Rendering Pipeline

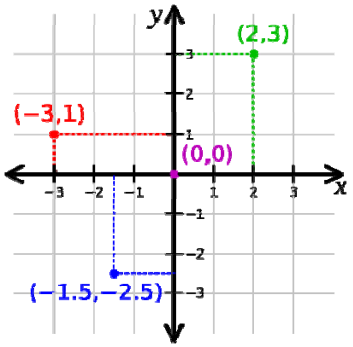
305890
Spring 2013
3/15/2013
Kyoung Shin Park

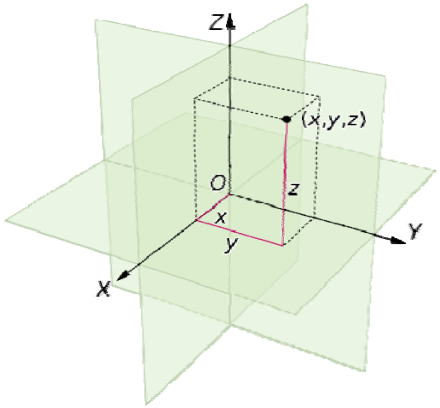
Overview

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

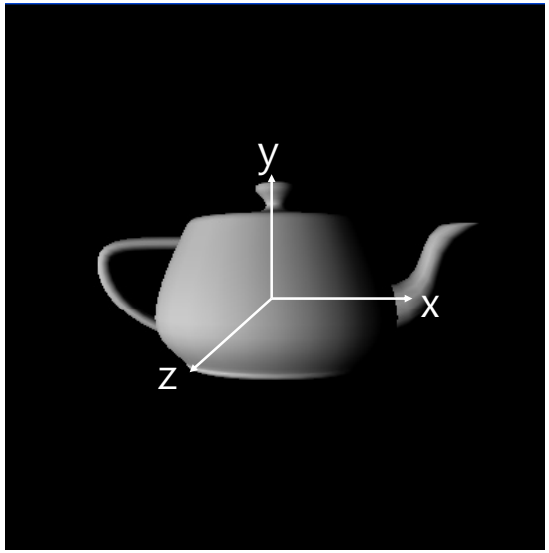
Coordinate Systems

- 2D Cartesian Coordination Systems
- 3D Cartesian Coordination Systems



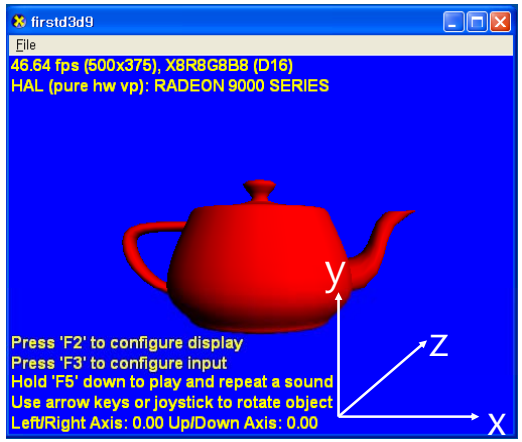


3D Coordinate Systems



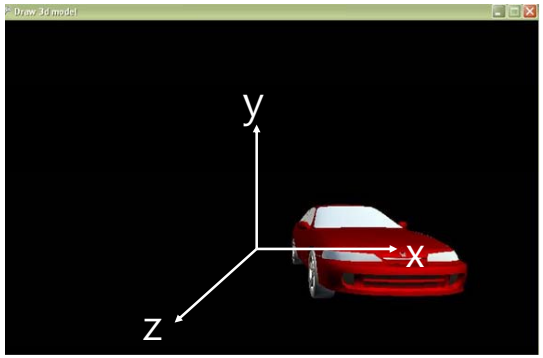
- OpenGL coordinate system is right-handed
- x+ to the right
- y+ up
- z+ coming out of the screen

3D Coordinate Systems



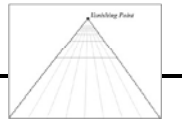
- Direct3D coordinate system is left-handed
- x+ to the right
- y+ up
- z+ forward

3D Coordinate Systems



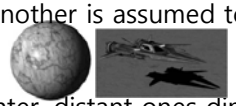
- XNA coordinate system is right-handed
- Same as OpenGL

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.

