3D Pipeline

Logistics

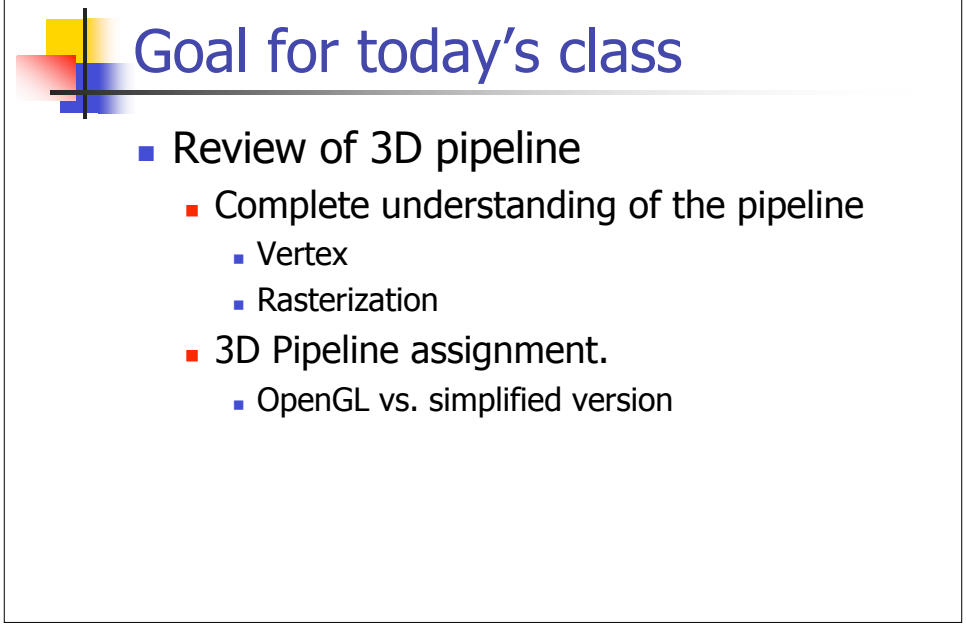
- Updates on assignments and such:
  - Assignment 2: 2D Algorithms
    - Most graded
  - Assignment 3: OpenGL 3D/Animation
    - Grading begun
  - Assignment 4: OpenGL Realism
    - Due Wednesday, November 7th
  - Midterm Pipeline Implementation
    - ALL GRADED
  - Homework 2: Transformations
    - ALL GRADED
  - Homework 3: Perspective
    - Past due / not graded
- All submissions via mycourses.

