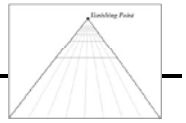
Rendering Pipeline

305890
Spring 2010
4/9/2010
Kyoung Shin Park

Overview

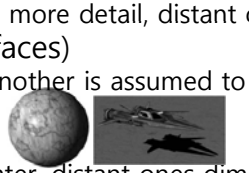
- 3D Illusion
- 3D Object representations in Direct3D
- Model the virtual camera
- Understand the rendering pipeline
  - The process of taking a geometric description of a 3D scene and generating a 2D image from it

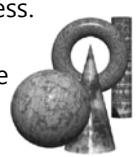
3D Illusion



- Linear perspective
  - Objects get smaller the further away they are and parallel line converge in distance.
- Size of known objects
  - We expect certain object to be smaller than others.

- Detail (texture gradient)
  - Close objects appear in more detail, distant objects less.
- Occlusion (hidden surfaces)
  - An object that blocks another is assumed to be in the foreground.

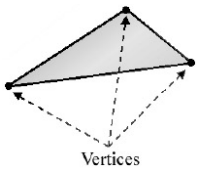


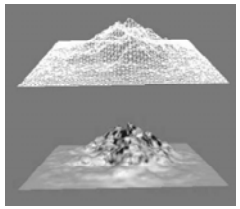


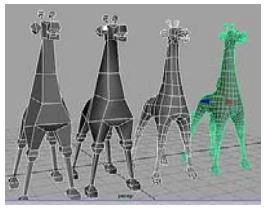
- Lighting and Shadows
  - Closer objects are brighter, distant ones dimmer. Shadow is a form of occlusion.
- Relative motion (motion parallax due to head motion)
  - Objects further away seem to move more slowly than objects in the foreground.

Model Representation

- A scene is composed of objects or models
- An object is represented as a triangle mesh approximation
- A triangle is defined by its the three vertices
- Model representation in Direct3D
  - Vertex format
  - Triangle
  - Index







Vertex Format

- Vertex
  - Spatial location
  - Additional properties: normal, color, texture coordinates
- 1st Step: Declare a custom vertex structure
  - Position, Color

```
struct ColorVertex {
    float _x, _y, _z;          // position
    DWORD _color;
};
```

  - Position, Normal, Texture coordinates

```
struct NormalTexVertex {
    float _x, _y, _z;          // position
    float _nx, _ny, _nz;       // normal vector
    float _u, _v;              // texture coords
};
```

Vertex Format

- 2nd Step: Declare a vertex declaration by an array of D3DVERTEXELEMENT9 elements

```
typedef struct _D3DVERTEXELEMENT9 {
    BYTE Stream; // we just use one vertex stream - 0
    BYTE Offset; // offset from the start of the vertex structure (in bytes)
    BYTE Type; // data types of the vertex element (D3DDECLTYPE_FLOAT1/2/3/4)
    BYTE Method; // tessellation method
    BYTE Usage; // usage (D3DDECLUSAGE_POSITION/NORMAL/TEXCOORD/COLOI
    BYTE UsageIndex; // used to identify multiple vertex components of the same
                     // usage. UsageIndex is [0, 15]
} D3DVERTEXELEMENT9;
```

Vertex Format

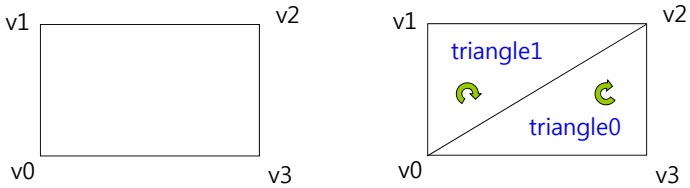
```
struct Vertex {
    D3DXVECTOR3 pos;
    D3DXVECTOR3 normal0;
    D3DXVECTOR3 normal1;
    D3DXVECTOR3 normal2;
};

D3DVERTEXELEMENT9 decl[] = {
  {0, 0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_POSITION, 0},
  {0, 12, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_NORMAL, 0},
  {0, 24, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_NORMAL, 1},
  {0, 36, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_NORMAL, 2},
  D3DDECL_END()
};

// D3DDECL_END macro is used to mark the end of a D3DVERTEXELEMENT9 array
```

Triangle

- Triangle
  - The basic building blocks of 3D objects
  - For example, to construct a quad we break it into 2 triangles.

