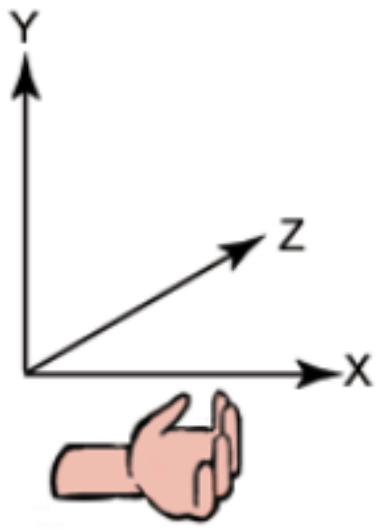
DirectX Programming

3D Spaces and Transformation

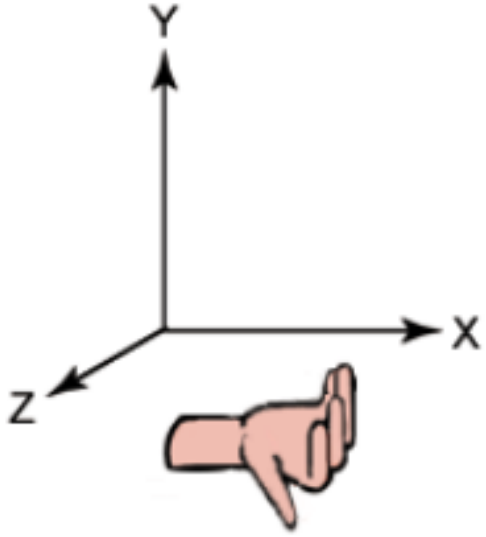
Computer Graphics, 2012 Spring
Jihye Yun

Coordinate System

- D3D uses a left-handed coordinate system
- Right-handed coordinate is also available
- But take care of backface-culling order



Left-handed

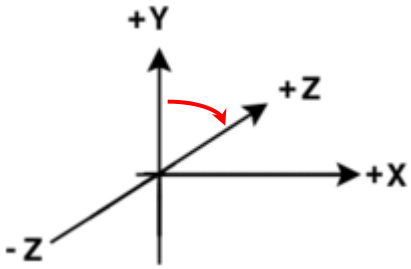


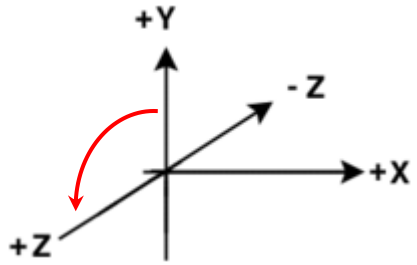
Right-handed

Coordinate System

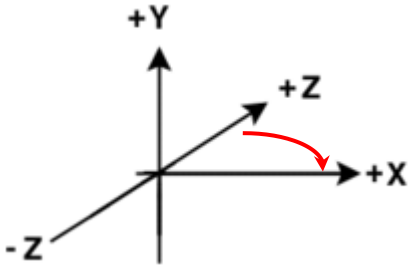
Left-handed vs. Right-handed

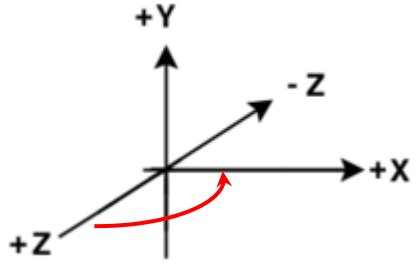
Positive x rotation



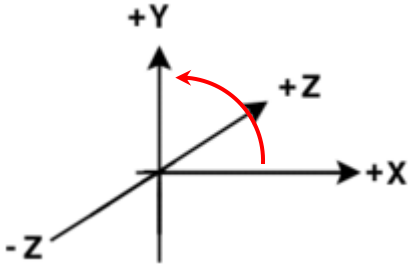


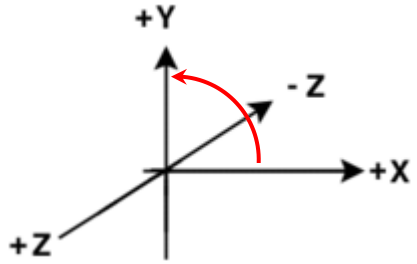
Positive y rotation





Positive z rotation

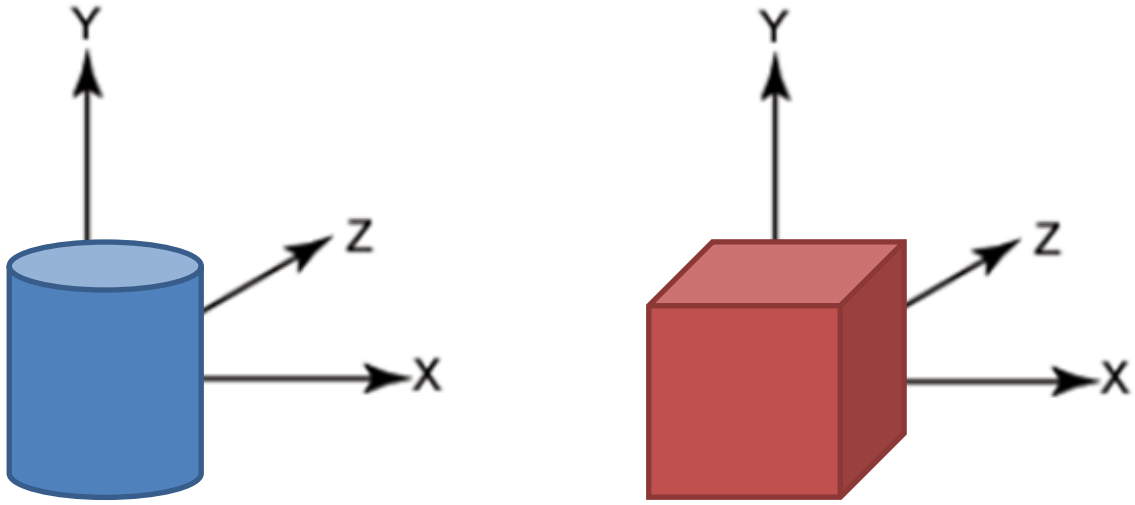




Object Space

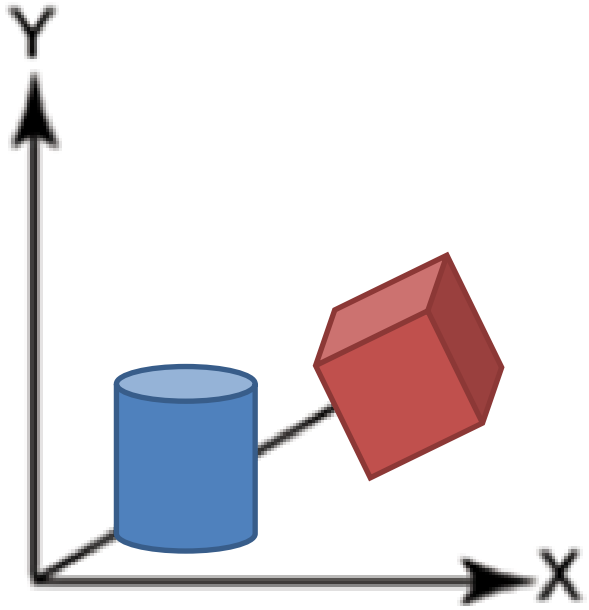
- Also called *model space*
- Usually, design and create models centered around the origin

