Beginning Direct3D Game Programming:

6. First Steps to Animation

jintaeks@gmail.com
Division of Digital Contents, DongSeo University.
April 2016
Understanding Transformations

✓ At the head of the pipeline, a model's vertices are declared relative to a **local coordinate system**. This is a local origin and an orientation.
✓ The first stage of the geometry pipeline transforms a model's vertices from their local coordinate system to a coordinate system that is used by all the objects in a scene. The process of reorienting the vertices is called the world transform.
In the next stage, the vertices that describe your 3D world are oriented with respect to a camera. That is, your application chooses a point-of-view for the scene, and world space coordinates are relocated and rotated around the camera's view, turning world space into camera space.
In this part of the pipeline, objects are usually scaled with relation to their distance from the viewer in order to give the illusion of depth to a scene; close objects are made to appear larger than distant objects, and so on.
In the final part of the pipeline, any vertices that will not be visible on the screen are removed.
World Transform

✓ You move an object in a 3D world using a world transformation.

✓ Figure 6.1 shows two cubes that are the same size but have different positions and orientations.
1. The cube is standing with its origin in the origin of the world coordinate system.
2. Rotate the cube.
3. Move the cube to its new position.
✓ 1. The cube is standing with its origin in the origin of the world coordinate system.
✓ 2. Move the cube to its new position.
✓ 3. Rotate the cube.

Figure 6.3
World transformations of a cube with steps 2 and 3 exchanged
✓ A 4×4 world matrix contains four vectors, which might represent the orientation and position of an object.

\[
\begin{bmatrix}
ux & uy & uz & 0 \\
vx & vy & vz & 0 \\
wx & wy & wz & 0 \\
tx & ty & tz & 1
\end{bmatrix}
\]

✓ The **u, v, and w vectors** describe the orientation of the object’s vertices compared to the world coordinate system.

✓ The **t vector** describes the position of the object’s vertex compared to the world coordinate system.

✓ Every vertex of the cube gets the correct orientation and position by being multiplied by this matrix.
To describe the position of the cube in Figure 6.4, you must multiply the following matrix by every vertex of the cube.

$$\begin{bmatrix}
1, 0, 0, 0 \\
0, 1, 0, 0 \\
0, 0, 1, 0 \\
2, 2, 2, 1
\end{bmatrix}$$

D3DMATRIX mat;
mat._11 = 1.0f; mat._12 = 0.0f; mat._13 = 0.0f; mat._14 = 0.0f;
mat._21 = 0.0f; mat._22 = 1.0f; mat._23 = 0.0f; mat._24 = 0.0f;
mat._31 = 0.0f; mat._32 = 0.0f; mat._33 = 1.0f; mat._34 = 0.0f;
mat._41 = 2.0f; mat._42 = 2.0f; mat._43 = 2.0f; mat._44 = 1.0f;
D3DXMatrixTranslation()

✓ D3DXMatrixTranslation() might look like this:

```c
inline VOID D3DXMatrixTranslation (D3DXMATRIX* m, FLOAT tx, FLOAT ty, FLOAT tz )
{
    D3DXMatrixIdentity(m);
    m._41 = tx; m._42 = ty; m._43 = tz;
}
```

= 
1 0 0 0
0 1 0 0
0 0 1 0
tx  ty tz 1
VOID D3DXMatrixRotationY( D3DXMATRIX* mat, FLOAT fRads )
{
    D3DXMatrixIdentity(mat);
    mat._11 = cosf( fRads );
    mat._13 = -sinf( fRads );
    mat._31 = sinf( fRads );
    mat._33 = cosf( fRads );
}

= cosf(fRads) 0 -sinf(fRads) 0
  0 0 0 0
sinf(fRads) 0 cosf(fRads) 0
  0 0 0 0
Concatenating Matrices

✓ One advantage of using matrices is that you can combine the effects of two or more matrices by multiplying them.

\[ C = M_1 \cdot M_2 \cdot M_{n-1} \cdot M_n \]

✓ Use the D3DXMatrixMultiply function to perform matrix multiplication.
D3DXMatrixMultiply()