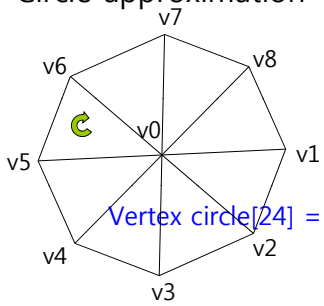



```
Vertex quad[6] = { v0, v1, v2,      // triangle 0
                   v0, v2, v3 };    // triangle 1
```

  - Direct3D vertex winding order is CW

Triangle

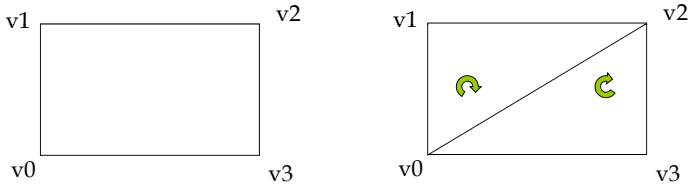
- Circle approximation




```
Vertex circle[24] = { v0, v1, v2,      // triangle 0
                      v0, v2, v3,      // triangle 1
                      v0, v3, v4,      // triangle 2
                      v0, v4, v5,      // triangle 3
                      v0, v5, v6,      // triangle 4
                      v0, v6, v7,      // triangle 5
                      v0, v7, v8,      // triangle 6
                      v0, v8, v1};     // triangle 7
```

Index

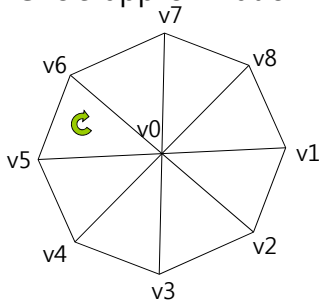
- Index list
  - Triangles that form a 3D object share many of the same vertices
  - 2 reasons why we do not want to duplicate vertices: increased memory & graphics processing
  - Hence, we build *vertex list* and *index list*




```
Vertex vertexList[4] = { v0, v1, v2, v3};      // vertex list
WORD indexList[6] = {0, 1, 2,                  // index list
                     0, 2, 3 };
```

Index

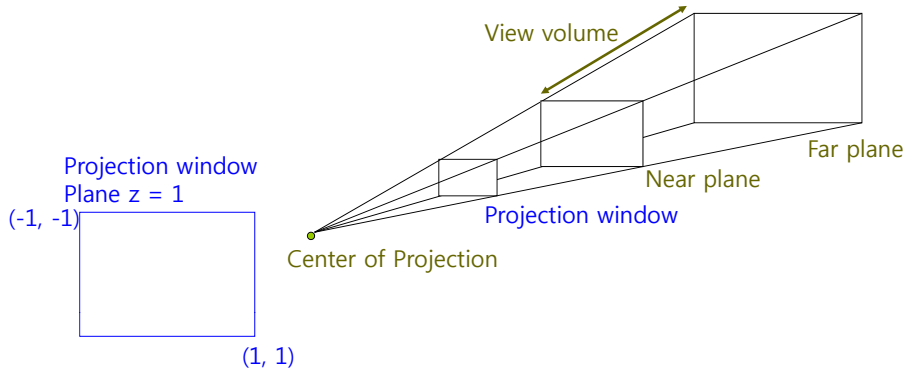
- Circle approximation




```
Vertex vertexList[9] = {v0, v1, v2, v3, v4, v5, v6, v7, v8};
WORD IndexList[24] = { 0, 1, 2, // triangle 0
                       0, 2, 3, // triangle 1
                       ...
                       0, 7, 8, // triangle 6
                       0, 8, 1}; // triangle 7
```

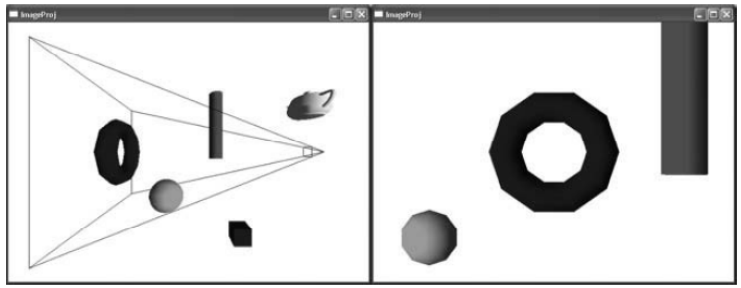
Virtual Camera

- Virtual Camera
  - Camera specifies what part of the world the viewer can see and thus what part of the world we need to generate a 2D image.
  - Projection window is defined as plane z=1, in Direct3D.