➔ Easier to perform transformation



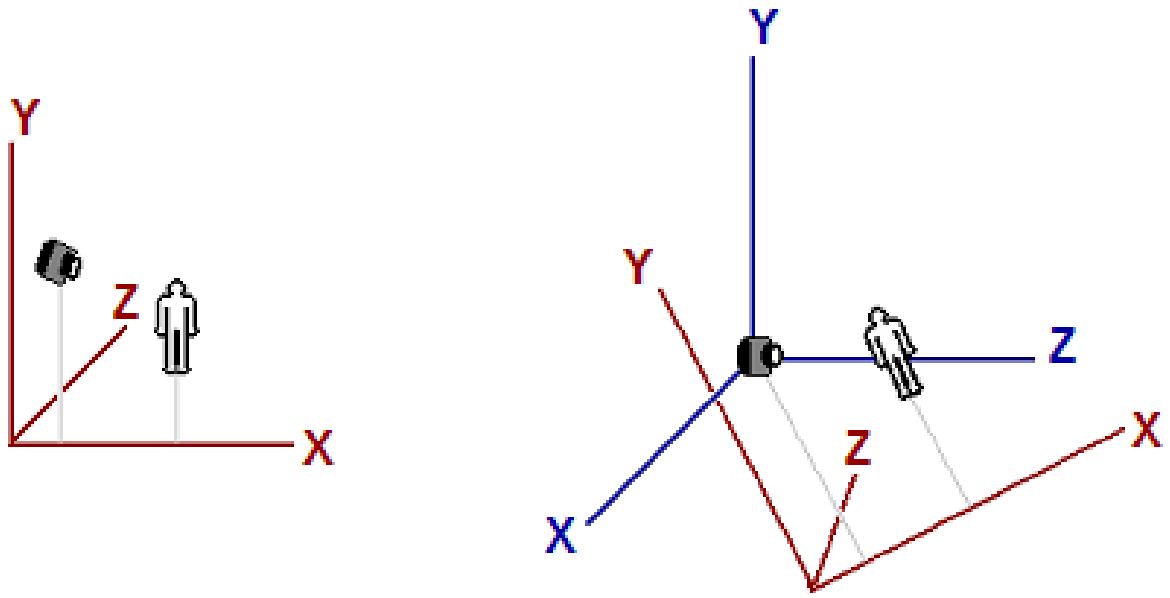
World Space

- Space shared by every object in the scene
- Define spatial relationship between objects that we wish to render



View Space

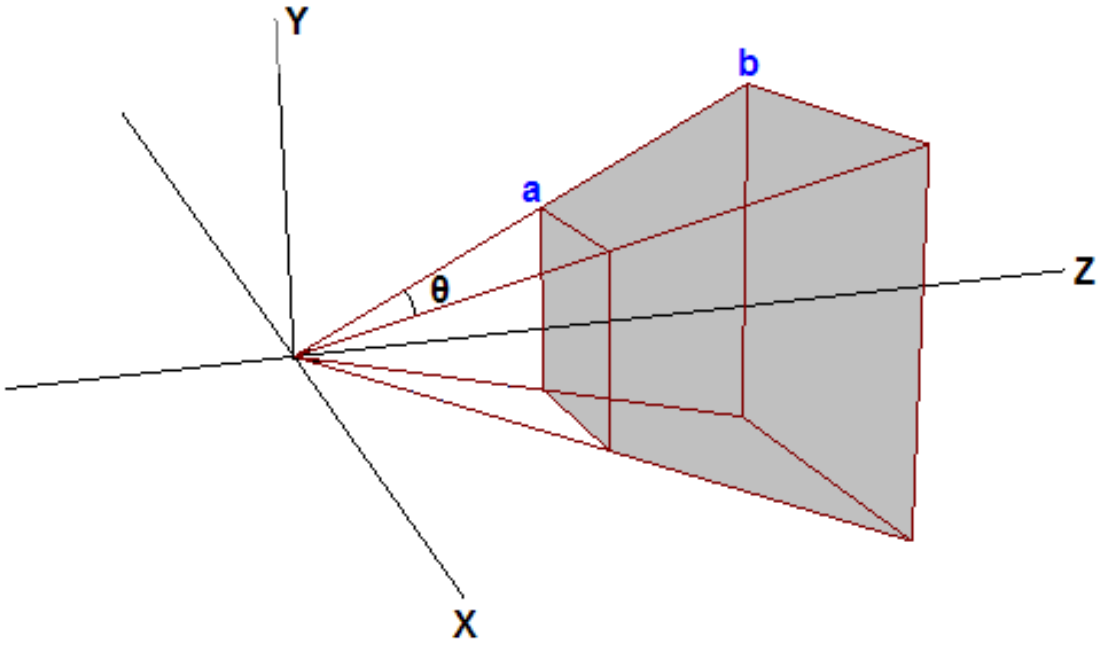
- Sometimes called *camera space*
- Origin is at the viewer or camera



