Cs 248

Transformations and Coordinate Systems

With some slides stolen from Kurt Akeley 2007

Overview

- Transformations
  - Affine
  - Projection
  - Viewport
- Coordinate Systems
- Hidden Surface Removal
- Rasterization
  - Barycentric coordinates
  - Sampling & antialiasing
- Interpolation
- Assignment 1 Q& A

Homogeneous 3D Coordinates

- $$\text{Represent point} \begin{pmatrix} x \\ y \\ z \end{pmatrix} \text{ as } \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

$$\text{Treat} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \text{ as equivalent to} \begin{pmatrix} wx \\ wy \\ wz \\ w \end{pmatrix}, w \neq 0$$

Affine 3D transformation

- Includes Translation, Rotation, Scale, and Shear

Rotation, Scale, Shear

Translation

$$v' = M\,v$$

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Affine Transforms

- Translation:

$$T(t_x, t_y, t_z) = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scaling (about the origin):

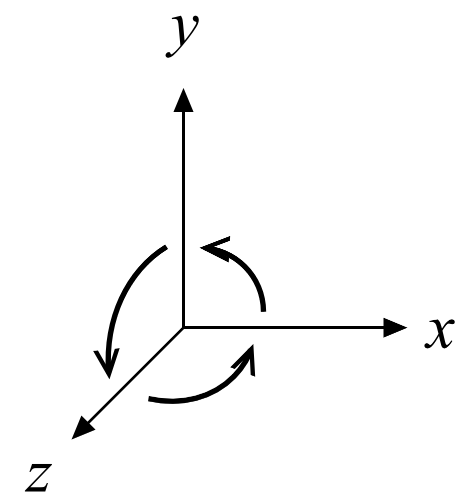
$$S(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotations

- $$R_x(q) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(q) & -\sin(q) & 0 \\ 0 & \sin(q) & \cos(q) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

  $$R_y(q) = \begin{bmatrix} \cos(q) & 0 & \sin(q) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(q) & 0 & \cos(q) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

  $$R_z(q) = \begin{bmatrix} \cos(q) & -\sin(q) & 0 & 0 \\ \sin(q) & \cos(q) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Right-hand rule for rotations by positive θ

$$\mathbf{v}' = R(\theta, \mathbf{a}) \cdot \mathbf{v}, \quad \text{where } R(\theta, \mathbf{a}) = \begin{pmatrix} & & & 0 \\ & \mathbf{R} & & 0 \\ & & & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\text{Let } \mathbf{b} = \frac{\mathbf{a}}{\|\mathbf{a}\|} = \begin{pmatrix} b_x \\ b_y \\ b_z \end{pmatrix}$$

$$\text{Then } \mathbf{R} = \mathbf{b} \cdot \mathbf{b}^T + \cos\theta \cdot \left(\mathbf{I} - \mathbf{b} \cdot \mathbf{b}^T \right) + \sin\theta \cdot \begin{pmatrix} 0 & -b_z & b_y \\ b_z & 0 & -b_x \\ -b_y & b_x & 0 \end{pmatrix}$$

How do I do this in OpenGL?

OpenGL matrix assignment commands

```
glLoadIdentity();
```

Remember to do this!

$$\mathbf{C}' = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

```
glLoadMatrixf(float m[16]);
glLoadMatrixd(double m[16]);
```

$$\mathbf{C}' = \begin{pmatrix} a_1 & a_5 & a_9 & a_{13} \\ a_2 & a_6 & a_{10} & a_{14} \\ a_3 & a_7 & a_{11} & a_{15} \\ a_4 & a_8 & a_{12} & a_{16} \end{pmatrix}$$

```
glLoadTransposeMatrixf(float m[16]);
glLoadTransposeMatrixd(double m[16]);
```

$$\mathbf{C}' = \begin{pmatrix} a_1 & a_2 & a_3 & a_4 \\ a_5 & a_6 & a_7 & a_8 \\ a_9 & a_{10} & a_{11} & a_{12} \\ a_{13} & a_{14} & a_{15} & a_{16} \end{pmatrix}$$

OpenGL matrix composition commands

```
glMultMatrixf(float m[16]);
glMultMatrixd(double m[16]);
```

$$\mathbf{C}' = \mathbf{C} \cdot \begin{pmatrix} a_1 & a_5 & a_9 & a_{13} \\ a_2 & a_6 & a_{10} & a_{14} \\ a_3 & a_7 & a_{11} & a_{15} \\ a_4 & a_8 & a_{12} & a_{16} \end{pmatrix}$$

```
glMultTransposeMatrixf(float m[16]);
glMultTransposeMatrixd(double m[16]);
```

$$\mathbf{C}' = \mathbf{C} \cdot \begin{pmatrix} a_1 & a_2 & a_3 & a_4 \\ a_5 & a_6 & a_7 & a_8 \\ a_9 & a_{10} & a_{11} & a_{12} \\ a_{13} & a_{14} & a_{15} & a_{16} \end{pmatrix}$$

OpenGL transformations commands

```
glRotatef(float θ, float x,
          float y, float z);

glRotated(double θ, double x,
          double y, double z);
```

$$\mathbf{C}' = \mathbf{C} \cdot \begin{pmatrix} & & & 0 \\ & \mathbf{R} & & 0 \\ & & & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

```
glTranslatef(float x, float y,
             float z);

glTranslated(double x, double y,
             double z);
```

$$\mathbf{C}' = \mathbf{C} \cdot \begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

```
glScalef(float x, float y, float z);

glScaled(double x, double y, double z);
```

$$\mathbf{C}' = \mathbf{C} \cdot \begin{pmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Why post-multiplication ?