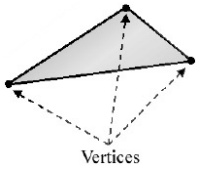
- Detail (texture gradient)
  - Close objects appear in more detail, distant objects less.
- Occlusion (hidden surfaces)
  - An object that blocks another is assumed to be in the foreground.

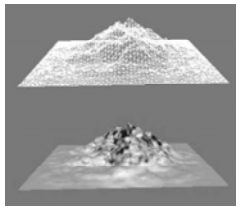


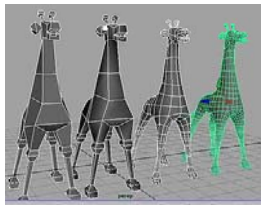
- Lighting and Shadows
  - Closer objects are brighter, distant ones dimmer. Shadow is a form of occlusion.
- Relative motion (motion parallax due to head motion)
  - Objects further away seem to move more slowly than objects in the foreground.

3D 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
  - Vertex format
  - Triangle
  - Index

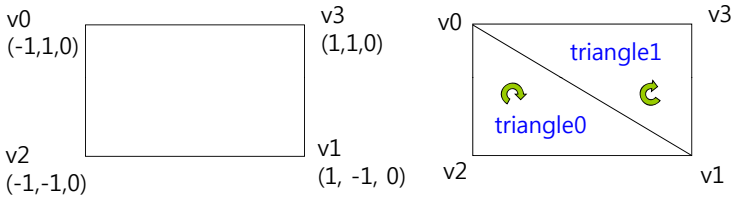






Triangle

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

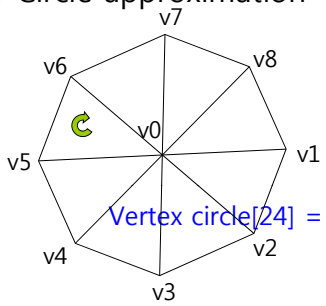



```
Vertex quad[6] = { v0, v1, v2,        // triangle 0
                   v0, v3, v1 };      // triangle 1
```

  - XNA vertex winding order is CW (same as Direct3D; opposite to OpenGL)

Triangle

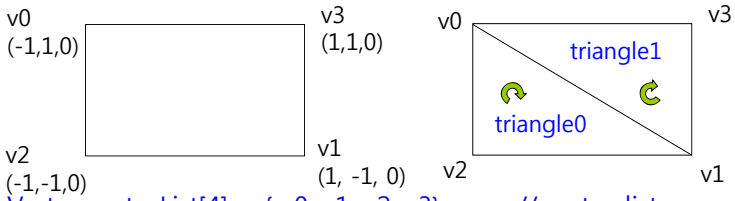
- Circle approximation




```
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

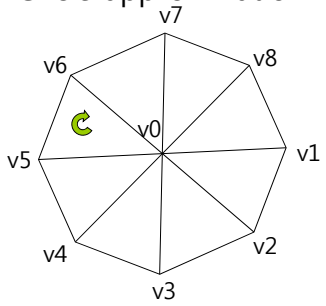
- Index list
  - Triangles that form a 3D object share many of the same vertices
  - 2 reasons why we do not want to duplicate vertices: increased memory & graphics processing
  - Hence, we build *vertex list* and *index list*




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

Index

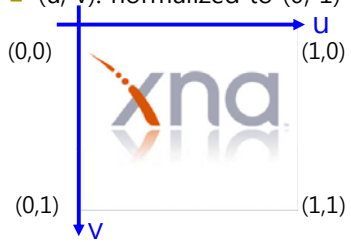
- Circle approximation



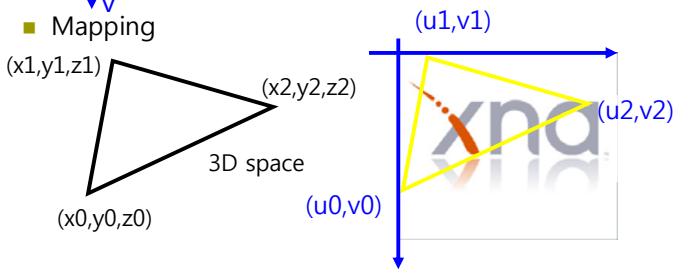