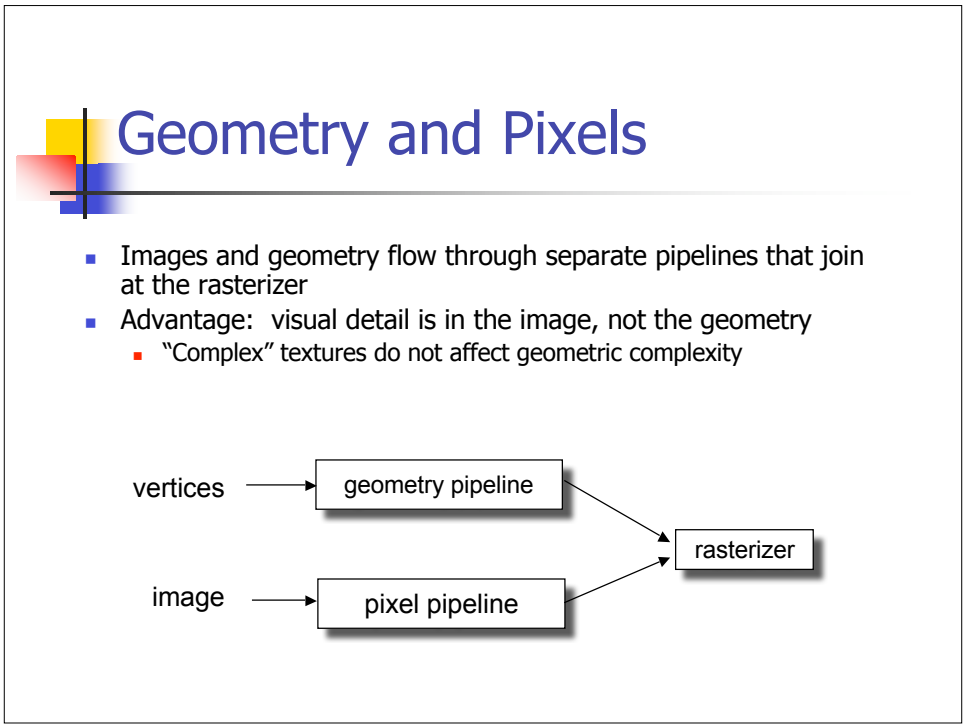
Logistics

- Final exam:
  - Wednesday, Nov 14
  - During class -- week 11
  - Brief review -- Monday Nov 12

Plan

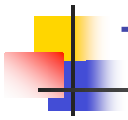
- Small change in plans
  - This Week
    - 3D Pipeline assignment
    - Global Illumination
  - Next Week: Week 10
    - Modeling


Goal for today's class
Review of 3D pipeline
Complete understanding of the pipeline
Vertex
Rasterization
3D Pipeline assignment.
OpenGL vs. simplified version


Geometry and Pixels
Images and geometry flow through separate pipelines that join at the rasterizer
Advantage: visual detail is in the image, not the geometry
"Complex" textures do not affect geometric complexity
vertices
geometry pipeline
image
pixel pipeline
rasterizer

Vertex Pipeline

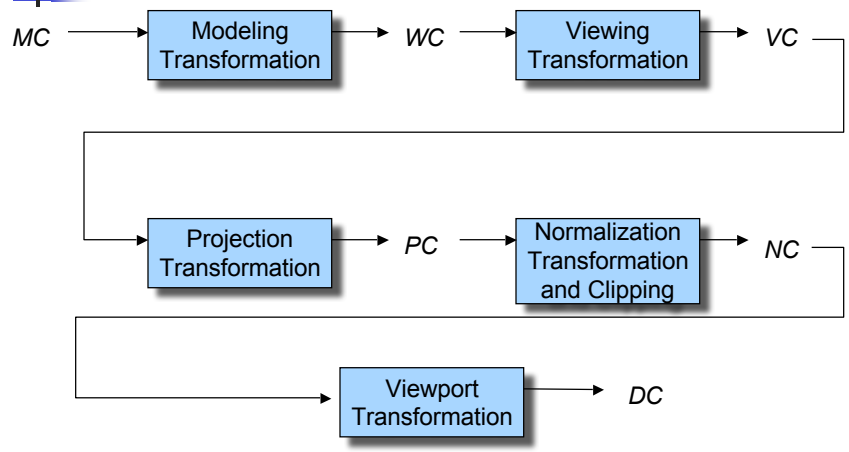
- Operates on polygon vertices
  - Transform, transform, transform
  - The composite matrix.
  - Eventually ends in transformation to 2D device space (integer -- discrete)



The composite matrix

- Single 4x4 Matrix that will take you from model to device.
- In OpenGL
  - Set up before defining polygon vertices
  - Each vertex will get transformed by "current" composite.

Graphics Vertex Pipeline

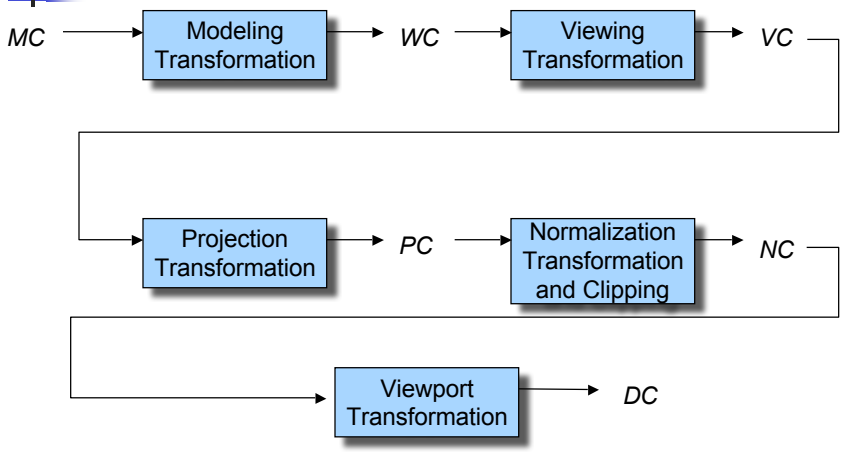


Model Coordinates

- Coordinates system in which the polygon is defined.

```
glBegin(GL_TRIANGLES);
   glVertex3f(-1.0f, -0.5f, -4.0f);
   glVertex3f( 1.0f, -0.5f, -4.0f);
   glVertex3f( 0.0f,  0.5f, -4.0f);
glEnd();
```