Rotate θ degrees about axis $\mathbf{a} = \begin{pmatrix} a_x \\ a_y \\ a_z \end{pmatrix}$ through point $\mathbf{p} = \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix}$

$$\mathbf{v}' = T\left(p_x, p_y, p_z\right)\cdot\left(R\left(\theta, \mathbf{a}\right)\cdot\left(T\left(-p_x, -p_y, -p_z\right)\cdot\mathbf{v}\right)\right)$$

```
glTranslated(px, py, pz);
glRotated(θ, a);
glTranslated(-px, -py, -pz);
DrawObject();
```

Matrix Stacks

- OpenGL implements stacks to save Modelview and Projection matrices
  - Useful for hierarchical drawing
- glPushMatrix()
- glPopMatrix()
- Stacks are initialized with single identity matrix
  - It is an error to pop this matrix

Duality

Transform the coordinate system

Transform the object

```
glMultMatrixf(mat1);
glMultMatrixf(mat2);
glMultMatrixf(mat3);
DrawObject();
```

Rotation about a specified point

Transform the coordinate system

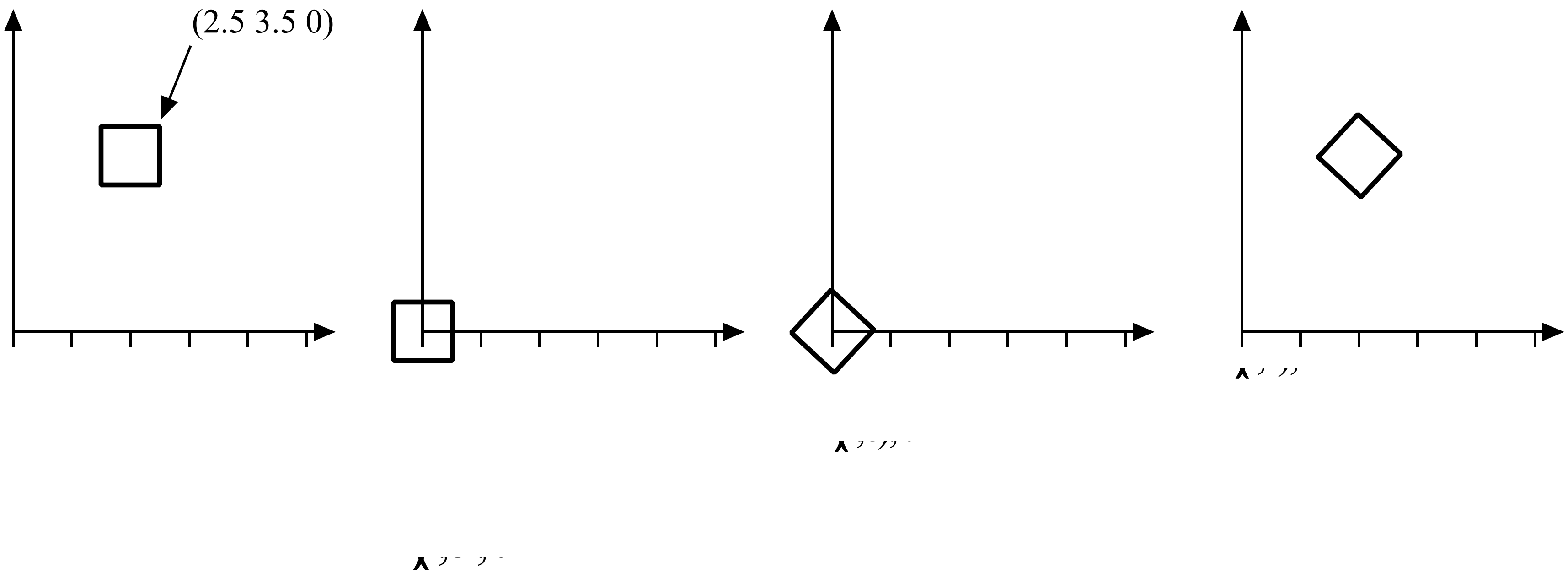
Transform the object

```
glTranslated(2, 3, 0);
glRotated(45, 0, 0, 1);
glTranslated(-2, -3, 0);
DrawObject();
```

Transform the object

Transformations are applied *opposite* the specified order

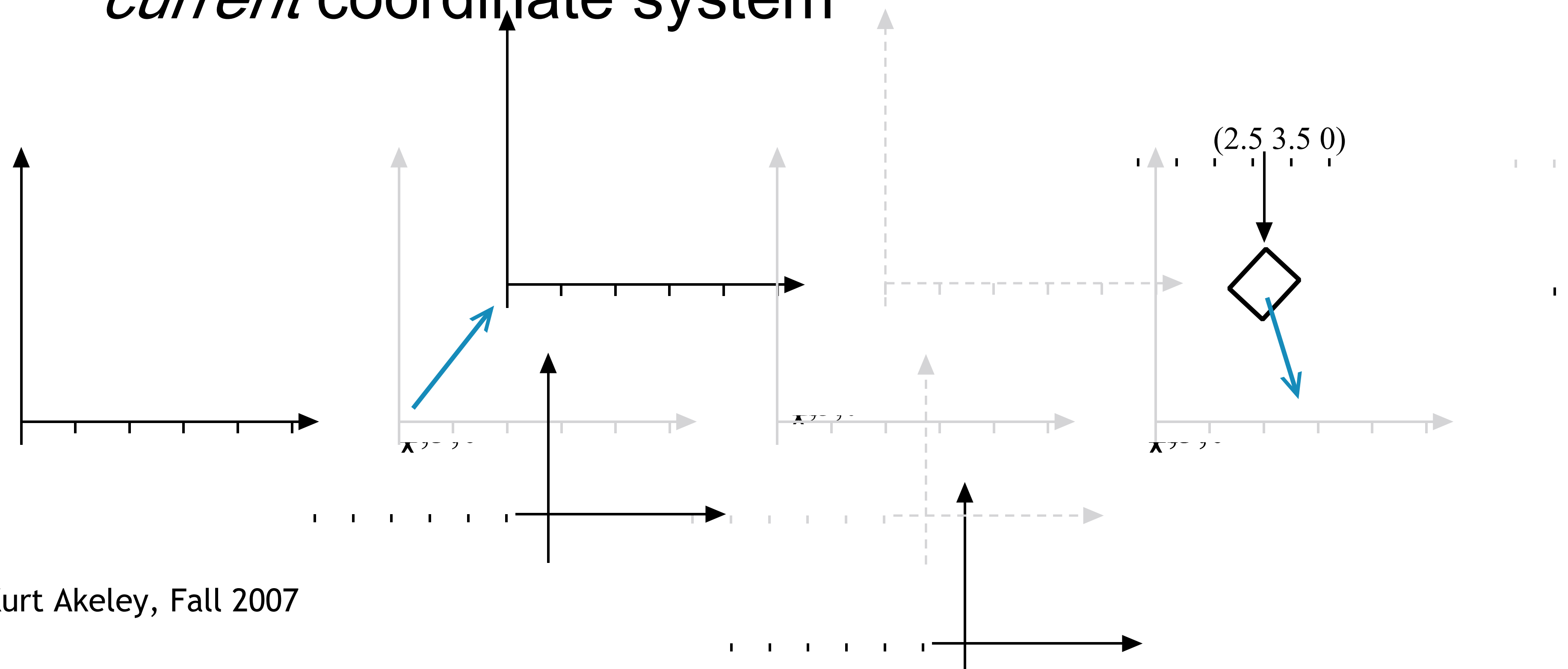
Each transformation operates in the fixed coordinate system



