point as the tip of another tree 2h units tall and 2d units away. Therefore, where a vertex 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. Field-of-view (FOV) denotes which objects are visible from a particular position, while looking in a particular direction. Humans have a field-of-view 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 field-of-view 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 4 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 above, 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 10, the easiest way to obtain a projection matrix is to call the D3DXMatrixPerspectiveFovLH() 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 3 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[] =
{
    { D3DXVECTOR3( -1.0f,  1.0f, -1.0f ), D3DXVECTOR4( 0.0f, 0.0f, 1.0f, 1.0f ) },
    { D3DXVECTOR3(  1.0f,  1.0f, -1.0f ), D3DXVECTOR4( 0.0f, 1.0f, 0.0f, 1.0f ) },
    { D3DXVECTOR3(  1.0f,  1.0f,  1.0f ), D3DXVECTOR4( 0.0f, 1.0f, 1.0f, 1.0f ) },
    { D3DXVECTOR3( -1.0f,  1.0f,  1.0f ), D3DXVECTOR4( 1.0f, 0.0f, 0.0f, 1.0f ) },
    { D3DXVECTOR3( -1.0f, -1.0f, -1.0f ), D3DXVECTOR4( 1.0f, 0.0f, 1.0f, 1.0f ) },
    { D3DXVECTOR3(  1.0f, -1.0f, -1.0f ), D3DXVECTOR4( 1.0f, 1.0f, 0.0f, 1.0f ) },
    { D3DXVECTOR3(  1.0f, -1.0f,  1.0f ), D3DXVECTOR4( 1.0f, 1.0f, 1.0f, 1.0f ) },
    { D3DXVECTOR3( -1.0f, -1.0f,  1.0f ), D3DXVECTOR4( 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.

```
// Create index buffer
DWORD 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 twelve 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 D3D10_BIND_INDEX_BUFFER, and since we declared our array using DWORD, we will use sizeof(DWORD).

```
bd.Usage = D3D10_USAGE_DEFAULT;
bd.ByteWidth = sizeof( DWORD ) * 36;        // 36 vertices needed for 12 triangles in a triangle list
bd.BindFlags = D3D10_BIND_INDEX_BUFFER;
bd.CPUAccessFlags = 0;
bd.MiscFlags = 0;
InitData.pSysMem = indices;
if( FAILED( g_pd3dDevice->CreateBuffer( &bd, &InitData, &g_pIndexBuffer ) ) )
    return FALSE;
```

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_pd3dDevice->IASetIndexBuffer( g_pIndexBuffer, DXGI_FORMAT_R32_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 3 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 3 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 C++. The 3 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:

```
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 3 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 to define 3 ID3D10EffectMatrixVariable pointers. ID3D10EffectMatrixVariable is an object that represents a matrix in the constant buffer. We can use these 3 objects to copy the matrices to the constant buffer. Therefore, our global variables have the following addition:

```
ID3D10EffectMatrixVariable* g_pWorldVariable = NULL;
ID3D10EffectMatrixVariable* g_pViewVariable = NULL;
ID3D10EffectMatrixVariable* g_pProjectionVariable = NULL;
D3DXMATRIX                  g_World;
D3DXMATRIX                  g_View;
D3DXMATRIX                  g_Projection;
```

To initialize the 3 ID3D10EffectMatrixVariable objects, we call ID3D10Effect::GetVariableByName() after creating the effect, passing the name of the variable we are interested in. GetVariableByName() returns an ID3D10EffectVariable interface, even though the underlying object is specific to the variable type defined in the FX file. In this case, the underlying objects are ID3D10EffectMatrixVariable. To get an ID3D10EffectMatrixVariable interface from an ID3D10EffectVariable, we can call its AsMatrix () method:

```
g_pWorldVariable = g_pEffect->GetVariableByName( "World" )->AsMatrix();
g_pViewVariable = g_pEffect->GetVariableByName( "View" )->AsMatrix();
g_pProjectionVariable = g_pEffect->GetVariableByName( "Projection" )->AsMatrix();
```

The next thing that we need to do is coming up with 3 matrices which 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 D3DXMatrixLookAtLH() 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 D3DXMatrixPerspectiveFovLH(), 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 100, respectively. This means that anything closer than 0.1 or further than 100 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. Constant buffers are covered in depth in a later tutorial. For now, consider them the containers for constants passed to shaders. Because we are writing matrices to the constant buffer, we call ID3D10EffectMatrixVariable's SetMatrix() method, passing to it the matrix to be written to the buffer. This has to be done before we issue the Draw() call.

```
//
// Update variables
//
g_pWorldVariable->SetMatrix( (float*)&g_World );
g_pViewVariable->SetMatrix( (float*)&g_View );
g_pProjectionVariable->SetMatrix( (float*)&g_Projection );
```

© 2010 Microsoft Corporation. All rights reserved.
Send feedback to DxSdkDoc@microsoft.com.

Version: 1962.00