```
Vertex vertexList[9] = {v0, v1, v2, v3, v4, v5, v6, v7, v8};
WORD IndexList[24] = { 0, 1, 2, // triangle 0
                       0, 2, 3, // triangle 1
                       ...
                       0, 7, 8, // triangle 6
                       0, 8, 1}; // triangle 7
```

Texture Coordinates

- Texture Coordinates (same as Direct3D)
  - (u, v): normalized to (0, 1)



  - Mapping



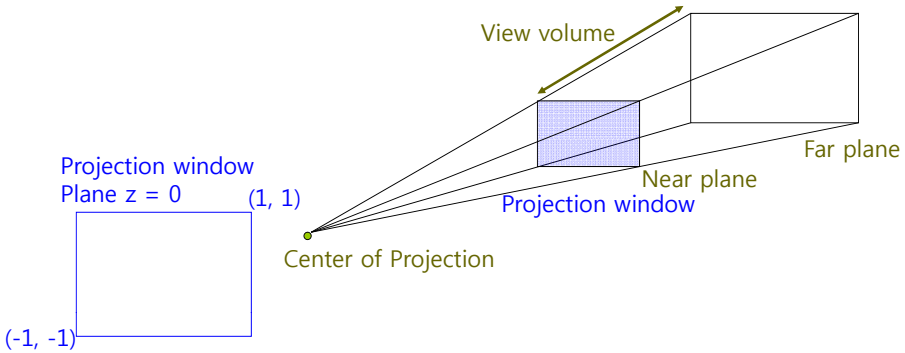
VertexPositionNormalTexture

- Vertex structure include texture coordinates

```
struct VertexPositionNormalTexture : IVertexType {
    public Vector3 Normal;
    public Vector3 Position;
    public Vector2 TextureCoordinate;
    public static readonly VertexDeclaration VertexDeclaration;
    public VertexPositionNormalTexture(Vector3 position, Vector3 normal,
    Vector2 textureCoordinate);
    public static bool operator !=(VertexPositionNormalTexture left,
    VertexPositionNormalTexture right);
    public static bool operator ==(VertexPositionNormalTexture left,
    VertexPositionNormalTexture right);
    public override bool Equals(object obj);
    public override int GetHashCode();
    public override string ToString();
}
```

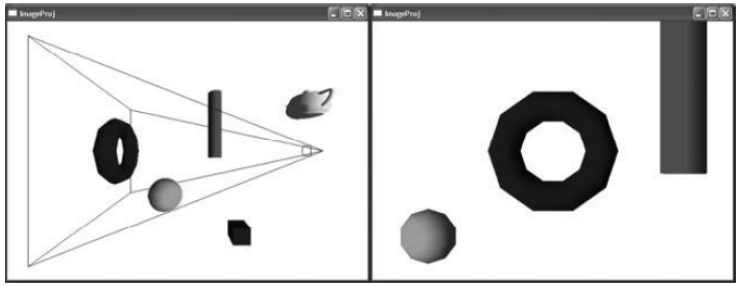
Virtual Camera

- 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=0, in XNA.



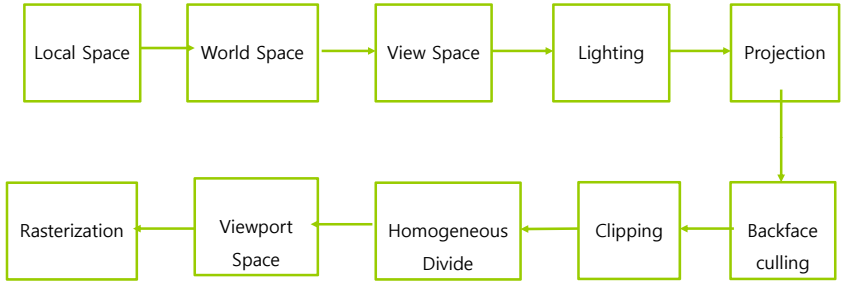
Rendering Pipeline

- 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.



DirectX9 Rendering Pipeline

- DirectX9 geometry stage 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*

Modeling Transformation

- Local space => World space

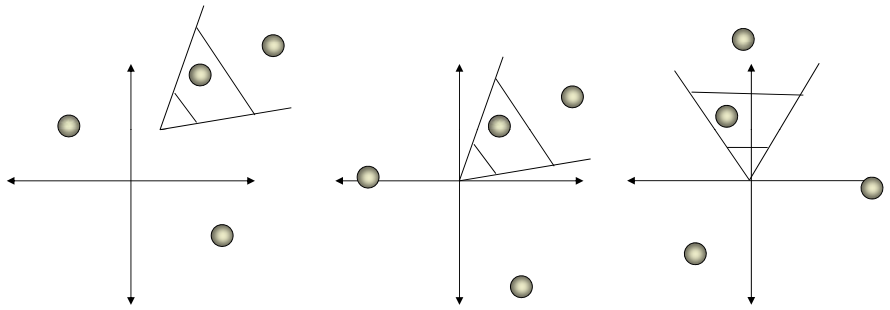
```
// place a rectangle in (3, 0, -10)
world = Matrix.CreateTranslation(new Vector3(3.0f, 0, -10.0f));
DrawRectangle(ref world);