D3DXMATRIX* D3DXMatrixMultiply(D3DXMATRIX* pOut, CONST D3DXMATRIX* pM1, CONST D3DMATRIX* pM2)
{
    FLOAT pM[16];
    ZeroMemory( pM, sizeof(D3DXMATRIX) );
    for( WORD i=0; i<4; i++ )
        for( WORD j=0; j<4; j++ )
            for( WORD k=0; k<4; k++ )
                pM[4*i+j] += pM1[4*i+k] * pM2[4*k+j];
    memcpy( pOut, pM, sizeof(D3DXMATRIX) );
    return (pOut);
}
After you prepare the world matrix, call the IDirect3DDevice9::SetTransform method to set it, specifying the D3DTS_WORLD macro for the first parameter.

```cpp
// Set up the rotation matrix to generate 1 full rotation (2*PI radians) 
// every 1000 ms. To avoid the loss of precision inherent in very high 
// floating point numbers, the system time is modulated by the rotation 
// period before conversion to a radian angle.
UINT iTime = timeGetTime() % 1000;
FLOAT fAngle = iTime * (2.0f * D3DX_PI) / 1000.0f;
D3DXMatrixRotationY(&matWorld, fAngle);
g_pd3dDevice->SetTransform(D3DTS_WORLD, &matWorld);
```
The view transform locates the viewer in world space, transforming vertices into camera space.

\[ V = T \cdot R_z \cdot R_y \cdot R_x \]

In this formula, \( V \) is the view matrix being created, \( T \) is a translation matrix that repositions objects in the world, and \( R_x \) through \( R_z \) are rotation matrices that rotate objects along the x-, y-, and z-axis.
Setting Up a View Matrix

✓ The following example creates a view matrix for left-handed coordinates.

```cpp
D3DXMATRIX mView;
D3DXVECTOR3 eye(2,3,3);
D3DXVECTOR3 at(0,0,0);
D3DXVECTOR3 up(0,1,0);
D3DXMatrixLookAtLH(&mView, &eye, &at, &up);
...
g_pd3dDevice->SetTransform(D3DTS_VIEW, &mView);
```
The projection matrix is typically a scale and perspective projection. The projection transformation converts the viewing frustum into a cuboid shape.

\[ \text{Aspect Ratio} = \frac{y}{x} = \frac{\tan(\text{vertical FOV}/2)}{\tan(\text{horizontal FOV}/2)} \]
How to Project?

Two triangles are similar. \( \triangle ABC \sim \triangle A'OC \)

\[ \frac{x}{x'} = \frac{z+d}{d} \]

\[ x' = \frac{xd}{z+d} \]

\[ y' = \frac{yd}{z+d} \]

\[ z' = 0 \]
\[
\begin{bmatrix}
    x' \\
    y' \\
    z' \\
    1
\end{bmatrix} \leftarrow \begin{bmatrix}
    x \\
    y \\
    0 \\
    z + d
\end{bmatrix} = \begin{bmatrix}
    d & 0 & 0 & 0 \\
    0 & d & 0 & 0 \\
    0 & 0 & 0 & 0 \\
    0 & 0 & 1 & d
\end{bmatrix} \begin{bmatrix}
    x \\
    y \\
    z \\
    1
\end{bmatrix}
\]

\[
\begin{bmatrix}
    x' \\
    y' \\
    z' \\
    1
\end{bmatrix} \leftarrow \begin{bmatrix}
    x \\
    y \\
    z + d
\end{bmatrix} / (z + d)
\]
In this matrix, $Z_n$ is the z-value of the near clipping plane. The variables $w$, $h$, and $Q$ have the following meanings. Note that $\text{fov}_w$ and $\text{fov}_h$ represent the viewport's horizontal and vertical fields of view, in radians.

$$w = \cot\left(\frac{\text{fov}_w}{2}\right)$$

$$h = \cot\left(\frac{\text{fov}_h}{2}\right)$$

$$Q = \frac{Z_g}{Z_g - Z_n}$$
Trigonometric Functions

✓ $z' = zQ - QZ_n$
✓ $z' = Z_n Q - QZ_n = 0$
✓ $z' = Z_f Q - QZ_n = Q(Z_f - Z_n) = Z_f$