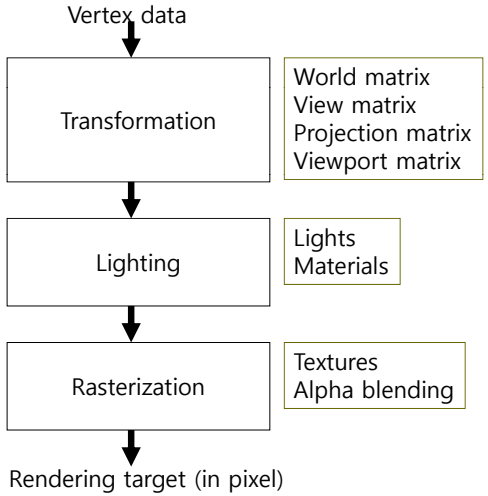
Rendering Pipeline

- Rendering pipeline refers to the entire sequence of steps necessary to generate a 2D image that can be displayed on a monitor screen based on what the virtual camera sees.



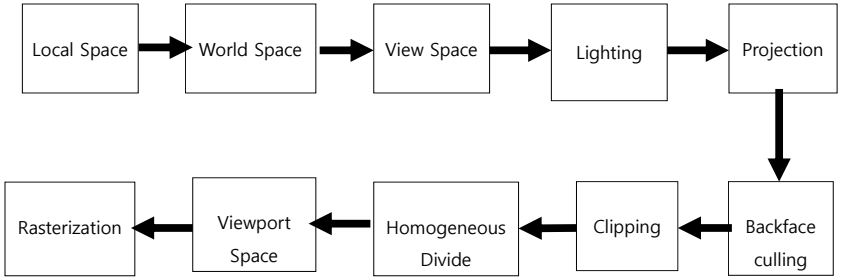
Rendering Pipeline

- 3D scene => 2D image



Rendering Pipeline

- Direct3D Rendering pipeline



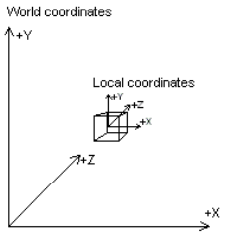
Local Space & World Space

- Local space (i.e., Modeling space)
  - The 3D object is constructed in a local coordinate system, where the object is the center of the coordinate system
- World space
  - Once the 3D model is built in local space, it is placed in a scene in world space, by executing a change of coordinates transformation (called *world transform*).

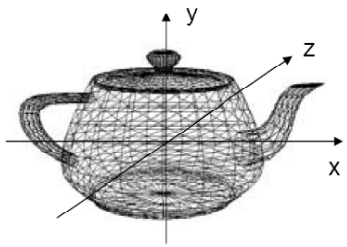
$$W = \begin{pmatrix} r_x & r_y & r_z & 0 \\ u_x & u_y & u_z & 0 \\ f_x & f_y & f_z & 0 \\ p_x & p_y & p_z & 1 \end{pmatrix}$$

$\vec{p}$ *is the origin*

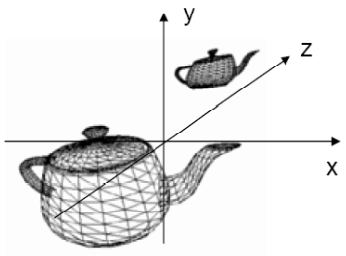
$\vec{r}, \vec{u}, \vec{f}$ *of LCS*



Local Space & World Space



local space



world space

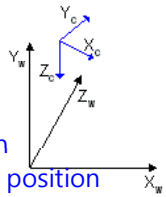
Modeling Transformation

- Local space => World space
  - IDirect3DDevice::SetTransform(D3DTS_WORLD, &worldMatrix);

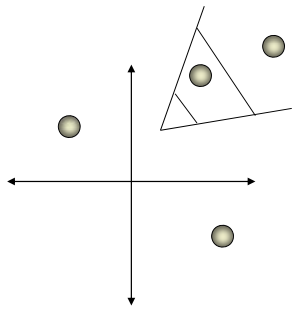
```
// place a cube in (-3, 2, 6), a sphere in (5, 0, -2)
D3DXMATRIX cubeWorldMatrix;
D3DXMatrixTranslation(&cubeWorldMatrix, -3.0, 2.0, 6.0);
D3DXMATRIX sphereWorldMatrix;
D3DXMatrixTranslation(&sphereWorldMatrix, 5.0, 0.0, -2.0);
// set transform for cube
Device->SetTransform(D3DTS_WORLD, &cubeWorldMatrix);
drawCube();
// set transform for sphere
Device->SetTransform(D3DTS_WORLD, &sphereWorldMatrix);
drawSphere();
```