Transform the coordinate system

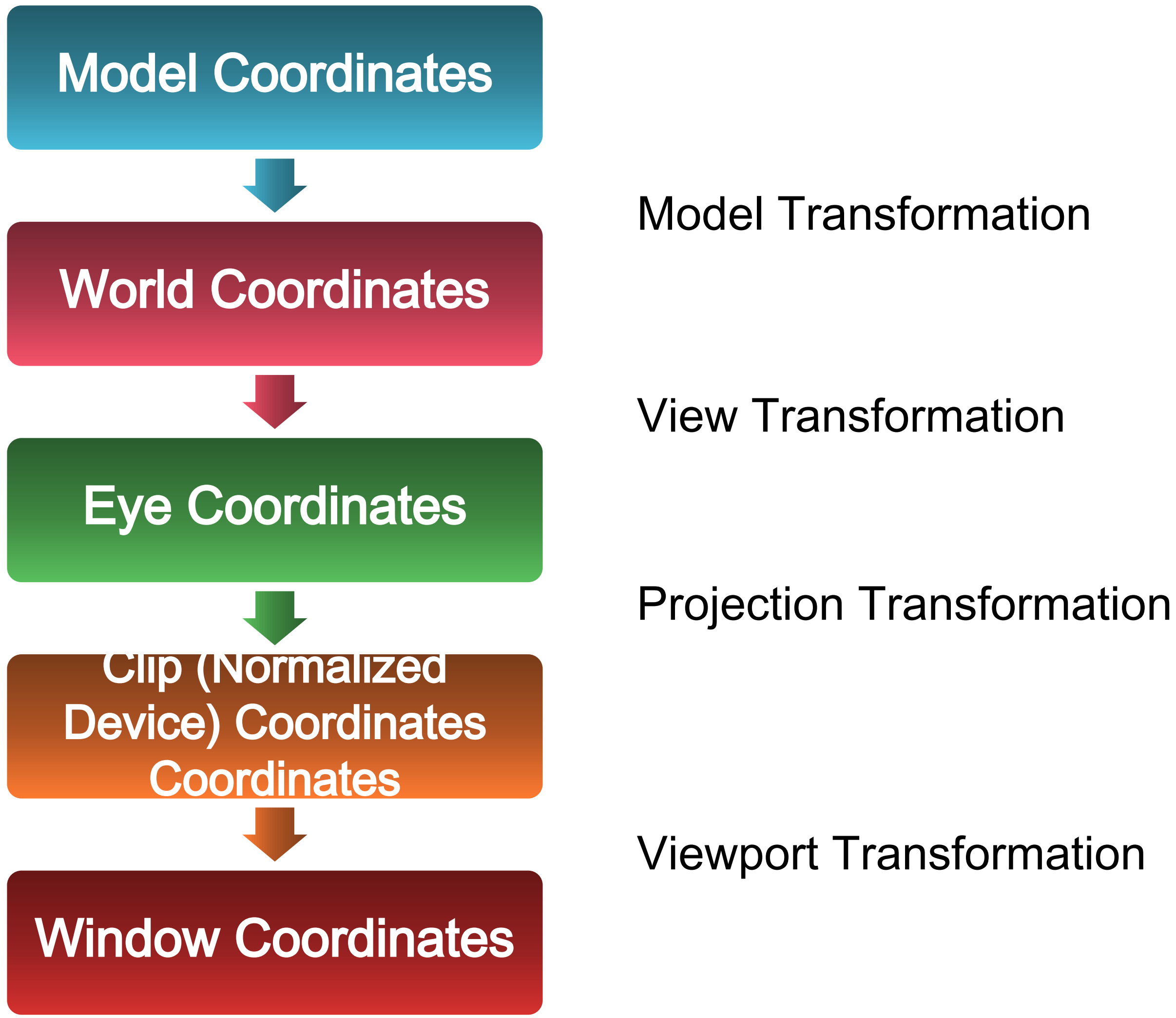
Transformations are applied in the order specified