The same object in world space and in view space

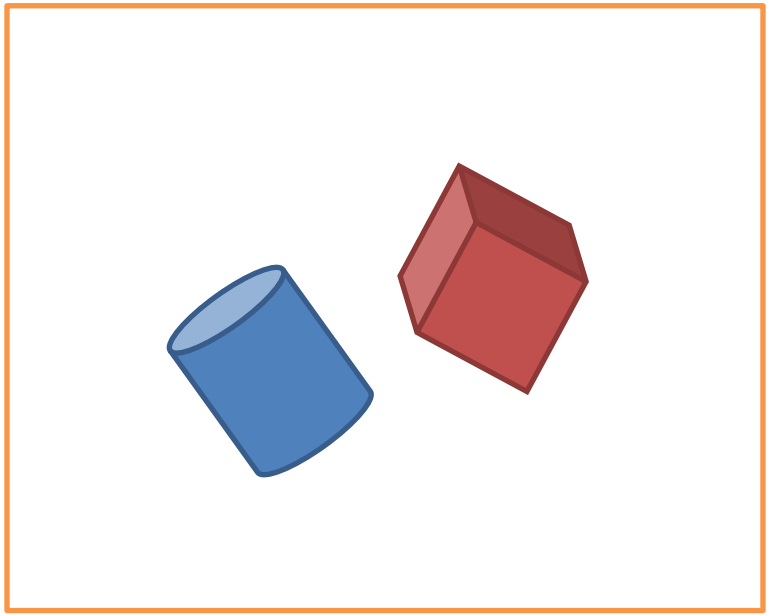
Projection Space

- Define the area in the 3D scene that objects are visible from the camera's point of view

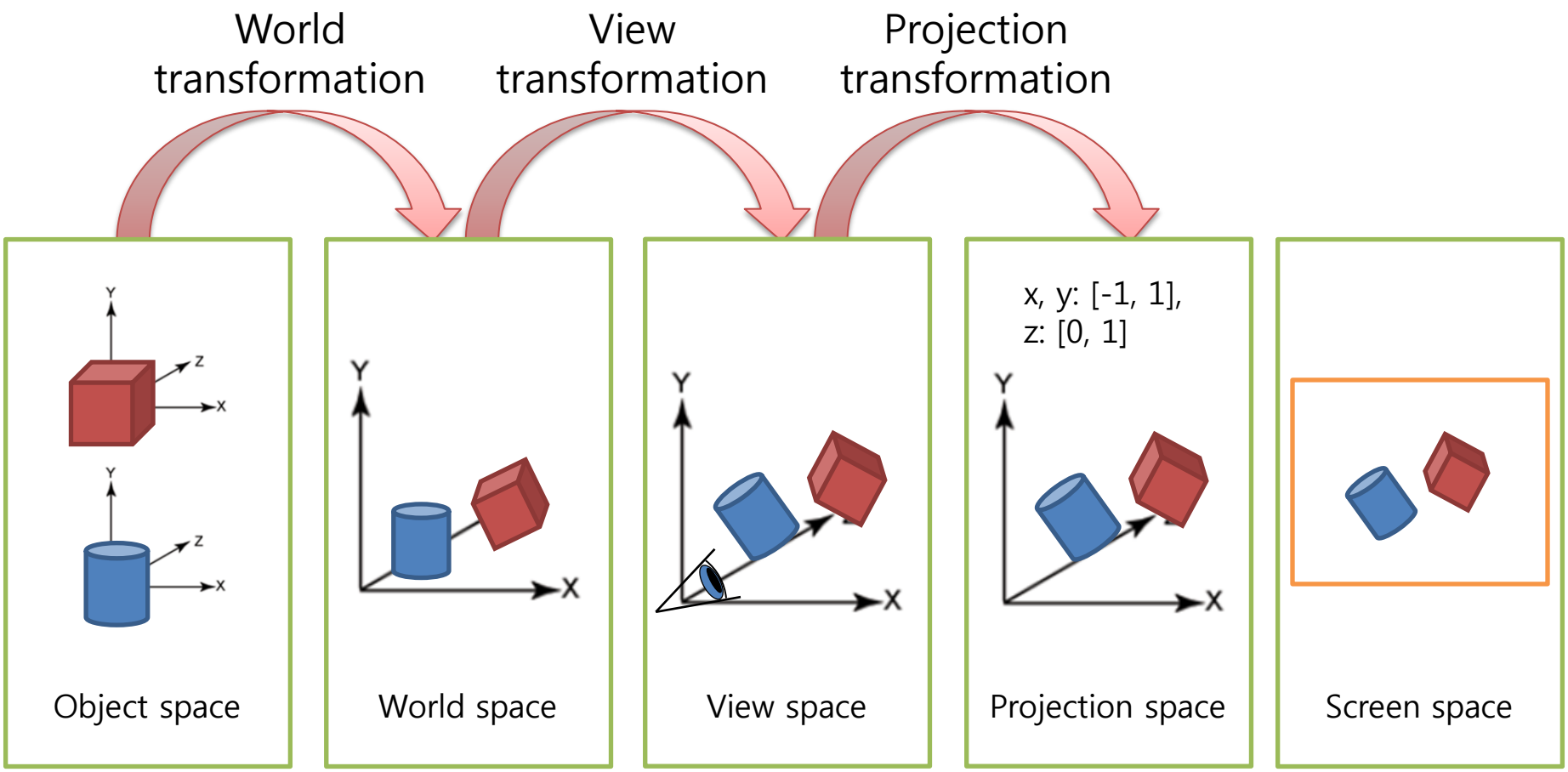


Screen Space

- Used to refer to locations in the frame buffer (2D space)



Space-to-Space Transformation



Space-to-Space Transformation

- View transformation
  - **XMMatrixLookAtLH**(XMVECTOR eyePosition,
    XMVECTOR focusPosition,
    XMVECTOR upDirection)
  - XMMatrixLookAtRH
  - XMMatrixLookToLH
  - XMMatrixLookToRH