// set transform for rectangle
world = Matrix.CreateScale(0.75f) *
    Matrix.CreateRotationX(MathHelper.ToRadians(15.0f)) *
    Matrix.CreateRotationY(MathHelper.ToRadians(15.0f)) *
    Matrix.CreateTranslation(new Vector3(-3.0f, -1.0f, -5.0f));
DrawRectangle(ref world);
```

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
  - void Matrix.CreateLookAt (
    ref Vector3 cameraPosition, // camera position
    ref Vector3 cameraTarget, // camera look-at position
    ref Vector3 cameraUpVector, // world up (0, 1, 0)
    out Matrix result // ViewMatrix
    );

View Space



Objects and the camera in World Space

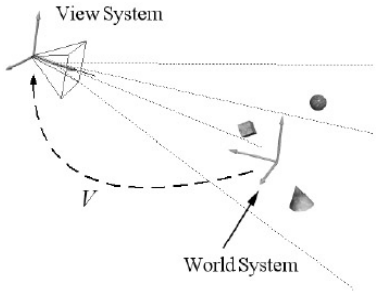
Translate the camera to the origin of World Space

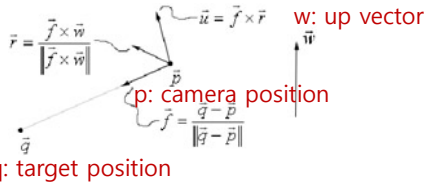
Rotate the camera to align into +Z-axis. Objects are also transformed.

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

```
// the camera is located in (0, 0, 3), looking down the origin (0, 0, 0)
// set camera
private Vector3 cameraPosition = new Vector3(0.0f, 0.0f, 3.0f);
private Vector3 cameraTarget = Vector3.Zero;
private Vector3 cameraUpVector = Vector3.Up;
// set view matrix
private Matrix view;
Matrix.CreateLookAt(ref cameraPosition, ref cameraTarget, ref cameraUpVector, out view);

private BasicEffect effect;
effect.View = view;
```

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