Each transformation operates relative to the *current* coordinate system

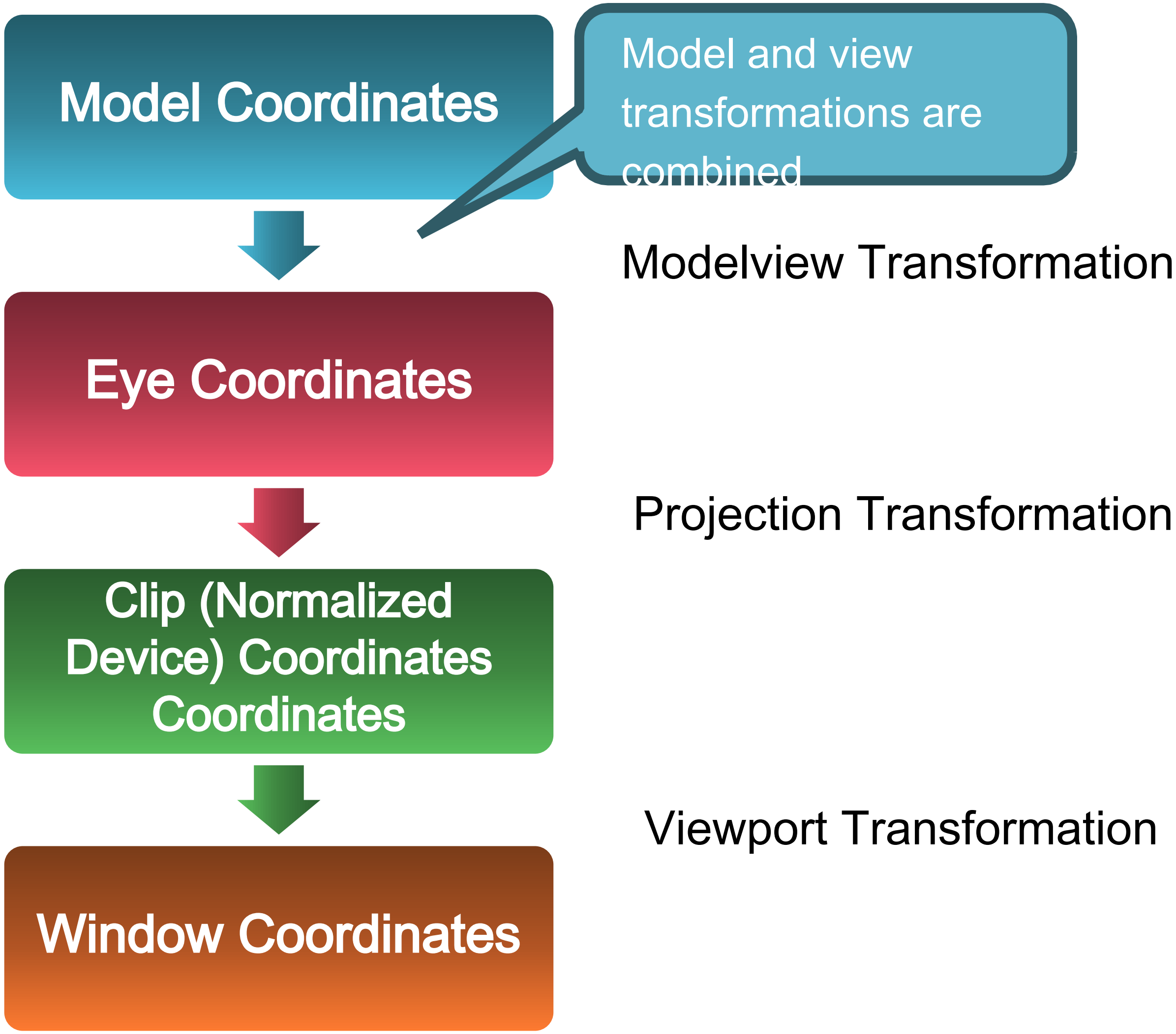


Coordinate Systems

Coordinate System Pipeline



OpenGL Coordinate System Pipeline

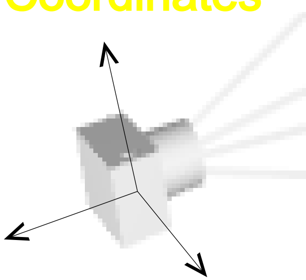


Coordinates Systems

Model Coordinates

Model Coordinates

Model Coordinates



Eye Coordinates

World Coordinates

Images courtesy http://www.songho.ca/opengl/gl_transform.html

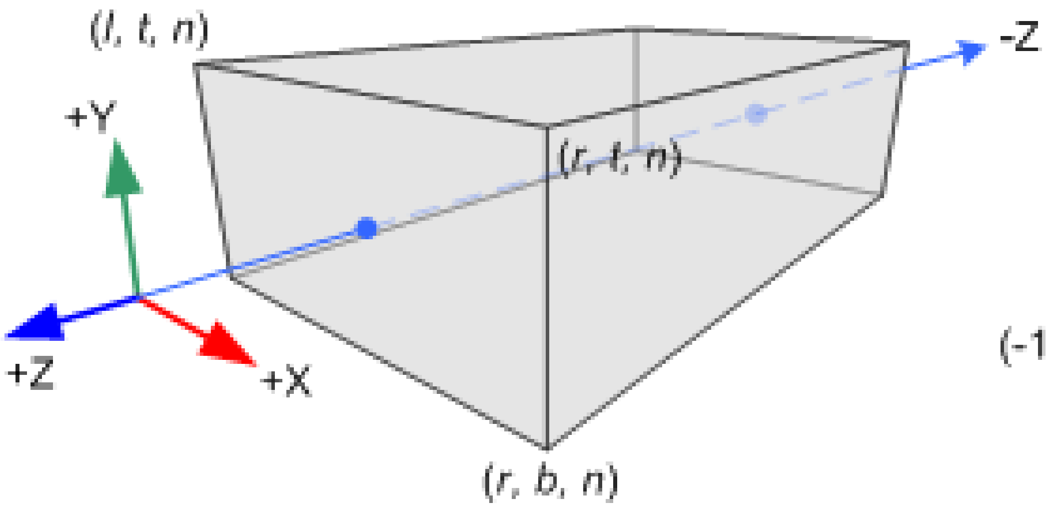
Model & Eye coordinates

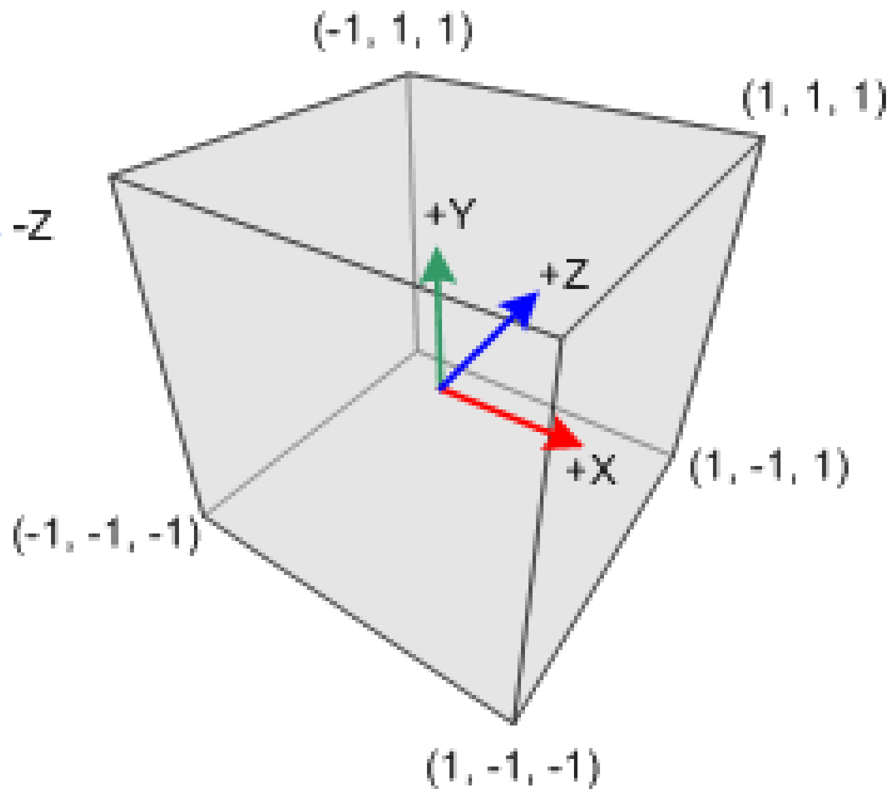
- Model coordinates
  - Defined by modeler
  - Specific to each program
- Eye Coordinates
  - Camera is at the origin
  - Camera looks down –Z axis