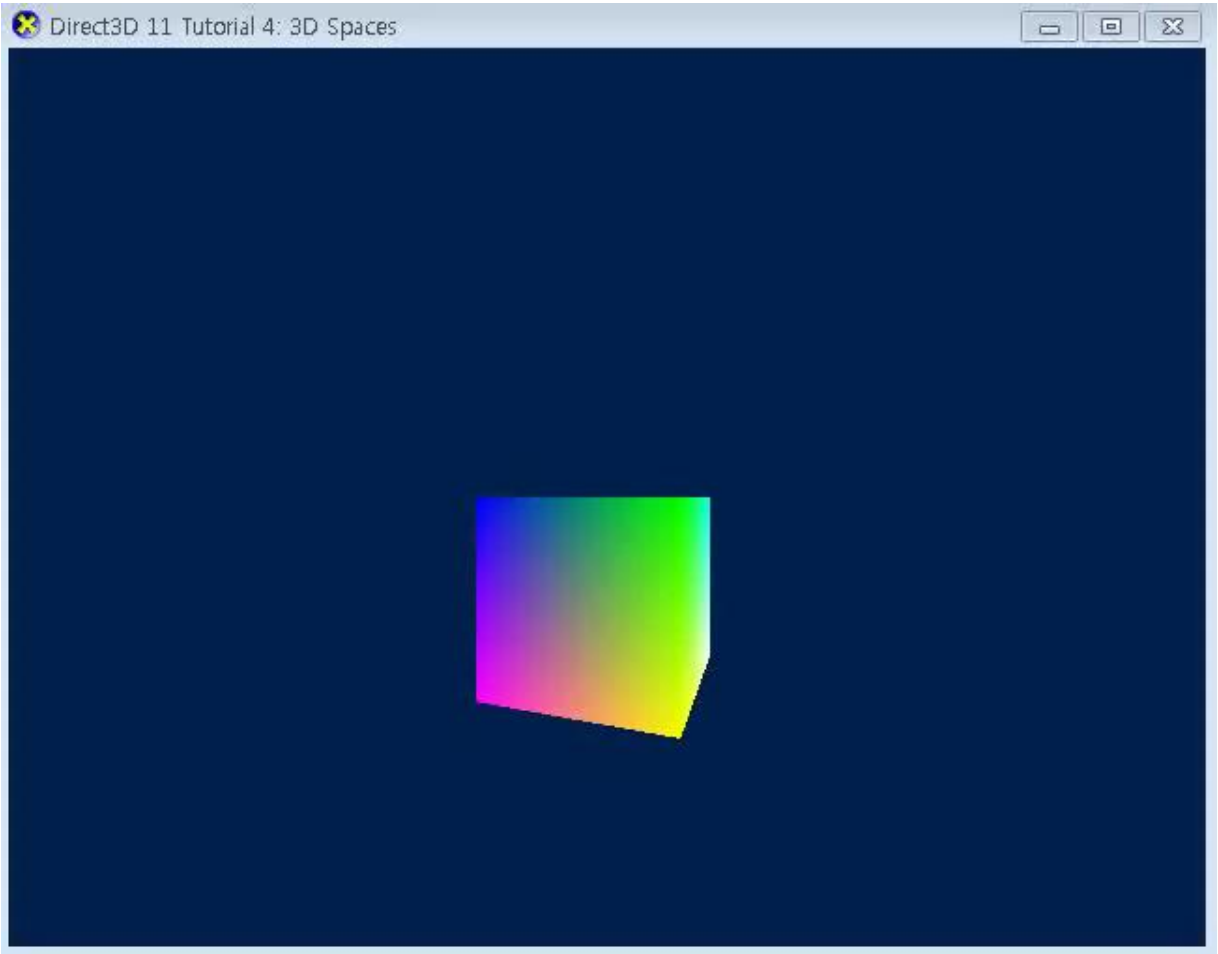
Space-to-Space Transformation

- Projection transformation
  - **XMMatrixPerspectiveFovLH**(float fovAngleY,
    float aspectRatio,
    float nearZ,
    float farZ)
  - XMMatrixPerspectiveFovRH
  - XMMatrixPerspectiveLH
  - XMMatrixPerspectiveRH
  - XMMatrixPerspectiveOffCenterLH
  - XMMatrixPerspectiveOffCenterRH

Space-to-Space Transformation

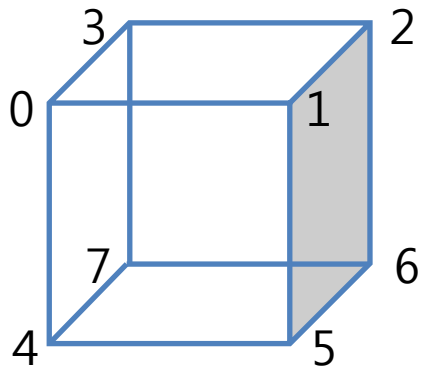
- Projection transformation
  - **XMMatrixOrthographicLH**(float viewWidth,
    float viewHeight,
    float nearZ,
    float farZ)
  - XMMatrixOrthographicRH
  - XMMatrixOrthographicOffCenterLH
  - XMMatrixOrthographicOffCenterRHs

Tutorial 4: 3D Spaces



Modify Vertex Buffer

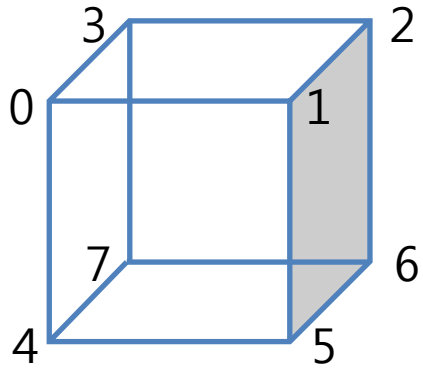
```
SimpleVertex vertices[ ] = {
        { XMFLOAT3(-1.0f, 1.0f, -1.0f), XMFLOAT4(0.0f, 0.0f, 1.0f, 1.0f) },
        { XMFLOAT3( 1.0f, 1.0f, -1.0f), XMFLOAT4(0.0f, 1.0f, 0.0f, 1.0f) },
        { XMFLOAT3( 1.0f, 1.0f, 1.0f), XMFLOAT4(0.0f, 1.0f, 1.0f, 1.0f) },
        { XMFLOAT3(-1.0f, 1.0f, 1.0f), XMFLOAT4(1.0f, 0.0f, 0.0f, 1.0f) },
        { XMFLOAT3(-1.0f, -1.0f, -1.0f), XMFLOAT4(1.0f, 0.0f, 1.0f, 1.0f) },
        { XMFLOAT3( 1.0f, -1.0f, -1.0f), XMFLOAT4(1.0f, 1.0f, 0.0f, 1.0f) },
        { XMFLOAT3( 1.0f, -1.0f, 1.0f), XMFLOAT4(1.0f, 1.0f, 1.0f, 1.0f) },
        { XMFLOAT3(-1.0f, -1.0f, 1.0f), XMFLOAT4(0.0f, 0.0f, 0.0f, 1.0f) },
};
```



Create Index Buffer

- Specify which points to use in each triangle

```
// create index buffer
WORD 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,
};
```



Create Index Buffer

```
D3D11_BUFFER_DESC bd;
Zeromemory( &bd, sizeof(bd) );
bd.Usage = D3D11_USAGE_DEFAULT;
bd.ByteWidth = sizeof(WORD) * 36;
bd.BindFlags = D3D11_BIND_INDEX_BUFFER;
bd.CPUaccessFlags = 0;
bd.MiscFlags = 0;
InitData.pSysMem = indices;

if( FAILED( g_pd3dDevice->CreateBuffer( &bd, &InitData,
                                        &g_pIndexBuffer ) ) )
    return FALSE;

g_pImmediateContext->IASetIndexBuffer( g_pIndexBuffer,
                                       DXGI_FORMAT_R16_UINT, 0);
```