```
void Matrix.CreatePerspectiveFieldOfView(
    float fieldOfView,// field of view in y-axis (in radian)
    float aspectRatio,// aspect  ratio (= screen width/screen height)
    float nearPlaneDistance,  // z-value of near plane
    float farPlaneDistance,    // z-value of far plane
    out Matrix result          // ProjectionMatrix
)
```

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

Perspective Projection

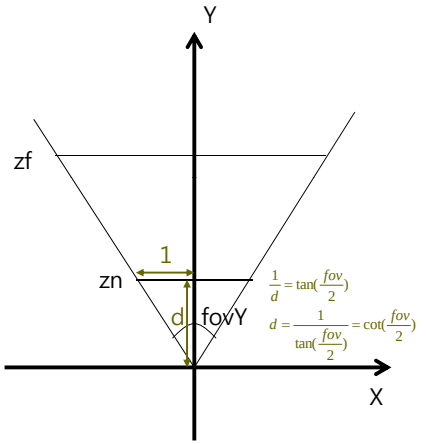
- Projection plane in front of the center of projection

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

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

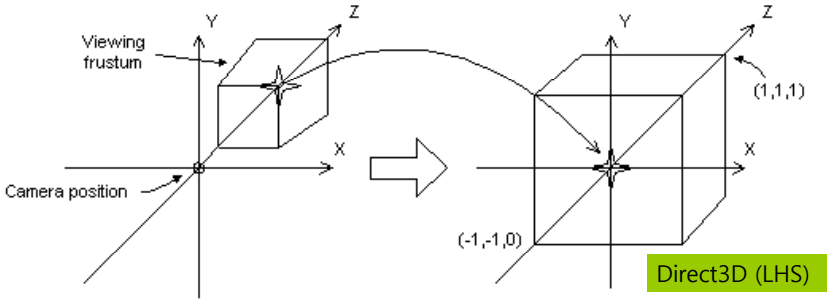
$$xScale = yScale / Aspect$$

$$Aspect = weight / height$$



Perspective Projection

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



Projection Transformation

- Projection Transformation

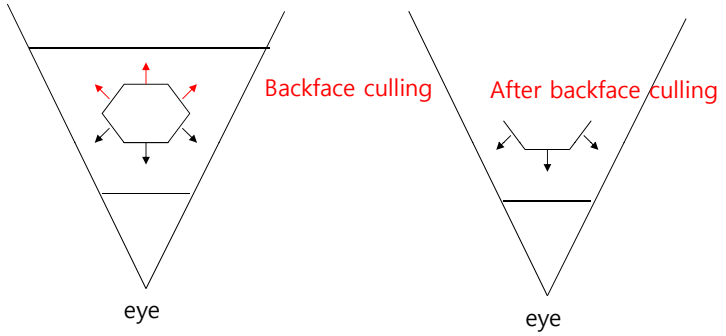
```
// 45 degree FOV, near plane at 0.0001, far plane at 1000.0 frustum
// projection matrix.
// set camera
private Matrix projection;
float aspectRatio = (float)graphics.GraphicsDevice.Viewport.Width/
    (float)graphics.GraphicsDevice.Viewport.Height;
Matrix.CreatePerspectiveFieldOfView(
    Math.Helper.PiOver4, aspectRatio, 1.0f, 100.0f, out projection);

private BasicEffect effect;
effect.Projection = projection;
```

Backface culling

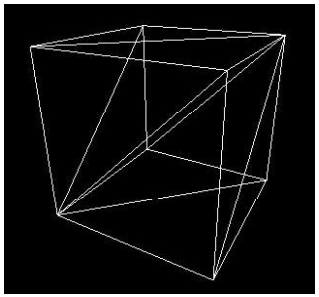
- Backface culling
  - A polygon has the front face and the back face.
  - Backface culling can quickly discard about half of the scene's dataset from further processing – an excellent speed up.
- Determine which polygons are front facing or back facing
  - By default, triangles with clockwise winding order are front facing
  - Visibility test: planeNormal • viewVector > 0
- Set culling
  - RasterizerState.CullMode = CullMode.None;
  - Value
    - NONE: disable backface culling
    - CW: triangles with a clockwise winding are culled
    - CCW: triangles with a counterclockwise winding are culled (default)

Backface culling

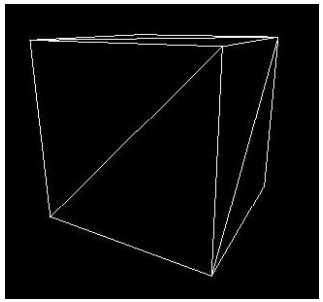


Backface culling

No Culling (All faces are seen)

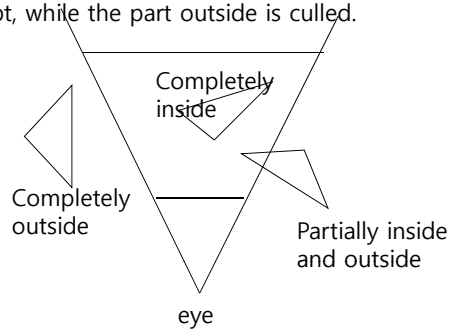


Backface Culling



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)

Viewport class members