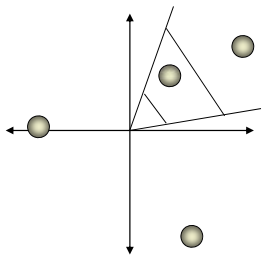
View Space

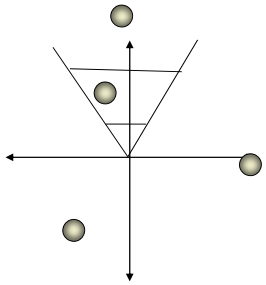
- Geometry object and camera is specified in world space, and then transformed to view space for projection.
- View space transformation
  - Translate the camera to the origin of world space, and then rotate it to align into +z-axis.
- World space => view space
  - D3DXMATRIX *D3DMatrixLookAtLH (
    D3DXMATRIX* pOut,
    CONST D3DXVECTOR3* pEye, // camera position
    CONST D3DXVECTOR3* pAt, // camera look-at position
    CONST D3DXVECTOR3* pUp // world up (0, 1, 0)
    );



View Space


Objects and the camera in World Space

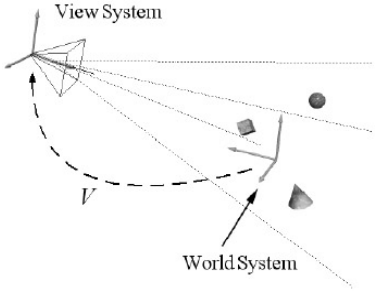

Translate the camera to the origin of World Space

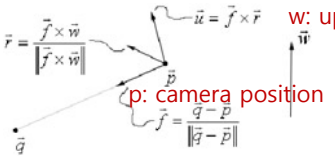

Rotate the camera to align into +Z-axis. Objects are also transformed.

View Space

$$V = (RT)^{-1} = T^{-1}R^{-1}$$

$$= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -p_x & -p_y & -p_z & 1 \end{pmatrix} \begin{pmatrix} r_x & u_x & f_x & 0 \\ r_y & u_y & f_y & 0 \\ r_z & u_z & f_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} r_x & u_x & f_x & 0 \\ r_y & u_y & f_y & 0 \\ r_z & u_z & f_z & 0 \\ -\vec{p}\bullet\vec{r} & -\vec{p}\bullet\vec{u} & -\vec{p}\bullet\vec{f} & 1 \end{pmatrix}$$





Viewing Transformation

- World space => View space
  - IDirect3DDevice::SetTransform(D3DTS_VIEW, &viewMatrix);

```
// the camera is located in (5, 3, -10), looking down the origin (0, 0, 0)
// set camera
D3DXVECTOR3 position(5.0, 3.0, -10.0);
D3DXVECTOR3 lookat(0.0, 0.0, 0.0);
D3DXVECTOR3 worldup(0.0, 1.0, 0.0);
// set view matrix
D3DMATRIX viewMatrix;
D3DXMatrixLookAtLH(&viewMatrix, &position, &lookat, &worldup);
Device->SetTransform(D3DTS_VIEW, &viewMatrix);
```

Lighting

- Lighting
  - Lights are specified directly in World Space relative to the overall scene.
  - We can always transform lights into local space or view space.

Projection

- Projection
  - All the vertices of the 3D scene are in View Space and lighting has been completed, a projection transformation is applied.
  - Perspective projection vs. Orthogonal projection
- Projection matrix

```
D3DXMATRIX *D3DXMatrixPerspectiveFovLH(
    D3DXMATRIX *pOut,
    FLOAT fovY,       // field of view in y-axis (in radian)
    FLOAT Aspect,     // aspect  ratio (= screen width/screen height)
    FLOAT zn,         // z-value of near plane
    FLOAT zf // z-value of far plane
)
```

Aspect ratio는 projection window(정사각형)을 screen window space(직사각형)으로 만드는 과정에서 왜곡을 보정하는 역할