Graphics Vertex Pipeline



World Coordinates

- Coordinate system of your world.
  - Transformations: Translation, Rotation, Scaling.

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

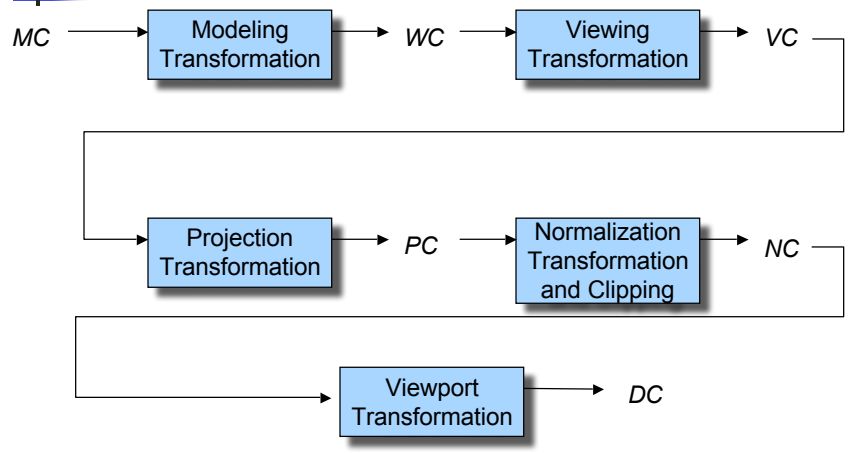
translation scaling Rotate x Rotate y Rotate z

World Coordinates

- Defining transformations

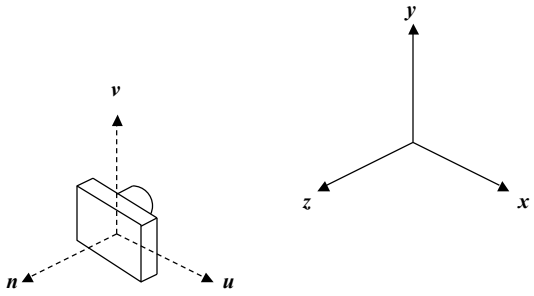
```
glTranslatef( tx, ty, tz )
glScalef( sx, sy, sz )
glRotatef( angle, x, y, z )
```

Graphics Vertex Pipeline



View Coordinates

- Sometime called camera coordinates
- World as seen from pov of camera



View transform

- The *eyepoint*, *lookat*, and the *up* vector define the *viewing-reference coordinate* system, also known as camera or eye coordinates.
- The default camera orientation has the camera/viewing-reference/eye coordinate system ***coincident*** with the world axes, i.e.,
  - `Eyepoint` at (0, 0, 0)
  - `Lookat` (0, 0, -1)
  - `Up` vector (0, 1, 0) anchored at *eyepoint*
  - Then
    - $n = eyepoint - lookat$ is (0, 0, 1) (normalized*)
    - $u = up \times n$ is (1, 0, 0) (normalized)
    - $v = n \times u$ is (0, 1, 0)
    - *normalized means a vector of length 1