<table>
<thead>
<tr>
<th>Function</th>
<th>Abbreviation</th>
<th>Description</th>
<th>Identities (using radians)</th>
</tr>
</thead>
<tbody>
<tr>
<td>sine</td>
<td>sin</td>
<td>opposite hypotenuse</td>
<td>$\sin \theta = \cos \left(\frac{\pi}{2} - \theta\right) = \frac{1}{\csc \theta}$</td>
</tr>
<tr>
<td>cosine</td>
<td>cos</td>
<td>adjacent hypotenuse</td>
<td>$\cos \theta = \sin \left(\frac{\pi}{2} - \theta\right) = \frac{1}{\sec \theta}$</td>
</tr>
<tr>
<td>tangent</td>
<td>tan (or tg)</td>
<td>opposite adjacent</td>
<td>$\tan \theta = \frac{\sin \theta}{\cos \theta} = \cot \left(\frac{\pi}{2} - \theta\right) = \frac{1}{\cot \theta}$</td>
</tr>
<tr>
<td>cotangent</td>
<td>cot (or cotan or cotg or ctg or ctn)</td>
<td>adjacent opposite</td>
<td>$\cot \theta = \frac{\cos \theta}{\sin \theta} = \tan \left(\frac{\pi}{2} - \theta\right) = \frac{1}{\tan \theta}$</td>
</tr>
<tr>
<td>secant</td>
<td>sec</td>
<td>hypotenuse adjacent</td>
<td>$\sec \theta = \csc \left(\frac{\pi}{2} - \theta\right) = \frac{1}{\cos \theta}$</td>
</tr>
<tr>
<td>cosecant</td>
<td>csc (or cosec)</td>
<td>hypotenuse opposite</td>
<td>$\csc \theta = \sec \left(\frac{\pi}{2} - \theta\right) = \frac{1}{\sin \theta}$</td>
</tr>
</tbody>
</table>
Cot(x) graph
D3DXMatrixPerspectiveFovLH()

D3DXMATRIX * D3DXMatrixPerspectiveFovLH(
    D3DXMATRIX * pOut,
    FLOAT fovy,
    FLOAT Aspect,
    FLOAT zn,
    FLOAT zf
);

// For the projection matrix, we set up a perspective transform (which
// transforms geometry from 3D view space to 2D viewport space, with
// a perspective divide making objects smaller in the distance). To build
// a perspective transform, we need the field of view (1/4 pi is common),
// the aspect ratio, and the near and far clipping planes (which define at
// what distances geometry should be no longer be rendered).
D3DXMATRIXA16 matProj;
D3DXMatrixPerspectiveFovLH( &matProj, D3DX_PI / 4, 1.0f, 1.0f, 100.0f );
g_pd3dDevice->SetTransform( D3DTS_PROJECTION, &matProj );
Working with the Viewport

✓ Conceptually, a **viewport** is a two-dimensional (2D) rectangle into which a 3D scene is projected.

✓ In Direct3D, the rectangle exists as coordinates within a Direct3D **surface** that the system uses as a **rendering target**.
A viewing frustum is a 3D volume in a scene positioned relative to the viewport's camera.

The shape of the volume affects how models are projected from camera space onto the screen.
The viewing frustum is defined by fov (field of view) and by the distances of the front and back clipping planes, specified in z-coordinates.
Viewport Rectangle

✓ You define the **viewport rectangle** in C++ by using the **D3DVIEWPORT9** structure.

- The **D3DVIEWPORT9** structure is used with the following viewport manipulation methods exposed.
  - **IDirect3DDevice9::GetViewport**
  - **IDirect3DDevice9::SetViewport**

✓ Use **IDirect3DDevice9::Clear** to **clear** the viewport.
You can use the following settings for the members of the D3DVIEWPORT9 structure to achieve this in C++.

```c
typedef struct D3DVIEWPORT9 {
    DWORD X;
    DWORD Y;
    DWORD Width;
    DWORD Height;
    float MinZ;
    float MaxZ;
} D3DVIEWPORT9, *LPD3DVIEWPORT9;
```

D3DVIEWPORT9 viewData = { 0, 0, width, height, 0.0f, 1.0f };