D3D11_USAGE

- Directly reflect whether a resource is accessible by the CPU and/or the GPU

D3D11_USAGE_DEFAULT	A resource that requires read and write access by the GPU
D3D11_USAGE_IMMUTABLE	A resource that can only be read by the GPU
D3D11_USAGE_DYNAMIC	A resource that is accessible by GPU (read only) and the CPU (write only)
D3D11_USAGE_STAGING	A resource that supports data transfer (copy) from the GPU to the CPU

D3D11_USAGE

- Directly reflect whether a resource is accessible by the CPU and/or the GPU

Usage	Default	Dynamic	Immutable	Staging
GPU-Read	o	o	o	o
GPU-Write	o			o
CPU-Read				o
GPU-Write		o		o

Modify Vertex Shader

- Transform the input vertex positions
  - World, view, projection transformation
  - Use the constant buffer

```
cbuffer ConstantBuffer : register ( b0 )
{
        matrix World;
        matrix View;
        matrix Projection;
}
```

Modify 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;
}
```

Set up Matrices

- When initiating the rendering by calling Draw(), vertex shader reads the matrices stored in constant buffer

- Before rendering, copy the values of transformation matrices to the shader constant buffer

Set up Matrices

- Copy matrices to the shader constant buffer

```
ID3D11Buffer* g_pConstantBuffer = NULL;
XMMATRIX g_World, g_View, g_Projection;

D3D11_BUFFER_DESC bd;
ZeroMemory( &bd, sizeof(bd) );
bd.Usage = D3D11_USAGE_DEFAULT;
bd.ByteWidth = sizeof( ConstantBuffer );
bd.BindFlags = D3D11_BIND_CONSTANT_BUFFER;
bd.CPUAccessFlags = 0;
if( FAILED( g_pd3dDevice->CreateBuffer( &bd, NULL,
                                        &g_pConstantBuffer) ) )
    return FALSE;
```

Set up Matrices

```
 // Initialize the world matrix
g_World = XMMatrixIdentity();

// Initialize the view matrix
XMVECTOR Eye = XMVectorSet( 0.0f, 1.0f, -5.0f, 0.0f );
XMVECTOR At = XMVectorSet( 0.0f, 1.0f, 0.0f, 0.0f );
XMVECTOR Up = XMVectorSet( 0.0f, 1.0f, 0.0f, 0.0f );
g_View = XMMatrixLookAtLH( Eye, At, Up );

// Initialize the projection matrix
g_Projection = XMMatrixPerspectiveFovLH( XM_PIDIV2,
                                         width / (FLOAT)height, 0.01f, 100.0f );
```

Update Constant Buffers

- Because matrices are arranged differently in memory in C++ and HLSL, must transpose the matrices before updating them