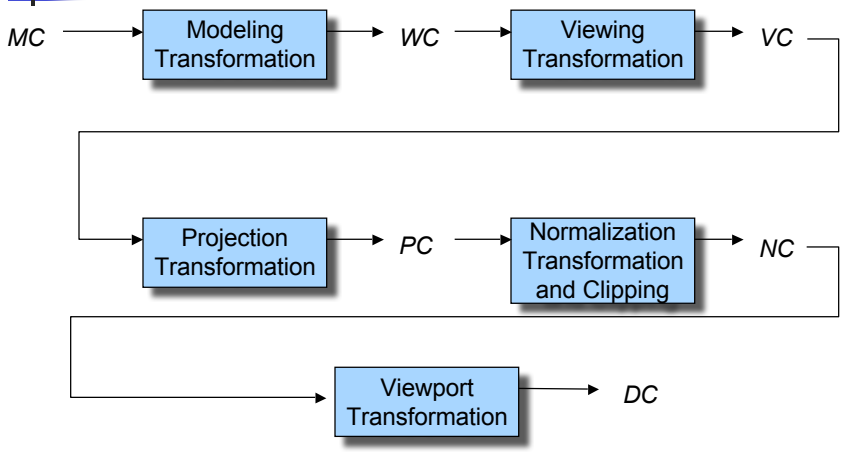
```
gluLookAt( eye.x, eye.y, eye.z, lookat.x,
  lookat.y, lookat.z, up.x, up.y, up.z )
```

World to Camera Coordinate Transformation

- So, we can do something similar in 3D
  - Build your own set of world and camera coordinates
  - Come up with the four transformations needed to make the axes coincide
  - These are the same four transformations that will be applied to all vertices in the world to get them in terms of camera coordinates
  - The composite of these is equivalent to

$$\begin{pmatrix} u_x & u_y & u_z & -\mathbf{u}\bullet\text{eyepoint} \\ v_x & v_y & v_z & -\mathbf{v}\bullet\text{eyepoint} \\ n_x & n_y & n_z & -\mathbf{n}\bullet\text{eyepoint} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

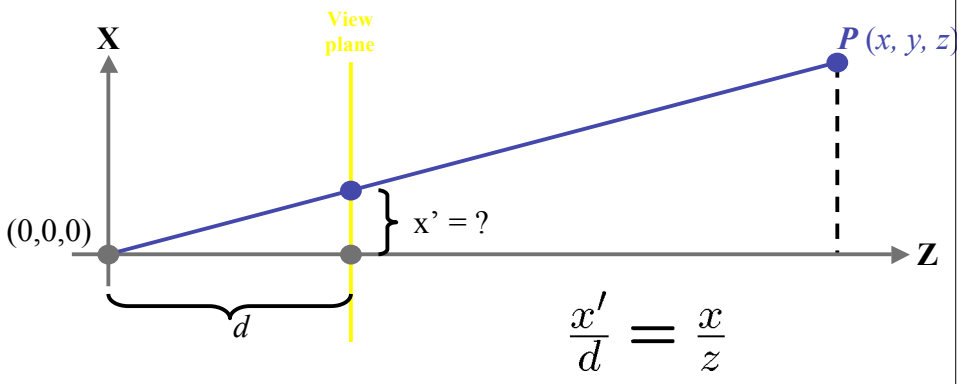
Graphics Vertex Pipeline



Projection

- Projecting 3D points onto "film plane"
  - Orthographic
  - Perspective

Perspective Projection



$$\frac{x'}{d} = \frac{x}{z}$$

Perspective Projection

$$M_{perspective} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix}$$

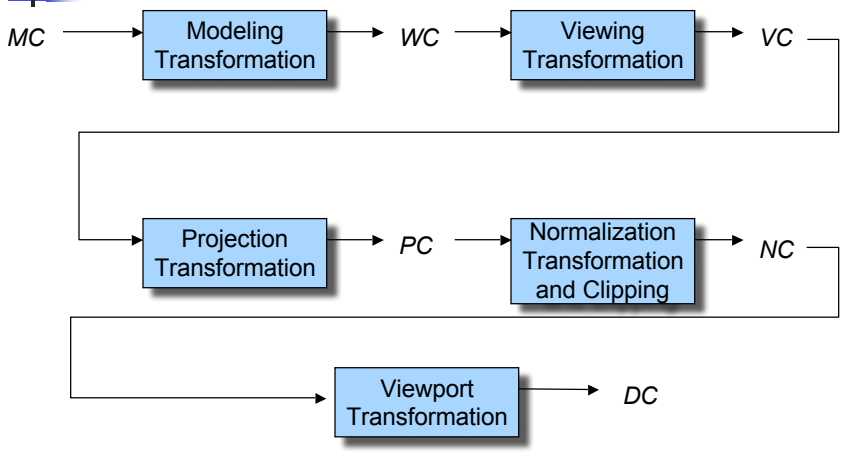
```
glFrustum(left, right, bottom, top, near, far)
```