Projection

- Eye coordinates clip coordinates
- Projection transforms the entire viewing volume to the unit cube ([-1,1] x [-1,1] x [-1,1])
  - That is after projection, everything inside the unit cube is visible and should be rasterized
- This makes clipping calculations trivial
- Standard OpenGL projection matrices flip the z axis. This changes the coordinate system from a right-handed system to a

Orthographic Project





Images courtesy http://www.songho.ca/opengl/

Orthographic Projection

- Basically scales and translates view

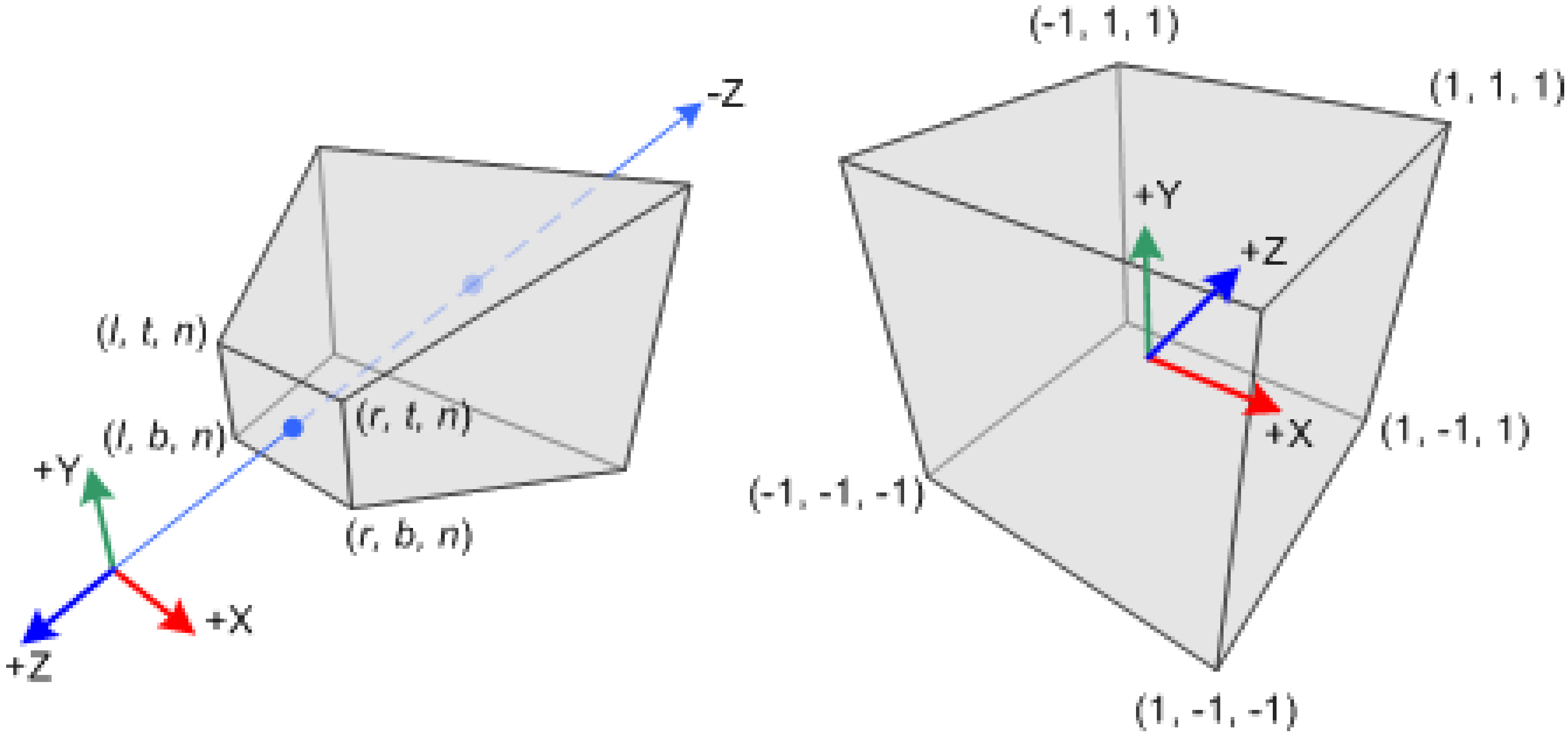
```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(l,r,b,t,n,f);
```

`n` and `f` are signed distances, not coordinates

$$\begin{pmatrix} x_c \\ y_c \\ z_c \\ w_c \end{pmatrix} = \mathbf{P} \cdot \begin{pmatrix} x_e \\ y_e \\ z_e \\ w_e \end{pmatrix}$$

$$\mathbf{P}' = \mathbf{P} \cdot \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Perspective Projection



