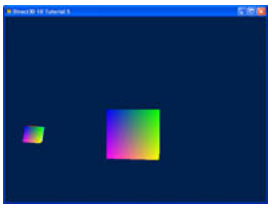
Tutorial 5: 3D Transformation

Collapse All



Summary

In the previous tutorial, we rendered a cube from model space to the screen. In this tutorial, we will extend the concept of transformations and demonstrate simple animation that can be achieved with these transformations.

The outcome of this tutorial will be an object that orbits around another. It would be useful to demonstrate the transformations and how they can be combined to achieve the desired effect. Future tutorials will be building on this foundation as we introduce new concepts.

Source

(SDK root)\Samples\C++\Direct3D10\Tutorials\Tutorial05

Transformation

In 3D graphics, transformation is often used to operate on vertices and vectors. It is also used to convert them in one space to another. Transformation is performed via multiplication with a matrix. There are typically three types of primitive transformation that can be performed on vertices: translation (where it lies in space relative to the origin), rotation (its direction in relation to the x, y, z frame), and scaling (its distance from origin). In addition to those, projection transformation is used to go from view space to projection space. The D3DX library contains APIs that can conveniently construct a matrix for many purposes such as translation, rotation, scaling, world-to-view transformation, view-to-projection transformation, etc. An application can then use these matrices to transform vertices in its scene. A basic understanding of matrix transformations is required. We will briefly look at some examples below.

Translation

Translation refers to moving or displacing for a certain distance in space. In 3D, the matrix used for translation has the form

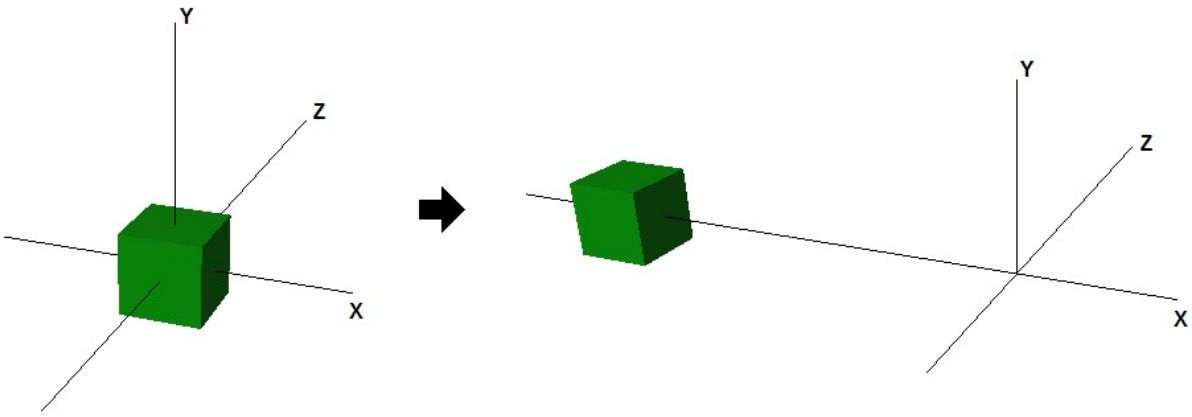
```
1  0  0  0
0  1  0  0
0  0  1  0
a  b  c  1
```

where (a, b, c) is the vector that defines the direction and distance to move. For example, to move a vertex -5 unit along the X axis (negative X direction), we can multiply it with this matrix:

```
 1  0  0  0
 0  1  0  0
 0  0  1  0
-5  0  0  1
```

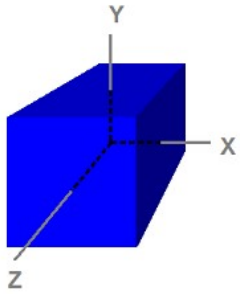
If we apply this to a cube object centered at origin, the result is that the box is moved 5 units towards the negative X axis, as figure 5 shows, after translation is applied.