```
ConstantBuffer cb;
cb.mWorld = XMMatrixTranspose( g_World );
cb.mView = XMMatrixTranspose( g_View );
cb.mProjection = XMMatrixTranspose( g_Projection );
g_pImmediateContext->UpdateSubresource( g_pConstantBuffer, 0,
                                        NULL, &cb, 0, 0);
```

Resource

- Resource types

ID3D11Buffer	Access buffer data (vertex, index, constant buffer)
ID3D11Texture1D	Access data in a 1D texture or 1D texture array
ID3D11Texture2D	Access data in a 2D texture or 2D texture array
ID3D11Texture3D	Access data in a 3D texture or 3D texture array

DirectXMath Library

- Optimal and portable interface for arithmetic and linear algebra on single-precision floating-point vectors or matrices
  - Structures
    - D3DXCOLOR, D3DXVECTOR3, D3DXMATRIX, D3DXQUATERNION,…
  - Functions
    - D3DXVec3Normalize, D3DXVec3Cross, D3DXMatrixTranspose, D3DXMatrixLookAtLH, D3DXMatrixRotationQuaternion, …

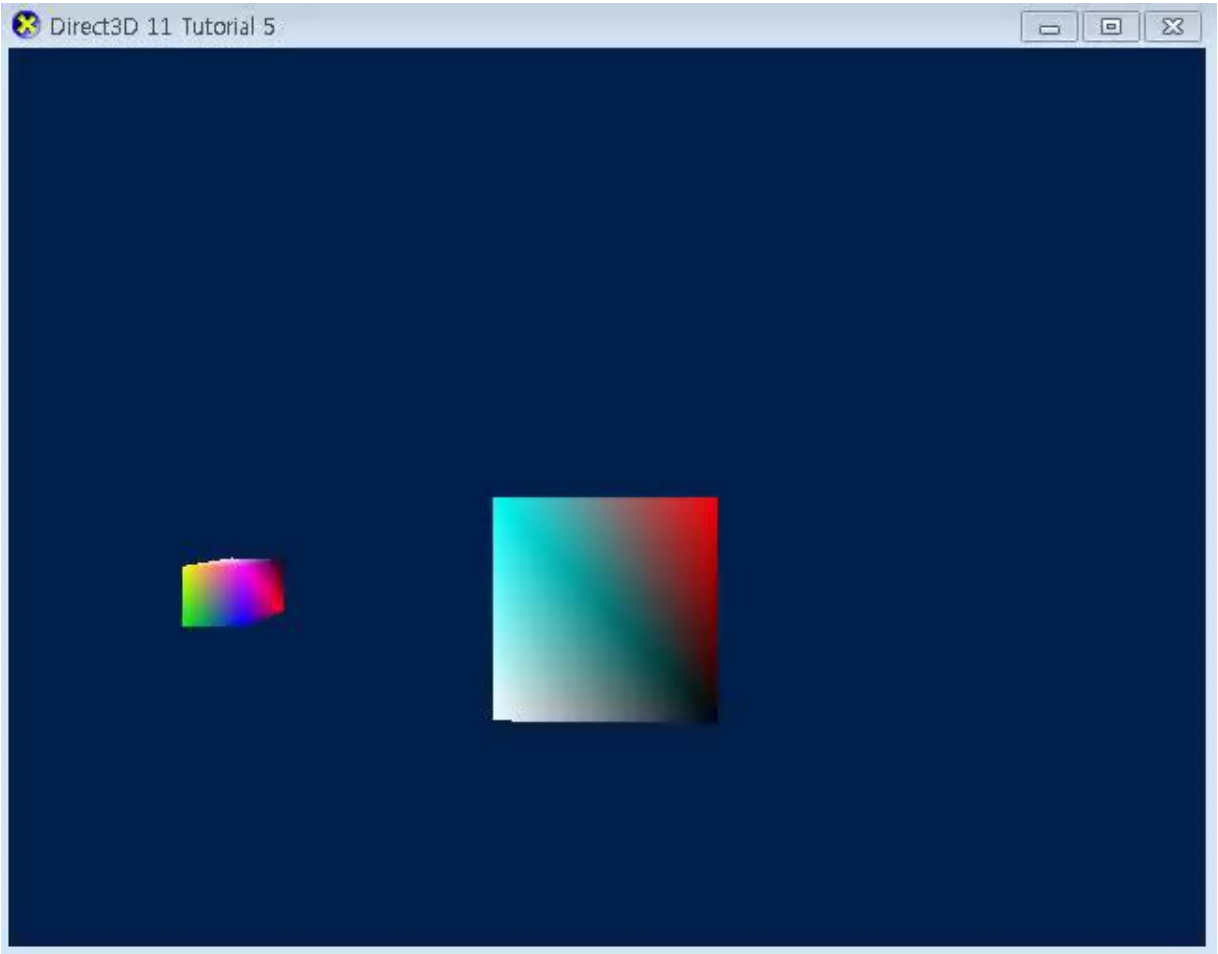
Declared in <D3DX10Math.h>

XNA Math Library