```
AspectRatio;        // aspect radio
Bounds;             // size of this resource
MinDepth, MaxDepth;     // range of min, max depth values
TitleSafeArea;      // title safe area of the current viewport
Width, Height;      // width, height dimension of the viewport
X, Y;               // pixel coords of the upper-left corner
```

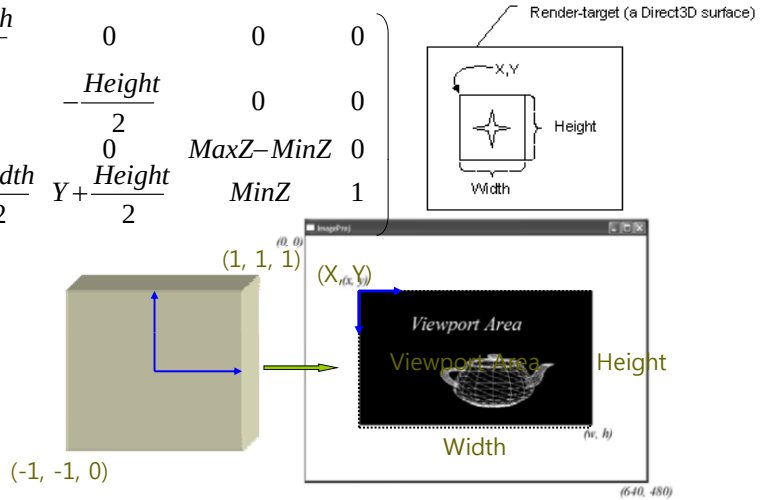
- Viewport matrix

```
Viewport vp(0, 0, 640, 480);
graphics.GraphicsDevice.Viewport = vp;
```

Viewport

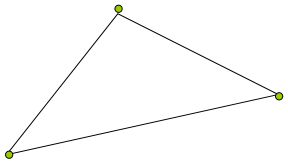
- Viewport Matrix

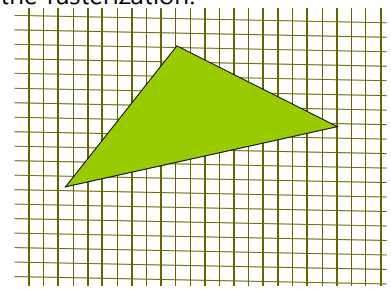
$$\begin{pmatrix} \frac{Width}{2} & 0 & 0 & 0 \\ 0 & -\frac{Height}{2} & 0 & 0 \\ 0 & 0 & MaxZ-MinZ & 0 \\ X+\frac{Width}{2} & Y+\frac{Height}{2} & MinZ & 1 \end{pmatrix}$$



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.





BasicEffect

- Using the basic effect class requires a set of world, view, and projection matrices, a vertex buffer, a vertex declaration, and an instance of the **BasicEffect** class.
- Initialize BasicEffect with transformation and light values

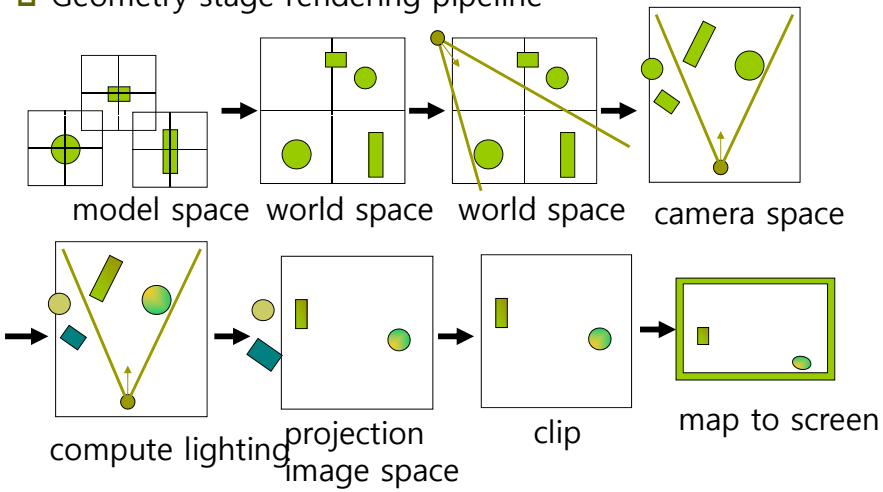
```
private BasicEffect effect;
// Initialize Effect
effect = new BasicEffect(graphics.GraphicsDevice);
// Draw
effect.World = world;
effect.Projection = projection;
effect.View = view;
effect.EnableDefaultLighting();
effect.TextureEnabled = true;
effect.Texture = texture;
```

BasicEffect