Orthographic Projection

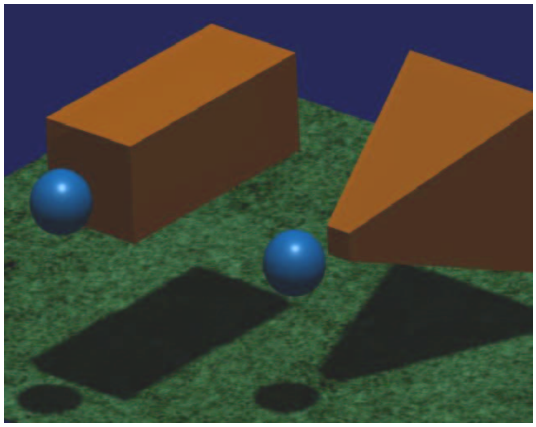
$$\begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

```
glOrtho(left, right, bottom, top, near, far)
```

Graphics Vertex Pipeline



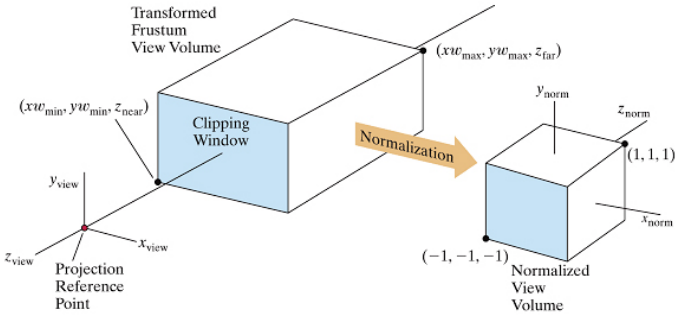
View Volumes



Orthographic Perspective/Frustum

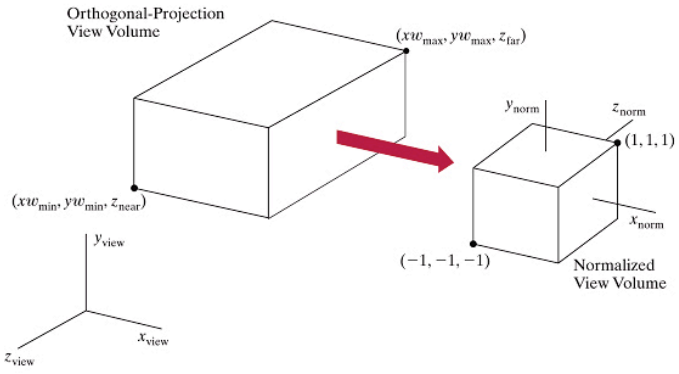
Frustum Normalization in OpenGL

Remember that all projections in OpenGL map the view volume to a 2 X 2 X 2 cube.



Orthographic Normalization in OpenGL

All projects in OpenGL map the view volume to a 2 X 2 X 2 cube.



Normalization

- Converts points in view volume to a standard canonical space.

Normalization - Ortho

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{2}{right-left} & 0 & 0 & \frac{-(right+left)}{right-left} \\ 0 & \frac{2}{bottom-top} & 0 & \frac{-(bottom+top)}{bottom-top} \\ 0 & 0 & \frac{2}{far-near} & \frac{-(far+near)}{far-near} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

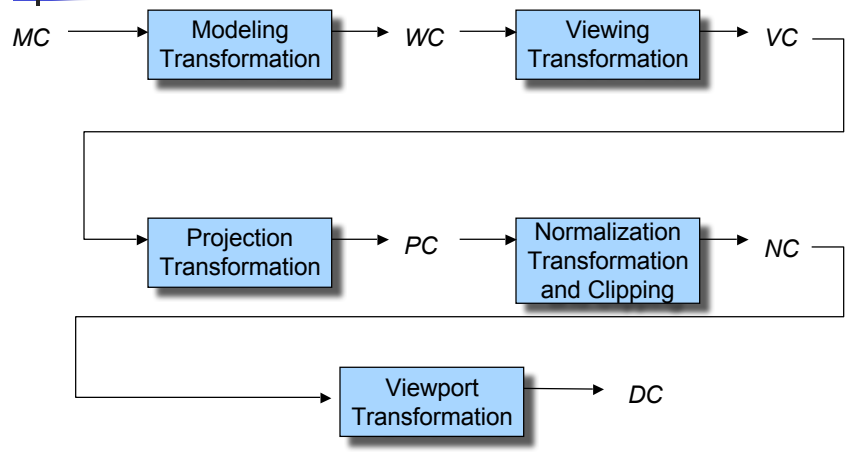
```
glOrtho(left, right, bottom, top, near, far)
```