- New C++ math library for product distributed with DirectX SDK, intended to replace math support from D3DX
  - Structures
    - XMVECTOR, XMMATRIX, …
  - Functions
    - XMVector3Normalize, XMVector3Cross XMMatrixTranspose, XMMatrixLookAtLH, XMMatrixRotationQuaternion, …

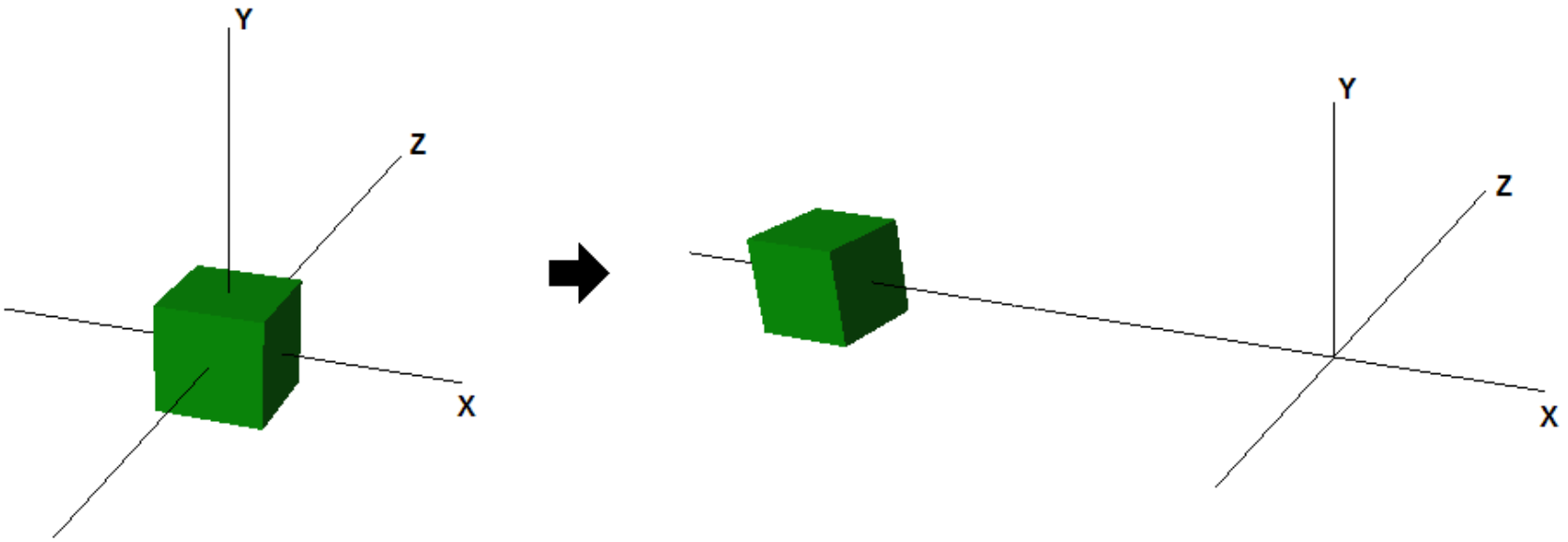
#include <xnamath.h>

Tutorial 5: 3D Transformation



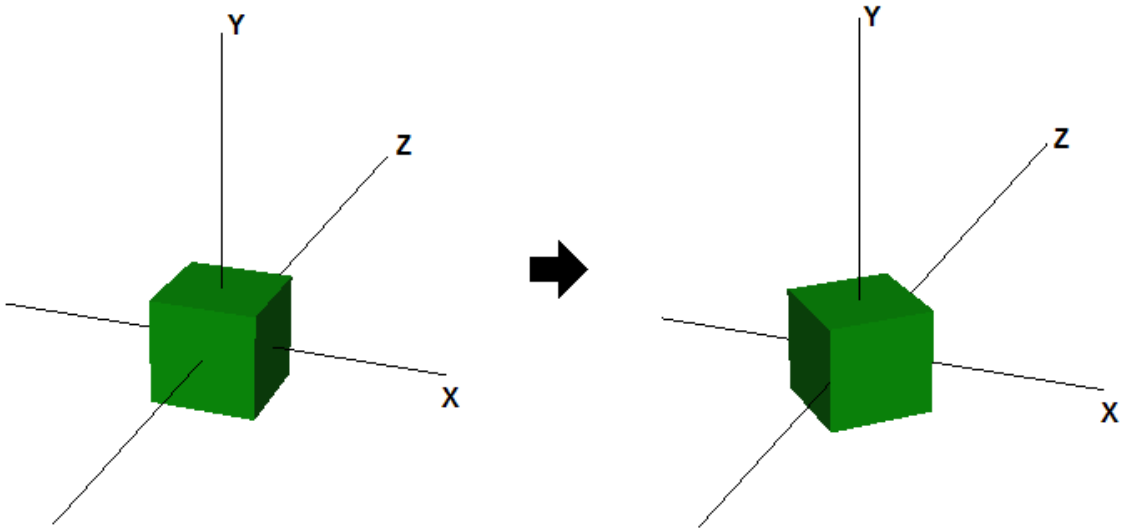
Translation

- Move or displace for a certain distance in space
  - **XMMatrixTranslation**(float x, float y, float z)



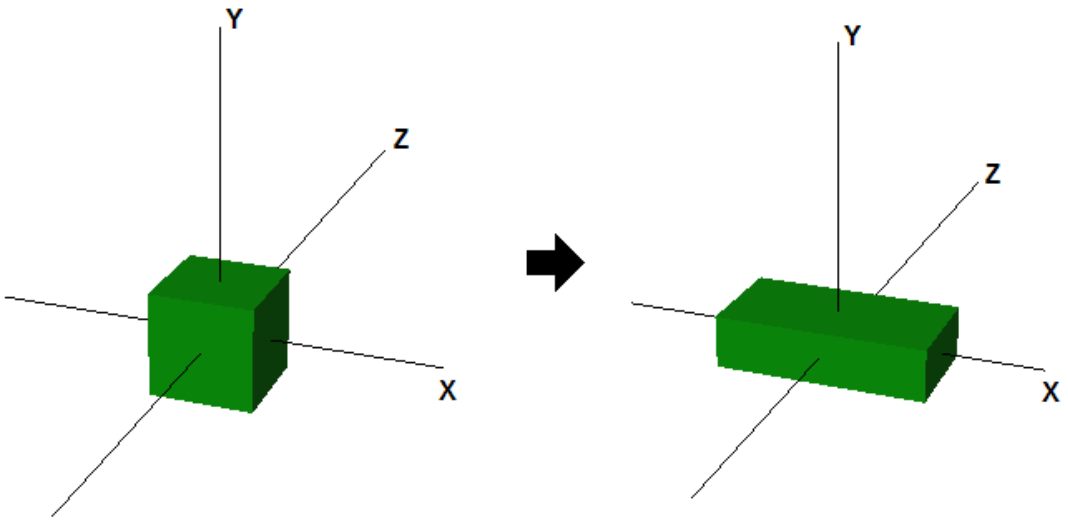
Rotation

- Rotate vertices about an axis going through the origin
  - **XMMatrixRotationX**(float angle)
  - **XMMatrixRotationY**(float angle)
  - **XMMatrixRotationZ**(float angle)



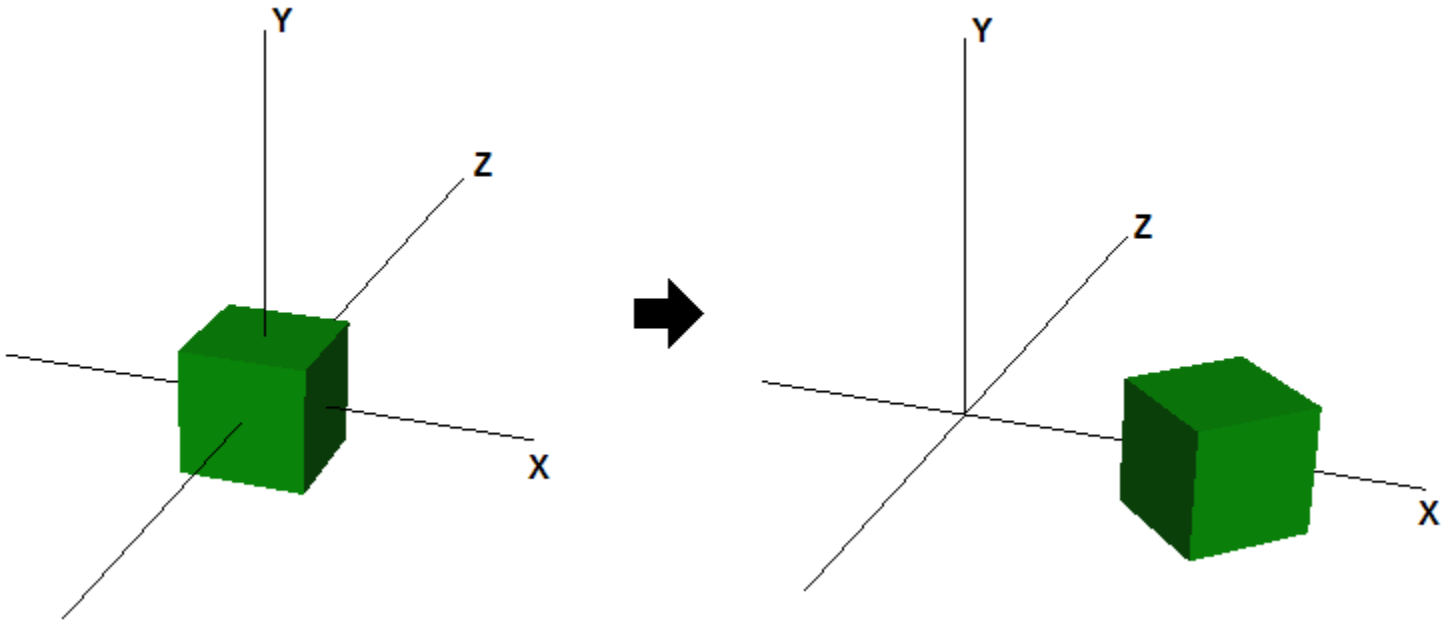
Scaling

- Enlarge or shrink the size of vector components along axis directions
  - **XMMatrixScaling**(float scaleX,
    float scaleY,
    float scaleZ)



Multiple Transformations

- Multiply all of the matrices first, then multiply the vector by the matrix



Create Orbit