Images courtesy http://www.songho.ca/opengl/

Perspective Projection

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glFrustum(l,r,b,t,n,f);
```

`n` and `f` are distances, not coordinates. Both must be positive.

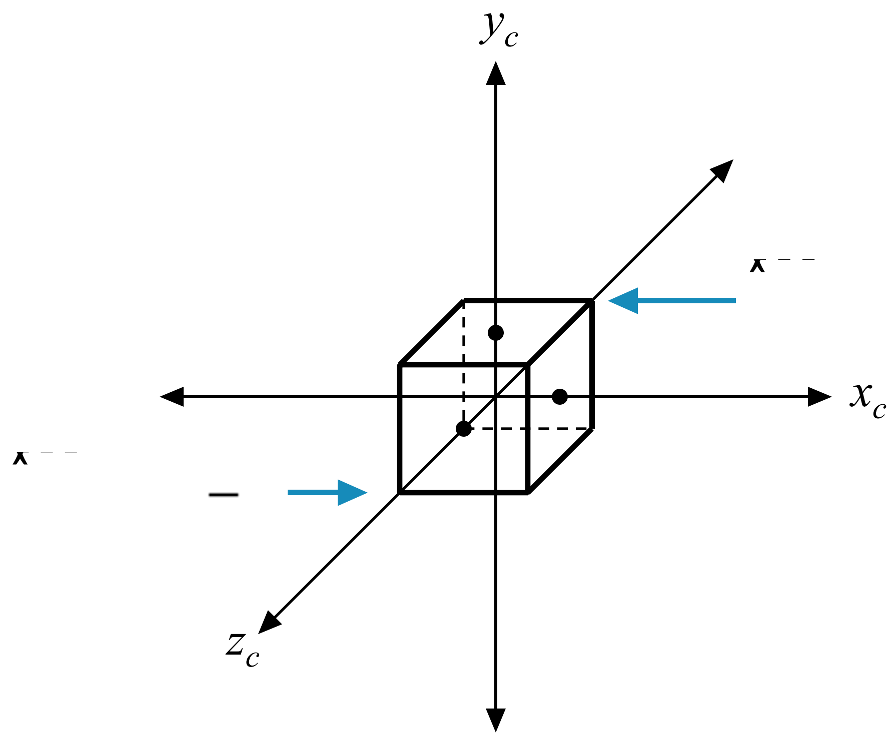
$$\begin{pmatrix} x_c \\ y_c \\ z_c \\ w_c \end{pmatrix} = \mathbf{P} \cdot \begin{pmatrix} x_e \\ y_e \\ z_e \\ w_e \end{pmatrix}$$

$$\mathbf{P}' = \mathbf{P} \cdot \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Clip testing

v_c is within the unit cube centered at the origin if:

$$-1 \leq \frac{x_c}{w_c} \leq 1$$
$$-1 \leq \frac{y_c}{w_c} \leq 1$$
$$-1 \leq \frac{z_c}{w_c} \leq 1$$



Homogenization

Divide by w_c**:**

$$\begin{pmatrix} x_d \\ y_d \\ z_d \\ 1 \end{pmatrix} = \begin{pmatrix} x_c / w_c \\ y_c / w_c \\ z_c / w_c \\ w_c / w_c \end{pmatrix}$$

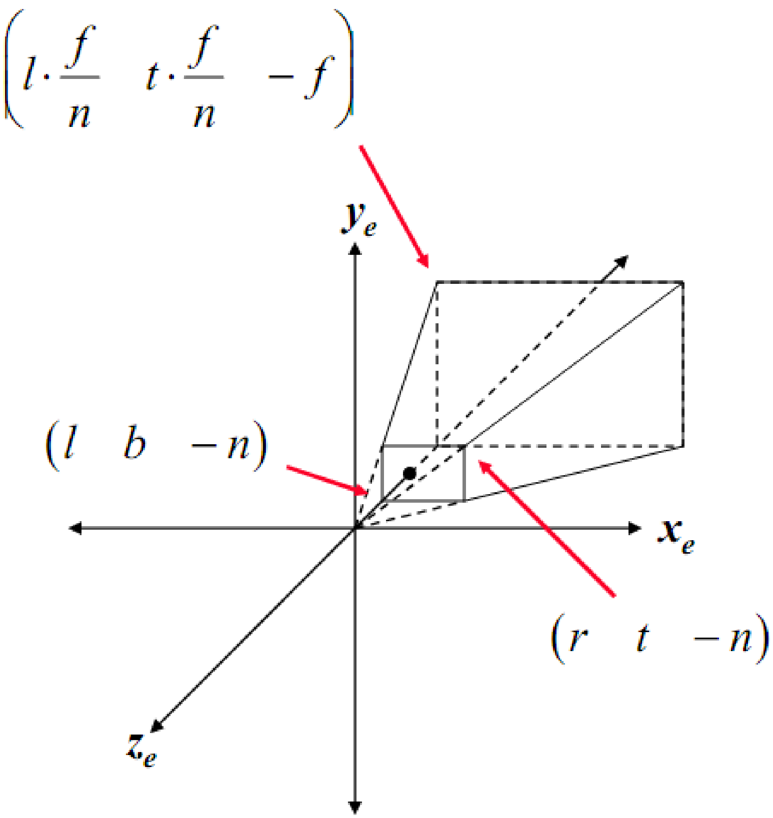
Discard w_d**:**

$$\mathbf{v}_d = \begin{pmatrix} x_d \\ y_d \\ z_d \end{pmatrix}$$

The Frustum

- $w_c = -z_e$
  $$z_c = -z_e \cdot \frac{f+n}{f-n} - w_e \cdot \frac{2fn}{f-n}$$
  $$= c \cdot z_e + d$$
  $$z_d = \frac{z_c}{w_c}$$
  $$= \frac{c \cdot z_e + d}{-z_e}$$
  $$= c + \frac{d}{-z_e^2}$$

As z_e gets larger, the same change in z_e results in a smaller changer in z_d



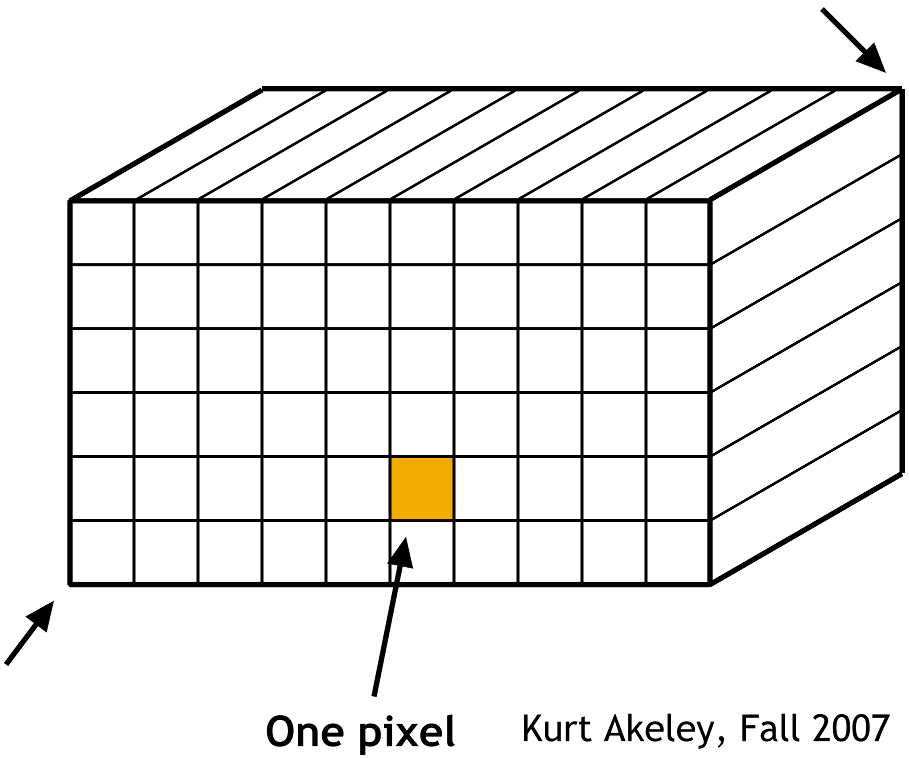
Viewport transformation

Transform the [-1,1] range device coordinates to window coordinates:

```
glViewport(int l, int b, int w, int h);
glDepthRange(double n, double f);
```

$$\begin{pmatrix} x_w \\ y_w \\ z_w \end{pmatrix} = \begin{pmatrix} \frac{w}{2} & 0 & 0 \\ 0 & \frac{h}{2} & 0 \\ 0 & 0 & \frac{f-n}{2} \end{pmatrix} \begin{pmatrix} x_d \\ y_d \\ z_d \end{pmatrix} + \begin{pmatrix} l + \frac{w}{2} \\ b + \frac{h}{2} \\ \frac{f+n}{2} \end{pmatrix}$$

```
glViewport(0, 0, 10, 6);
glDepthRange(0.0, 1.0);
```



Kurt Akeley, Fall 2007

Window coordinates

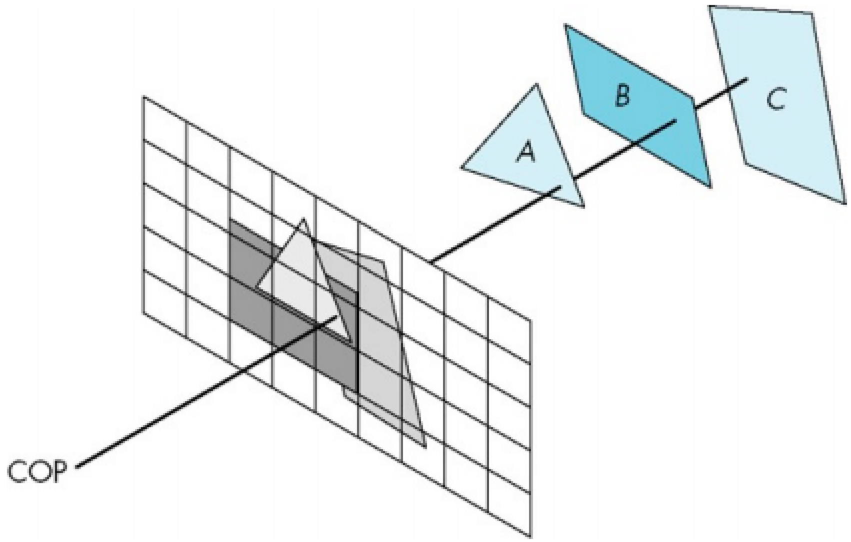
Window coordinates are:

- Continuous, not discrete
- Not homogeneous
  - Can be used directly for rasterization
- 3-D, not 2-D
  - Projection did not reduce 3-D to 2-D
  - 3-D homogeneous was reduced to 3-D non-homogeneous

Hidden surface removal

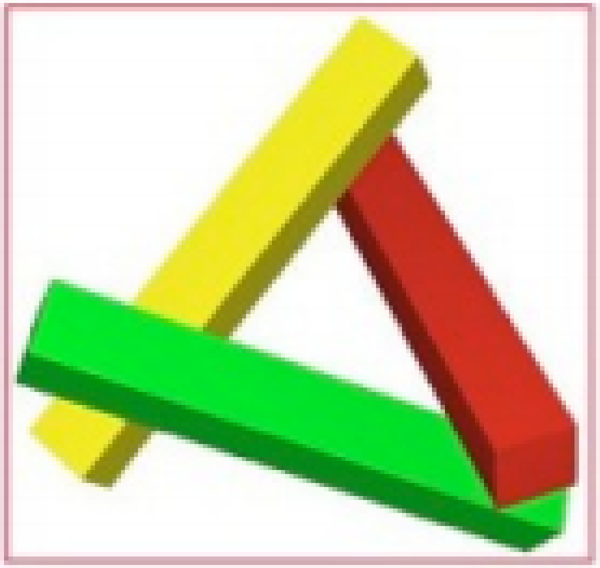
Hidden Surface Removal

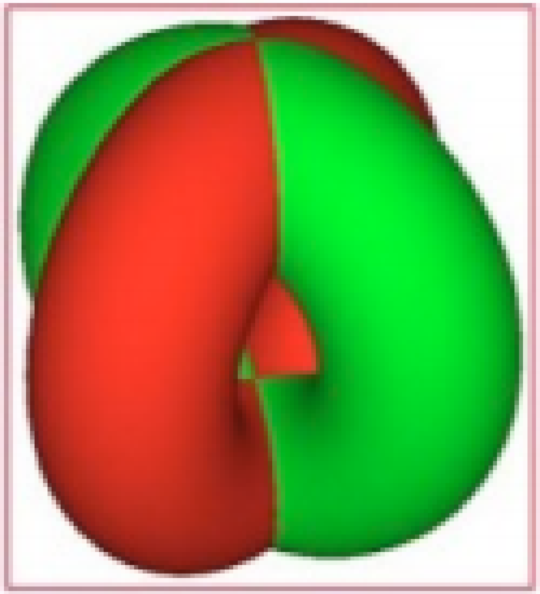
- Goal: to hide parts of objects that are not visible because they are covered ("occluded") by other objects



Painter's Algorithm