Perspective Projection

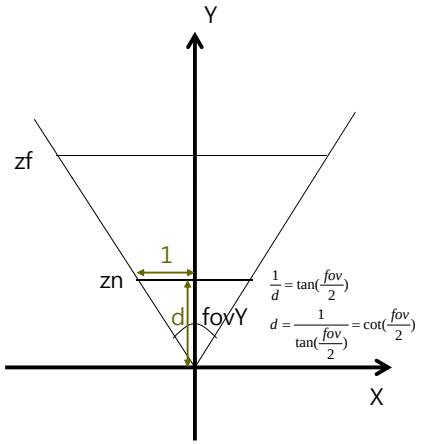
- Projection plane in front of the center of projection

$$\begin{pmatrix} xScale & 0 & 0 & 0 \\ 0 & yScale & 0 & 0 \\ 0 & 0 & \frac{zf}{zf-zn} & 1 \\ 0 & 0 & \frac{-zn*zf}{zf-zn} & 0 \end{pmatrix}$$

$$where\ yScale = \cot(fovY/2)$$

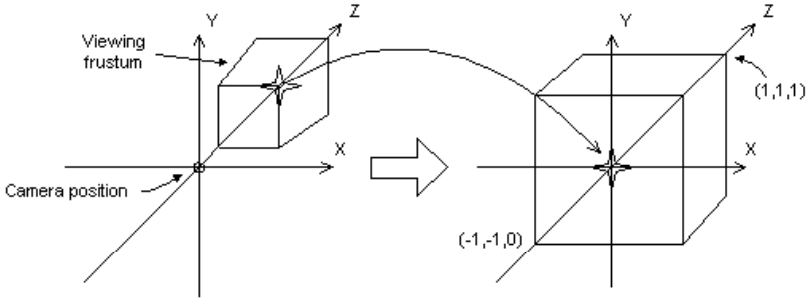
$$xScale = yScale / Aspect$$

$$Aspect = weight / height$$



Perspective Projection

- Direct3D view volume normalization
  - (-x, -y, zn) → (-1, -1, 0)
  - (x, y, zf) → (1, 1, 1)



Projection Transformation

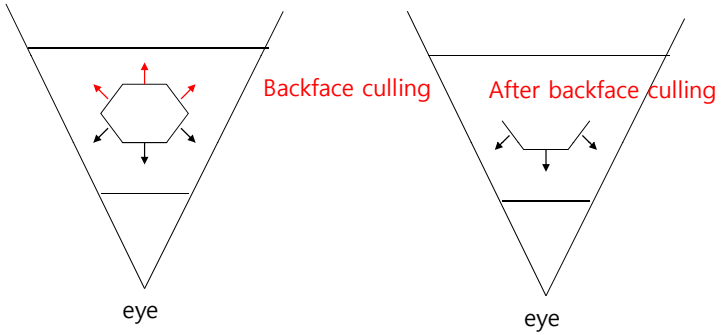
- Projection Transformation
  - IDirect3DDevice::SetTransform(D3DTS_PROJECTION, &projMatrix);

```
// 90 degree FOV, near plane at 1.0, far plane at 1000.0 frustum
// projection matrix.
// set camera
D3DXMATRIX projMatrix;
D3DXMatrixPerspectiveFovLH(
    &projMatrix,
    PI*0.5f,
    (float)width/(float)height,
    1.0f, 100.0f);
Device->SetTransform(D3DTS_PROJECTION, &projMatrix);
```

Backface culling

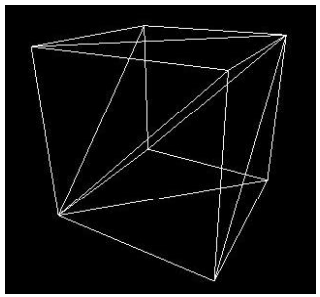
- Backface culling
  - A polygon has the front face and the back face.
  - Backface culling can quickly discard about half of the scene's dataset from further processing – an excellent speed up.
- Determine which polygons are front facing or back facing
  - By default, triangles with clockwise winding order are front facing
  - Visibility test: planeNormal • viewVector > 0
- Set culling
  - Device->setRenderState(D3DRS_CULLMODE, Value);
  - Value
    - D3DCULL_NONE: disable backface culling
    - D3DCULL_CW: triangles with a clockwise wind are culled
    - D3DCULL_CCW: triangles with a counterclockwise wind are culled (default)