Figure 1. The effect of translation



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 2. A cube defined in object space



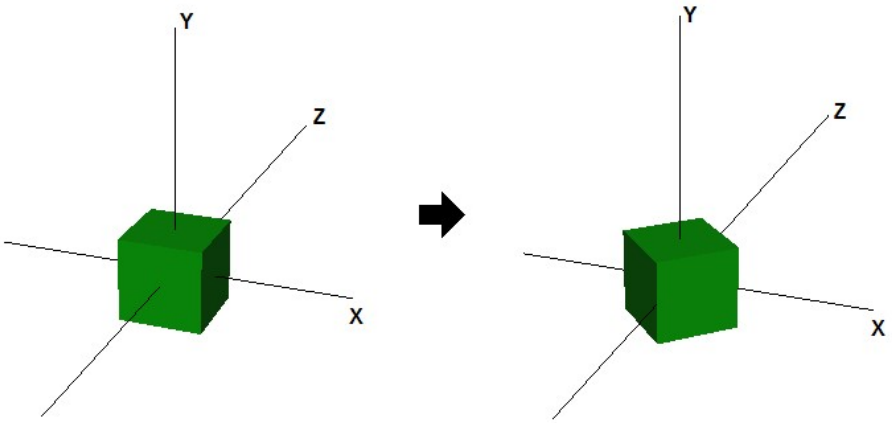
Rotation

Rotation refers to rotating vertices about an axis going through the origin. Three such axes are the X, Y, and Z axes in the space. An example in 2D would be rotating the vector [1 0] 90 degrees counter-clockwise. The result from the rotation is the vector [0 1]. The matrix used for rotating î degrees clockwise about the Y axis looks like this:

```
cosî   0  -sinî   0
 0     1     0    0
sinî   0   cosî   0
 0     0     0    1
```

Figure 6 shows the effect of rotating a cube centered at origin for 45 degrees about the Y axis.

Figure 3. The effect of rotation about the Y axis



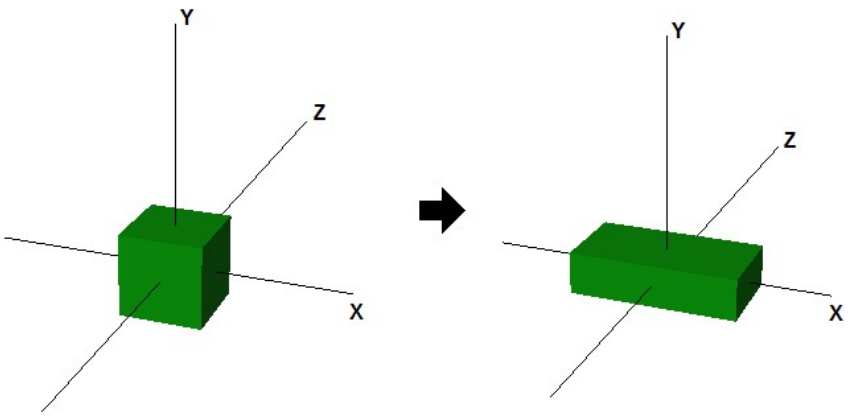
Scaling

Scaling refers to enlarging or shrinking the size of vector components along axis directions. For example, a vector can be scaled up along all directions or scaled down along the X axis only. To scale, we usually apply the scaling matrix below:

```
p  0  0  0
0  q  0  0
0  0  r  0
0  0  0  1
```

where p, q, and r are the scaling factor along the X, Y, and Z direction, respectively. The figure below shows the effect of scaling by 2 along the X axis and scaling by 0.5 along the Y axis.