```
foreach (EffectPass pass in effect.CurrentTechnique.Passes)
{
    pass.Apply();
     graphics.GraphicsDevice.DrawUserIndexedPrimitives(
                PrimitiveType.TriangleList, vertices, 0, vertices.Length,
                indices, 0, indices.Length / 3);
}
```

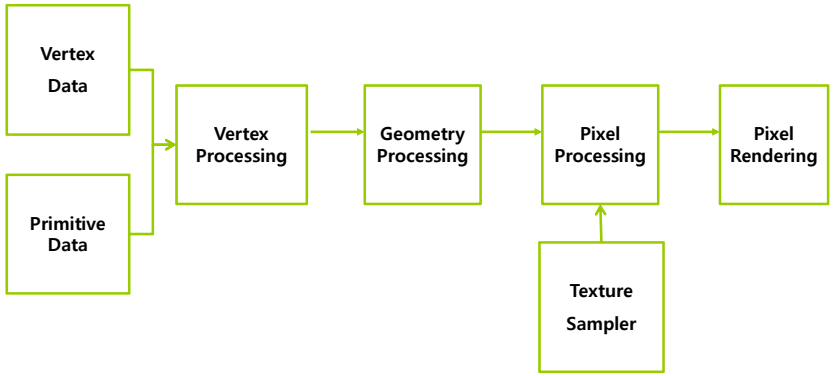
DirectX9 Rendering Pipeline

- Geometry stage rendering pipeline



XNA Rendering Pipeline

- The XNA Framework renders graphics by calling the DirectX9 rendering pipeline



XNA Rendering Pipeline

Pipeline Component	Description	Related Topics
Vertex Data	Vertex memory buffers provide storage for the untransformed model vertices. Typically, you can use a **VertexDeclaration** to describe what information (position, color, texture coordinates, normals, and so on) is defined for each vertex. Vertex buffers may contain either indexed or non-indexed vertex data.	**VertexBuffer, VertexDeclaration**
Primitive Data	Geometric primitives, including points, lines, triangles, and polygons, are referenced in the vertex data with index buffers. If a vertex buffer is not indexed, all of the vertices are placed in the vertex buffer in the order they are to be rendered. Because 3D-line lists or triangle lists often reference the same vertices multiple times, this can result in a large amount of redundant data. **Index buffers** allow you to list each vertex only once in the vertex buffer. An index buffer is a list of indices in the vertex buffer, given in the order that you want the vertices to render.	**IndexBuffer**

XNA Rendering Pipeline

Pipeline Component	Description	Related Topics
Vertex Processing	**The vertex shader of an Effect** transforms the vertices stored in the vertex buffer. You can use the world, view, and projection matrices defined for the **Effect** by the game to transform the vertices.	**Effect, BasicEffect, HLSL Shaders**
Geometry Processing	**Clipping, back face culling, attribute evaluation, and rasterization** are applied to the transformed vertices. Clipping is the process of removing triangles (or parts of triangles) that do not appear on screen (or into the scissor rectangle if scissor testing is enabled). Back face culling removes triangles that are not facing the camera. Rasterization is the process of assigning pixels to each triangle that remains on screen after the clipping and culling is complete.	**ScissorTestEnable, Render Targets, CullMode, Rasterization Rules**

XNA Rendering Pipeline

Pipeline Component	Description	Related Topics
Texture Sampler	Texture level-of-detail filtering is applied to textures that will be used by the pixel shader component of an Effect. This includes the **TextureAddressMode** of each texture coordinate, and the **TextureFilter**, to use in resizing the texture to fit the object on screen.	**SamplerState**
Pixel Processing	**The pixel shader of an Effect** uses geometry data to combine input vertex and texture data with lighting equations, which in turn yields the output pixel color values.	**Effect, BasicEffect, HLSL Shaders**
Pixel Rendering	Final rendering processes **modify pixel color values** with alpha, depth, or stencil testing, or by applying alpha blending. All resulting pixel values are presented to the output display.	**AlphaTestEnable, DepthBufferEnable, StencilEnable, AlphaBlendEnable**

Reference

- Direct3D Transformation Pipeline - http://msdn2.microsoft.com/en-us/library/bb206260.aspx
- XNA Rendering Pipeline http://msdn.microsoft.com/en-us/library/dd904179(v=xnagamestudio.31).aspx
- XNA BasicEffect class http://msdn.microsoft.com/en-us/library/bb203926.aspx