```
// update our time
t += XM_PI * 0.0125f;

// 1st cube:
//spinning in place and act as the center of the orbit
g_World = XMMatrixRotationY( t );

// 2nd cube:
// orbiting around the 1st one while spinning on its own axis
XMMATRIX mScale = XMMatrixScaling( 0.3f, 0.3f, 0.3f );
XMMATRIX mSpin= XMMatrixRoationZ( -t );
XMMATRIX mTranslate = XMMatrixTransation( -4.0f, 0.0f, 0.0f );
XMMATRIX mOrbit = XMMatrixRotationY( -t * 2.0f );
g_World2 = mScale * mSpin * mTranslate * mOrbit;
```

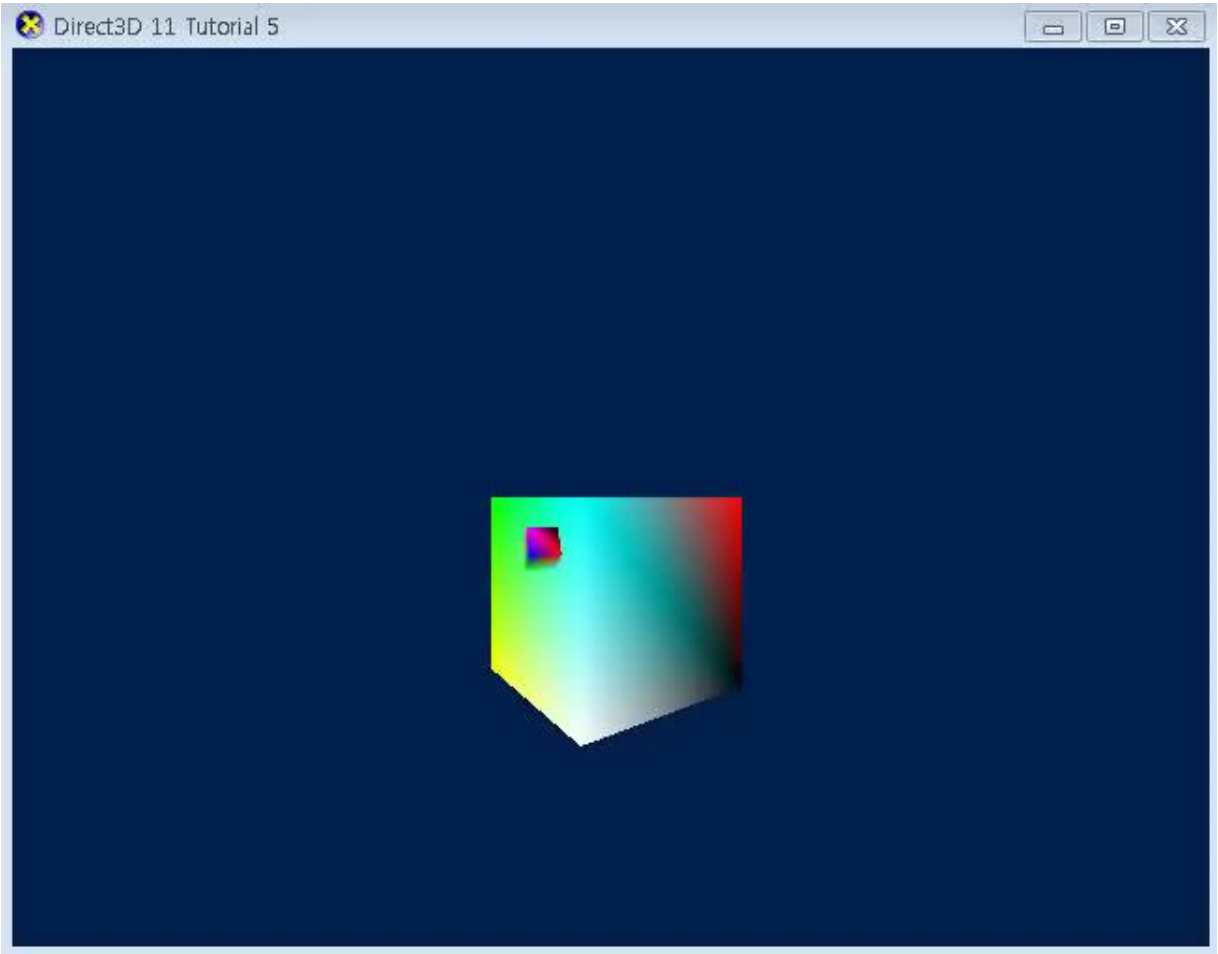
Create Orbit

```
// update variables for the 1st cube
ConstantBuffer cb1;
cb1.mWorld = XMMatrixTranspose( g_World );
cb1.mView = XMMatrixTranspose( g_View );
cb1.mProjection = XMMatrixTranspose( g_Projection );
g_pImmediateContext->UpdateSubresource( g_pConstantBuffer, 0,
                                        NULL, &cb1, 0, 0);

// render the 1st cube
g_pImmediateContext->VSSetShader( g_pVertexShader, NULL, 0 );
g_pImmediateContext->VSSetConstantBuffer( 0, 1, &g_pConstantBuffer );
g_pImmediateContext->PSSetShader( g_pPixelShader, NULL, 0 );
g_pImmediateContext->DrawIndexed( 36, 0, 0 );
```

Create Orbit

```
// update variables for the 2nd cube
ConstantBuffer cb2;
cb2.mWorld = XMMatrixTranspose( g_World2 );
cb2.mView = XMMatrixTranspose( g_View );
cb2.mProjection = XMMatrixTranspose( g_Projection );
g_pImmediateContext->UpdateSubresource( g_pConstantBuffer, 0,
                                        NULL, &cb2, 0, 0);

// render the 2nd cube
g_pImmediateContext->DrawIndexed( 36, 0, 0 );
```

Depth Buffer



Depth Buffer

- Keep track of he depth of every pixel drawn to the screen

- Check every pixel being drawn to the screen against the value stored in the depth buffer for that screen-space pixel

Depth Buffer

```
// create depth stencil texture
ID3D11Texture2D* g_pDepthStencil = NULL;
D3D11_TEXTURE2D_DESC descDepth;
ZeroMemory( & descDepth, sizeof(descDepth) );
descDepth.Width = width;
descDepth.Height = height;
descDepth.MipLevels = 1;
descDepth.ArraySize = 1;
descDepth.Format = DXGI_FORMAT_D24_UNORM_S8_UINT;
descDepth.SampleDesc.Count = 1;
descDepth.SampleDesc.Quality = 0;
descDepth.Usage = D3D11_USAGE_DEFAULT;
descDepth.BindFlags = D3D11_BIND_DEPTH_STENCIL;
descDepth.CPUAccessFlags = 0;
descDepth.MiscFlag = 0;
if( FAILED( g_pd3dDevice->CreateTexture2D( &descDepth, NULL,
                                           &g_pDepthStencil ) ) )
        return FALSE;
```

Depth Buffer

```
// create the depth stencil view
D3D11_DEPTH_STENCIL_VIEW_DESC descDSV;
ZeroMemory( &descDSV, sizeof(descDSV) );
descDSV.Format = descDepth.Format;
descDSV.ViewDimension = D3D11_DSV_DIMENSION_TEXTURE2D;

if( FAILED( g_pd3dDevice->CreateDepthStencilView(
                                        g_pDepthStencil, &descDSV,
                                        &g_pDepthStencilView ) ) )
        return FALSE;

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

Depth Buffer

- Must clear the depth buffer before rendering

```
// clear the depth buffer to 1.0 (max depth)
g_pImmediateContext->ClearDepthStencilView( g_pDepthStencilView,
                                            D3D11_CLEAR_DEPTH, 1.0f, 0 );
```