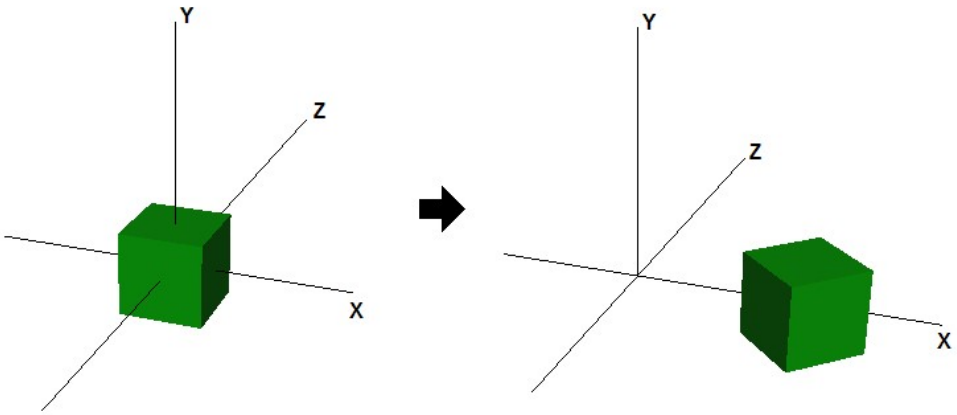
Figure 4. The effect of Scaling



Multiple Transformations

To apply multiple transformations to a vector, we can simply multiply the vector by the first transformation matrix, then multiply the resulting vector by the second transformation matrix, and so on. Because vector and matrix multiplication is associative, we can also multiply all of the matrices first, then multiply the vector by the product matrix and obtain an identical result. The figure below shows how the cube would end up if we combine a rotation and a translation transformation together.

Figure 5. The effect of rotation and translation



Creating the Orbit

In this tutorial, we will be transforming two cubes. The first one will rotate in place, while the second one will rotate around the first, while spinning on its own axis. The two cubes will have its own world transformation matrix associated with it, and this matrix will be reapplied to it in every frame rendered.

There are functions within D3DX that will assist in the creation of the rotation, translation, and scaling matrices.

- Rotations performed around the X, Y and Z axis are accomplished with the functions D3DXMatrixRotationX, D3DXMatrixRotationY, and D3DXMatrixRotationZ respectively. They create basic rotation matrices which rotate around one of the primary axis. Complex rotations around other axis can be done by multiplying together several of them.
- Translations can be performed by invoking the D3DXMatrixTranslation function. This function will create a matrix that will translate points specified by the parameters.
- Scaling is done with D3DXMatrixScaling. It scales only along the primary axes. If scaling along arbitrary axis is desired, then the scaling matrix can be multiplied with an appropriate rotation matrix to achieve the effect.

The first cube will be spinning in place and act as the center for the orbit. The cube has a rotation along the Y axis applied to the associated world matrix. This is done by calling the D3DXMatrixRotationY function shown below. The cube is rotated by a set amount each frame. Since the cubes are suppose to continuously rotate, the value which the rotation matrix is based on gets incremented with every frame.

```
// 1st Cube: Rotate around the origin
D3DXMatrixRotationY( &g_World1, t );
```

The second cube will be orbiting around the first one. To demonstrate multiple transformations, a scaling factor, and its own axis spin will be added. The formula used is shown right below the code (in comments). First the cube will be scale down to 30% size, and then it will be rotated along its spin axis (the Z axis in this case). To simulate the orbit, it will get translated away from the origin, and then rotated along the Y axis. The desired effect can be achieved by utilizing four separate matrices with its individual transformation (mScale, mSpin, mTranslate, mOrbit), then multiplied together.

```
// 2nd Cube:  Rotate around origin
D3DXMATRIX mTranslate;
D3DXMATRIX mOrbit;
D3DXMATRIX mSpin;
D3DXMATRIX mScale;
D3DXMatrixRotationZ( &mSpin, -t );
D3DXMatrixRotationY( &mOrbit, -t*2.0f );
D3DXMatrixTranslation( &mTranslate, -4.0f, 0.0f, 0.0f );
D3DXMatrixScaling( &mScale, 0.3f, 0.3f, 0.3f );

D3DXMatrixMultiply( &g_World2, &mScale, &mSpin );
D3DXMatrixMultiply( &g_World2, &g_World2, &mTranslate );
D3DXMatrixMultiply( &g_World2, &g_World2, &mOrbit );
//g_World2 = mScale * mSpin * mTranslate * mOrbit;
```

An important point to note is that these operations are not commutative. The order in which the transformations are applied matter. Experiment with the order of transformation and observe the results.