Backface culling

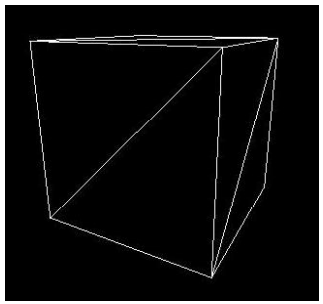


Backface culling

No Culling (All faces are seen)

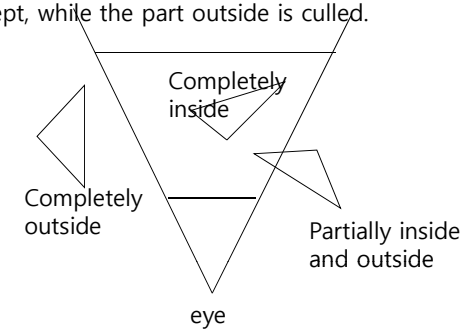


Backface Culling



Clipping

- Clipping
  - Clipping culls the geometry that is outside the viewing volume
  - 3 possible locations of triangle in the frustum:
    - Completely inside: it is kept
    - Completely outside: it is culled
    - Partially inside: then, the triangle is split into two parts. The part inside the frustum is kept, while the part outside is culled.
  - D3DRS_CLIPPING
    - Enable clipping or not



Viewport Transformation

- Viewport Transformation
  - Projection window => viewport (on screen)

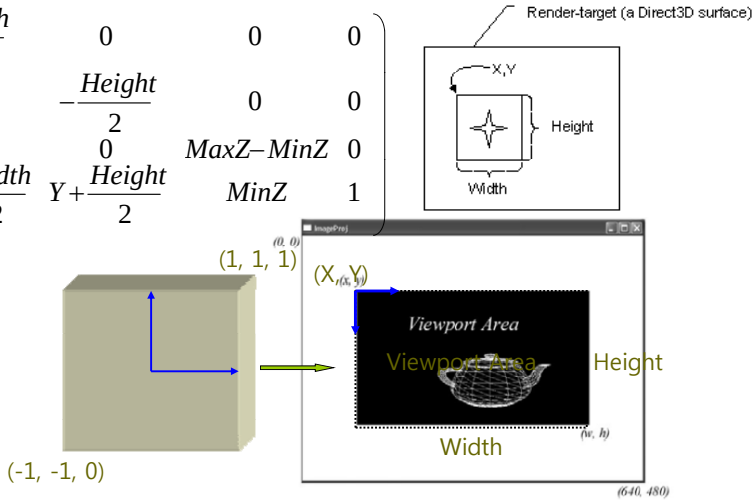
```
typedef struct _D3DVIEWPORT9 {
    DWORD X;        // pixel coords of the upper-left corner
    DWORD Y;        // pixel coords of the upper-left corner
    DWORD Width;    // width in pixels
    DWORD Height;   // height in pixels
    float MinZ;     // range of depth values
    float MaxZ;     // range of depth values
} D3DViewPORT9;
```

- Viewport matrix
  - D3DVIEWPORT9 vp = {0, 0, 640, 480, 0, 1};
  - Device->SetViewport(&vp);

Viewport

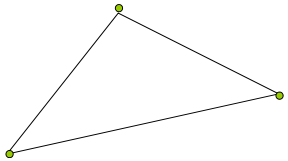
- Viewport Matrix

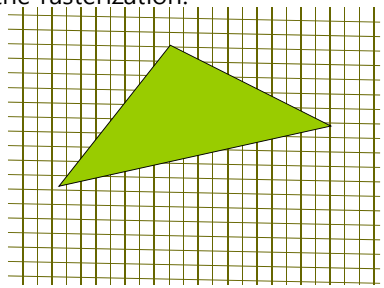
$$\begin{pmatrix} \frac{Width}{2} & 0 & 0 & 0 \\ 0 & -\frac{Height}{2} & 0 & 0 \\ 0 & 0 & MaxZ-MinZ & 0 \\ X+\frac{Width}{2} & Y+\frac{Height}{2} & MinZ & 1 \end{pmatrix}$$



Rasterization

- Rasterization
  - After the vertices are transformed to the back buffer, we have a list of 2D triangles in image space to be processed one by one.
  - Rasterization is responsible for computing the colors of the individual pixels that make up the interiors and boundaries of these triangles.
  - Pixel operations like texturing, pixel shaders, depth buffering, and alpha blending occur in the rasterization.





Reference

- Direct3D Transformation Pipeline - http://msdn2.microsoft.com/en-us/library/bb206260.aspx