After setting values in the D3DVIEWPORT9 structure, apply the viewport parameters to the device by calling its IDirect3DDevice9::SetViewport method.

```c
HRESULT hr;
hr = pd3dDevice->SetViewport(&viewData);
if(FAILED(hr))
    return hr;
```
A depth buffer, often called a **z-buffer** or a **w-buffer**, is a property of the device that stores depth information to be used by Direct3D.
Creating a Depth Buffer

✓ To create a depth buffer that is managed by Direct3D, set the appropriate members of the D3DPRESENT_PARAMETERS structure.

```c
D3DPRESENT_PARAMETERS d3dpp;
ZeroMemory( &d3dpp, sizeof(d3dpp) );
d3dpp.Windowed               = TRUE;
d3dpp.SwapEffect             = D3DSWAPEFFECT_COPY;
d3dpp.EnableAutoDepthStencil = TRUE;
d3dpp.AutoDepthStencilFormat = D3DFMT_D16;
```

✓ ...

```c
if( FAILED( g_pD3D->CreateDevice( D3DADAPTER_DEFAULT
  , D3DDEVTYPE_HAL, hWnd
  , D3DCREATE_SOFTWARE_VERTEXPROCESSING
  , &d3dpp, &d3dDevice ) ) )
  return E_FAIL;
```
Quaternions contain a scalar component and a 3D vector component.

\[
\begin{bmatrix}
  w \\
  v \\
  w (x, y, z)
\end{bmatrix}
\]

With the Direct3D quaternion class, you use the w, x, y, and z components by adding .x, .y, .z, and .w to the variable names.

Figure 6.13
A quaternion representing a rotation around Axis A
D3DXQUATERNION * D3DXQuaternionRotationAxis(
    D3DXQUATERNION *pOut,
    CONST D3DXVECTOR3 *pV,
    FLOAT Angle
);

D3DXQUATERNION * D3DXQuaternionRotationMatrix(
    D3DXQUATERNION * pOut,
    CONST D3DXMATRIX * pM
);

D3DXMATRIX * D3DXMatrixRotationQuaternion(
    D3DXMATRIX *pOut,
    CONST D3DXQUATERNION *pQ
);
This tutorial introduces the concept of matrices and shows how to use them.

- **Step 1** - Defining the World Transformation Matrix
- **Step 2** - Defining the View Transformation Matrix
- **Step 3** - Defining the Projection Transformation Matrix
Step 1 - Defining the World Transformation Matrix

✓ The world transformation matrix defines how to translate, scale, and rotate the geometry in the 3D model space.

D3DXMATRIX matWorld;
D3DXMatrixRotationY( &matWorld, timeGetTime() / 150.0f );
g_pd3dDevice->SetTransform( D3DTS_WORLD, &matWorld );
Step 2 - Defining the View Transformation Matrix

- The view transformation matrix defines the position and rotation of the view. The view matrix is the camera for the scene.

```cpp
D3DXVECTOR3 vEyePt ( 0.0f, 3.0f,-5.0f );
D3DXVECTOR3 vLookatPt( 0.0f, 0.0f, 0.0f );
D3DXVECTOR3 vUpVec ( 0.0f, 1.0f, 0.0f );
D3DXMATRIXA16 matView;
D3DXMatrixLookAtLH( &matView, &vEyePt, &vLookatPt, &vUpVec );
g_pd3dDevice->SetTransform( D3DTS_VIEW, &matView );
```
Step 3 - Defining the Projection Transformation Matrix

✓ The projection transformation matrix defines how geometry is transformed from 3D view space to 2D viewport space.

D3DXMATRIX matProj;
D3DXMatrixPerspectiveFovLH( &matProj, D3DX_PI/4, 1.0f, 1.0f, 100.0f );
g_pd3dDevice->SetTransform( D3DTS_PROJECTION, &matProj );
Practice

✓ Achieve similar animation by setting View Matrix.
✓ Rotate the triangle about x-axis.
✓ Translate the triangle about (5,5,0), then rotate.
✓ Rotate the triangle, the translate (5,5,0).
  – Check differences between two transforms.
✓ Modify the Fov of Projection matrix and check the visual differences.
MY BRIGHT FUTURE
동서대학교
DSU Dongseo University 동서대학교