Since all the transformation functions will create a new matrix from the parameters, the amount at which they rotate has to be incremented. This is done by updating the "time" variable.

```
// Update our time
t += D3DX_PI * 0.0125f;
```

Before the rendering calls are made, the technique must collect the variables for the shaders. Here, the world, view, and projection matrices are bound to the technique. Note that the world matrix is unique to each cube, and thus, changes for every object that gets passed into it.

```
//
// Render the first cube
//
D3D10_TECHNIQUE_DESC techDesc;
g_pTechnique->GetDesc( &techDesc );
for( UINT p = 0; p < techDesc.Passes; ++p )
{
    g_pTechnique->GetPassByIndex( p )->Apply(0);
    g_pd3dDevice->DrawIndexed( 36, 0, 0 );
}

//
// Update variables for the second cube
//
g_pWorldVariable->SetMatrix( (float*)&g_World2 );
```

```
g_pViewVariable->SetMatrix( (float*)&g_View );
g_pProjectionVariable->SetMatrix( (float*)&g_Projection );

//
// Render the second cube
//
for( UINT p = 0; p < techDesc.Passes; ++p )
{
    g_pTechnique->GetPassByIndex( p )->Apply(0);
    g_pd3dDevice->DrawIndexed( 36, 0, 0 );
}
```

The Depth Buffer

There is one other important addition to this tutorial, and that is the depth buffer. Without it, the smaller orbiting cube would still be drawn on top of the larger center cube when it went around the back of the latter. The depth buffer allows Direct3D to keep track of the depth of every pixel drawn to the screen. The default behavior of the depth buffer in Direct3D 10 is to check every pixel being drawn to the screen against the value stored in the depth buffer for that screen-space pixel. If the depth of the pixel being rendered is less than or equal to the value already in the depth buffer, the pixel is drawn and the value in the depth buffer is updated to the depth of the newly drawn pixel. On the other hand, if the pixel being draw has a depth greater than the value already in the depth buffer, the pixel is discarded and the depth value in the depth buffer remains unchanged.

The following code in the sample creates a depth buffer (a DepthStencil texture). It also creates a DepthStencilView of the depth buffer so that Direct3D 10 knows to use it as a Depth Stencil texture.

```
// Create depth stencil texture
D3D10_TEXTURE2D_DESC descDepth;
descDepth.Width = width;
descDepth.Height = height;
descDepth.MipLevels = 1;
descDepth.ArraySize = 1;
descDepth.Format = DXGI_FORMAT_D32_FLOAT;
descDepth.SampleDesc.Count = 1;
descDepth.SampleDesc.Quality = 0;
descDepth.Usage = D3D10_USAGE_DEFAULT;
descDepth.BindFlags = D3D10_BIND_DEPTH_STENCIL;
descDepth.CPUAccessFlags = 0;
descDepth.MiscFlags = 0;
hr = g_pd3dDevice->CreateTexture2D( &descDepth, NULL, &g_pDepthStencil );
if( FAILED(hr) )
    return hr;

// Create the depth stencil view
D3D10_DEPTH_STENCIL_VIEW_DESC descDSV;
descDSV.Format = descDepth.Format;
descDSV.ViewDimension = D3D10_DSV_DIMENSION_TEXTURE2D;
descDSV.Texture2D.MipSlice = 0;
hr = g_pd3dDevice->CreateDepthStencilView( g_pDepthStencil, &descDSV, &g_pDepthStencilView );
if( FAILED(hr) )
    return hr;
```

In order to use this newly created depth stencil buffer, the tutorial must bind it to the device. This is done by passing the depth stencil view to the third parameter of the OMSetRenderTargets function.

```
g_pd3dDevice->OMSetRenderTargets( 1, &g_pRenderTargetView, g_pDepthStencilView );
```

As with the render target, we must also clear the depth buffer before rendering. This ensures that depth values from previous frames do not incorrectly discard pixels in the current frame. In the code below the tutorial is actually setting the depth buffer to be the maximum amount (1.0).

```
//
// Clear the depth buffer to 1.0 (max depth)
//
g_pd3dDevice->ClearDepthStencilView( g_pDepthStencilView, D3D10_CLEAR_DEPTH, 1.0f, 0 );
```