Normalization - Perspective

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{2\times near}{right-left} & 0 & \frac{-(right+left)}{right-left} & 0 \\ 0 & \frac{2\times near}{bottom-top} & \frac{-(bottom+top)}{bottom-top} & 0 \\ 0 & 0 & \frac{far+near}{far-near} & \frac{-2\times far\times near}{far-near} \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

```
glFrustum(left, right, bottom, top, near, far)
```

Graphics Vertex Pipeline



Device (Screen) coords

- Viewport transform
  - Transforms points from canonical space to points on screen

Viewport Transform

- The normalized device coordinates (x,y),
  - $-1 \le x \le +1$ and $-1 \le y \le +1$, are converted into pixel coordinates $left \le x_w \le left+width$ and $bottom \le y_w \le bottom+height$.
  - The z coordinate $-1 \le z \le +1$ is scaled in the range $0 \le z_w \le 1$.

Viewport Transformation

- The sequence:

$$T(u_{min}, v_{min}) \cdot S(s_x, s_y) \cdot T(-x_{min}, -y_{min})$$

- The matrices:

$$= \begin{bmatrix} 1 & 0 & xv_{\min} \\ 0 & 1 & yv_{\min} \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \frac{xv_{\max} - xv_{\min}}{xw_{\max} - xw_{\min}} & 0 & 0 \\ 0 & \frac{yv_{\max} - yv_{\min}}{yw_{\max} - yw_{\min}} & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -xw_{\min} \\ 0 & 1 & -yw_{\min} \\ 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} \frac{xv_{\max} - xv_{\min}}{xw_{\max} - xw_{\min}} & 0 & -xw_{\min} \times \frac{xv_{\max} - xv_{\min}}{xw_{\max} - xw_{\min}} + xv_{\min} \\ 0 & \frac{yv_{\max} - yv_{\min}}{yw_{\max} - yw_{\min}} & -yw_{\min} \times \frac{yv_{\max} - yv_{\min}}{yw_{\max} - yw_{\min}} + yv_{\min} \\ 0 & 0 & 1 \end{bmatrix}$$

Viewport Transform

$$x_w = left + width \cdot \frac{x+1}{2}$$

$$y_w = bottom + height \cdot \frac{y+1}{2}$$

$$z_w = \frac{z+1}{2}$$

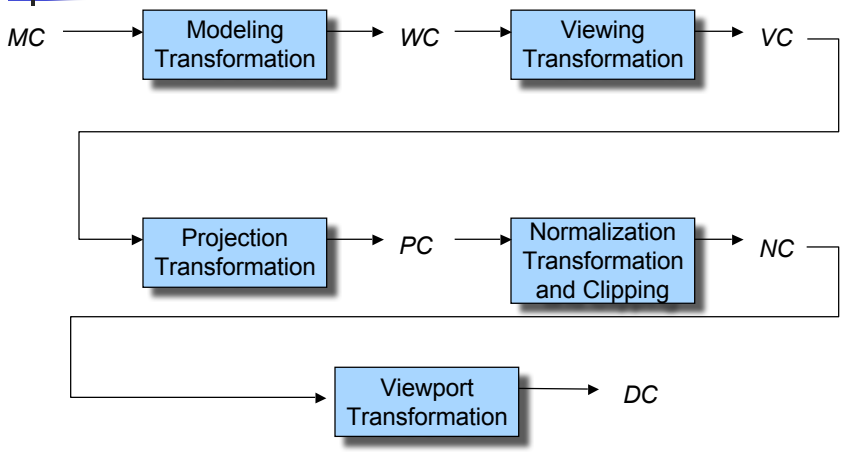
```
glViewport( left, bottom, width, height )
```

Viewport Transformation

- Note that this is a 3D matrix
  - To get from 4D to 3D, perform the homogeneous transform:

$$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \Rightarrow \begin{bmatrix} x/w \\ y/w \\ z/w \end{bmatrix}$$

Graphics Vertex Pipeline



Watch the order in which matricies are applied

At this point…

- You still only have vertices
- All vertices have been converted to screen coordinates.
- You are ready for rasterization.

- Questions?

Rasterization

- Convert vertices to polygons
- Issues:
  - Need to fill interior points
  - What color to shade?
  - Which polygon to draw at a given pixel?

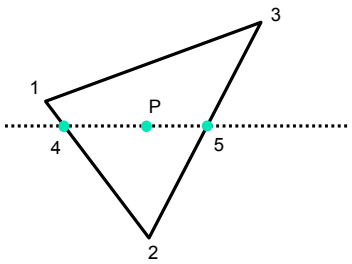
- We will modify the basic scan line algorithm.
- But first, we'll need more info at each of the vertices.

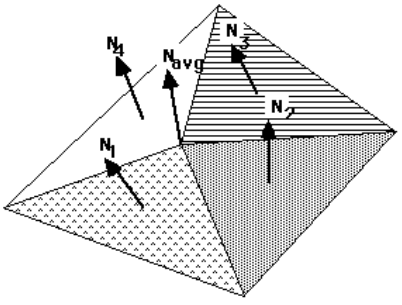
Shading Models

- Gouraud Shading
  - Illumination is interpolated across each polygon
  - Normals required at each polygon vertex, calculated as an average of the normals of each face that shares that vertex
  - Illumination is calculated for each polygon vertex
  - Interior points interpolated from endpoint illumination intensities

Shading Models

- Gouraud Shading – interpolating normals





Shading Model

- Gouraud Shading

  Need to maintain color at each vertex

Hwang

Calculating color

- Lighting
- Material properties
- Textures
- More info to maintain at each vertex
  - Material properties
  - Normals
  - Texture coords

Which polygon to render

- From Wikipedia:
  - *hidden surface determination* is the process used to determine *which surfaces and parts of surfaces* are *not visible* from a *certain viewpoint*.
  - A hidden surface determination algorithm is a solution to the *visibility problem*
  - the core differences between most rendering algorithms is how they handle this problem.

Stages of Hidden Surface Determinations

- Viewing Fustrum Culling
  - Remove objects not in view volume
  - I.e Clipping
- Backface culling
  - Remove faces that do not face the camera
- Occlusion Culling
  - Determines which portions of objects are hidden by other objects from a given viewpoint.
- Contribution Culling
  - Objects whose screen projection are so small are thrown away.

Depth (Z) Buffer Algorithm

- Requires two arrays the size of the screen
  - Depth – initialized as "far away"
  - Color – initialized to background color
- Used in OpenGL

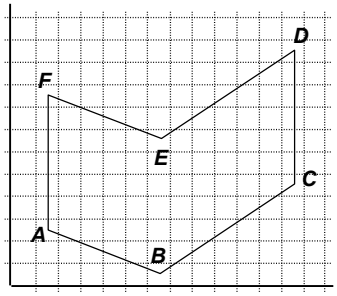
```
For each polygon,
    For each y,
        For each x ,
            If z < depth[x][y],
                Depth[x][y] = z
                Color[x][y] = polygon color
```

Basic scan line algorithm

- Must do this for all edges and all scan lines, not just one edge
- Can generalize this to handle arbitrarily complex polygons using an *edge table*
- The ET contains all edges, sorted by y_{min}
- We move edges from ET into an *active edge table* as we encounter them
- Note: our text calls AET the active edge list (AEL)

Building the Edge Table

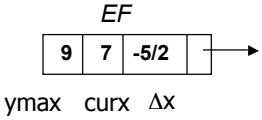
- Our figure's edges:
  - *AB, BC, CD, DE, EF, FA*
- We create a bucket list for each scan line which indicates which edge(s) intersect that scan line
- Each edge is listed only the *first time* is is crossed
  - E.g., *FA* will appear for scan line 3, but not for lines 4-9
- Bucket chains are sorted by increasing x of the lower endpoint
  - What if multiple buckets have the same x endpoint?



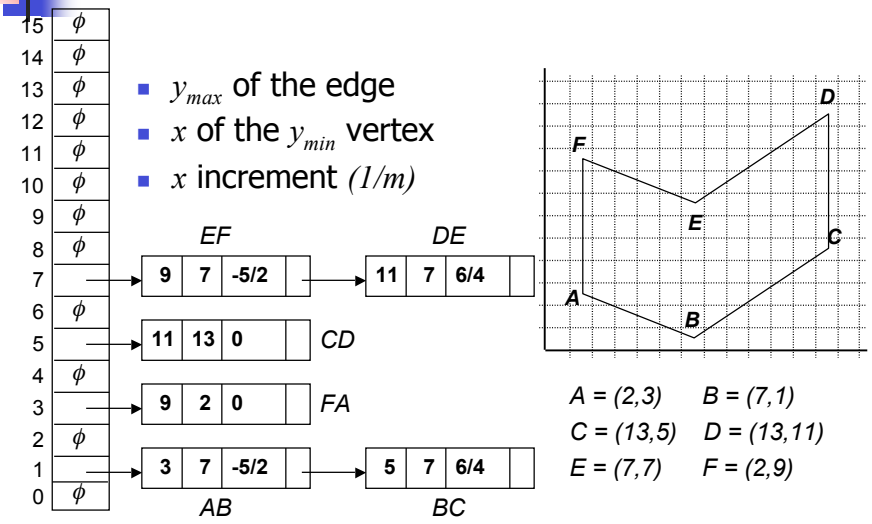
A = (2,3) *B = (7,1)*
C = (13,5) *D = (13,11)*
E = (7,7) *F = (2,9)*

Edge Table Buckets

- Each bucket contains the following things:
  - y_{max} of the edge
  - Current x of the edge
    - Initially, x of the vertex with the y_{min} coordinate
  - The x increment $(1/m)$
  - A link to the next bucket
- We build the ET as an array of pointers to buckets, indexed by scan line number



Edge Table Example



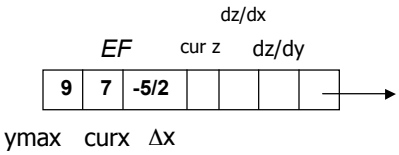
- y_{max} of the edge
- x of the y_{min} vertex
- x increment *(1/m)*

Fill algorithm

- Form the ET
- Initialize AET to empty
- Set y to the smallest y in the ET which has buckets
- Repeat until both ET and AET are empty:
  - Remove AET entries where $y = y_{max}$
  - Move from ET[y] to AET when $y_{min} = y$
  - Sort AET on x
  - Fill pixels on scan line y using pairs of x coords from AET
  - Increment y
  - For each non-vertical edge in AET, update x for new y
    - How to detect non-vertical edge?
    - Look at inverse slope - if *1/m != 0*, have non-vertical edge

Adding to the basic scan line algorithm

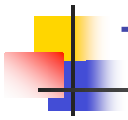
- Add other values to be interpolated to the edge table buckets
  - E.g.
    - Z (depth) values
    - Colors
    - Texture coords



Questions?

The FINAL Option

- For the daring…
  - Build a 3D pipeline implementing the vertex and rasterization pipeline presented today.

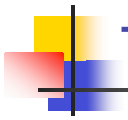


The FINAL Option

- Vertex additions
  - 3D transforms
  - Projections (Ortho and Frustrum)
  - 3D Camera Model
  - Perspective divide
  - Storage of depth and color at vertices
- CLIPPING in 2D

The FINAL Option

- Rasterization Additions
  - Basic Z-Buffer algorithm
  - Z value and color interpolation in scan line algorithm.
  - Gouraud Shading
- NO PHONG Illumination
- NO TEXTURE Mapping



The FINAL Option

- Your own implementation of select OpenGL / Glu routines.
- Use modified 2D drawing routines from Midterm / assignment 2
- Only allowed to use setPixel().

The FINAL Option

- List of functions to implement with complete write-up (and test program) by Wednesday.
- Due date:
  - Wednesday, November 14th.
- Questions?