- Default behavior is to draw each primitive in the order it is specified, overwriting previous pixel
- To hide occluded geometry, we need to sort primitives and render back to front
- Not always possible. Incompatible with pipeline model





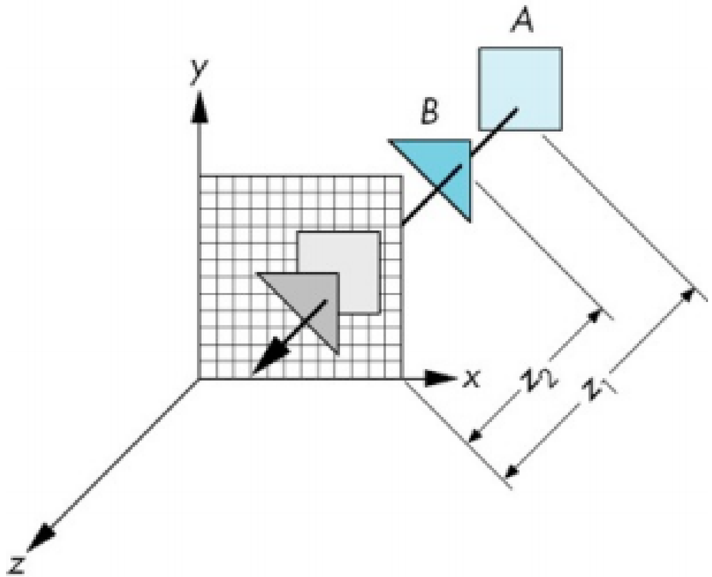
Z-buffering

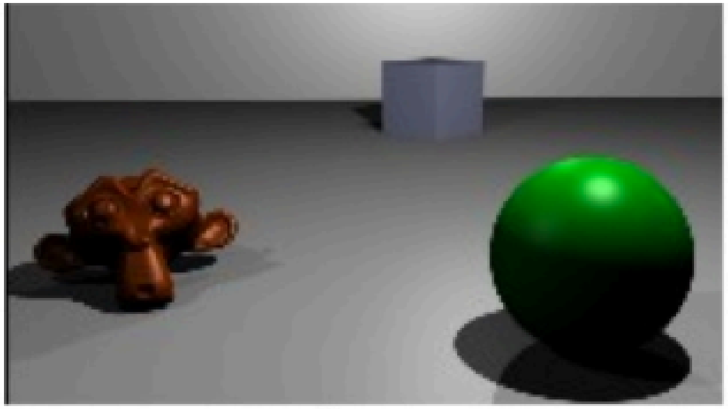
- Remember we kept the depth values of each vertex in screen coordinates
- Z-buffer algorithm:

  use a separate buffer to store depth values at each pixel

  Before filling in a pixel, check to see if the current fragment depth < stored value in z-buffer

Z-buffer





Scene

Depth Buffer

Rasterization

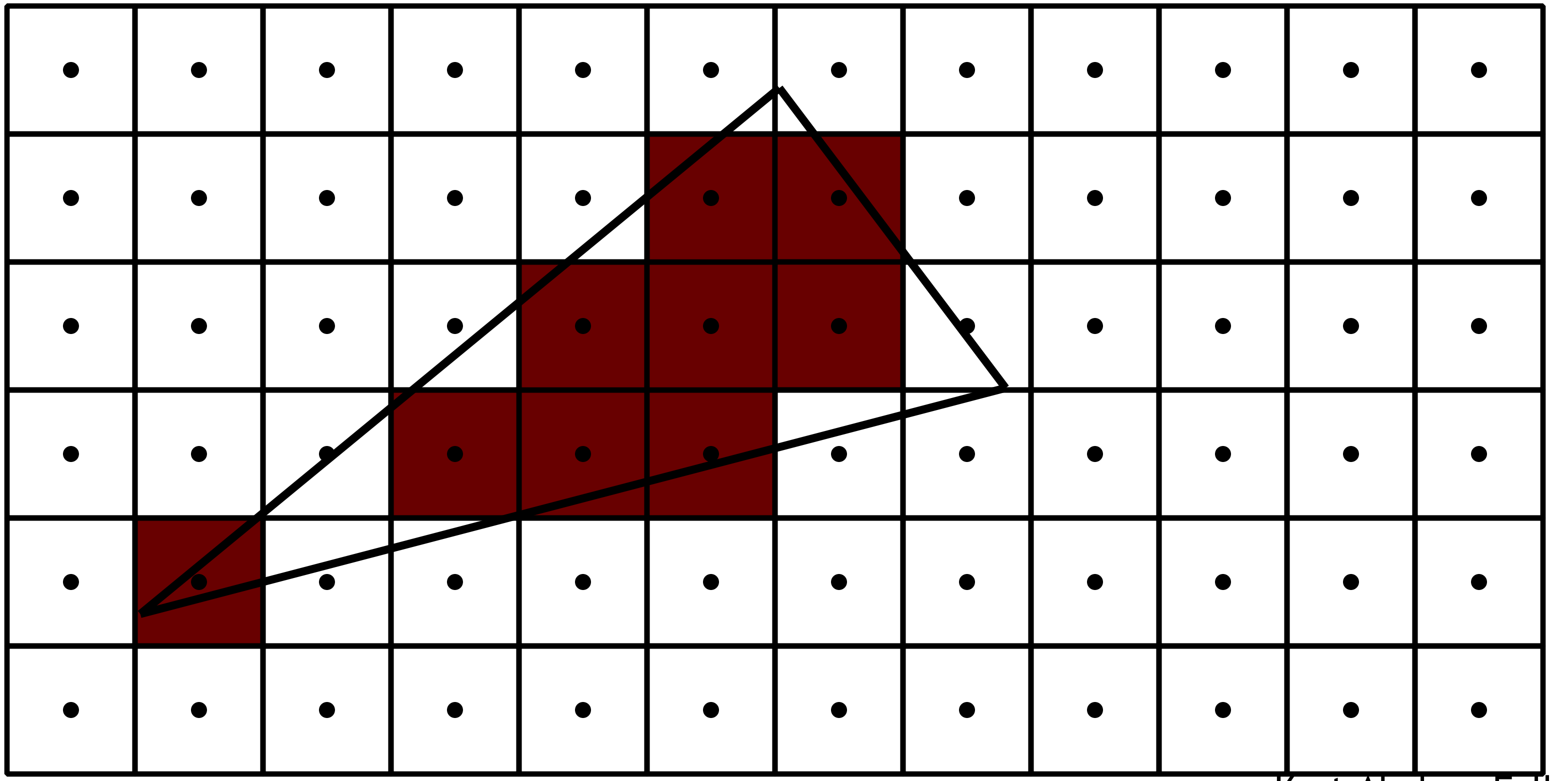
Rasterization

- Rasterization is the process of sampling primitives in window coordinates at each pixel
- Many ways to do this
- Have to worry about aliasing
  - What is aliasing? How does it appear in graphics?
  - What is the Nyquist frequency. What is the optimal sampling frequency?

Point-sampled fragment selection

Generate fragment if pixel center is inside triangle

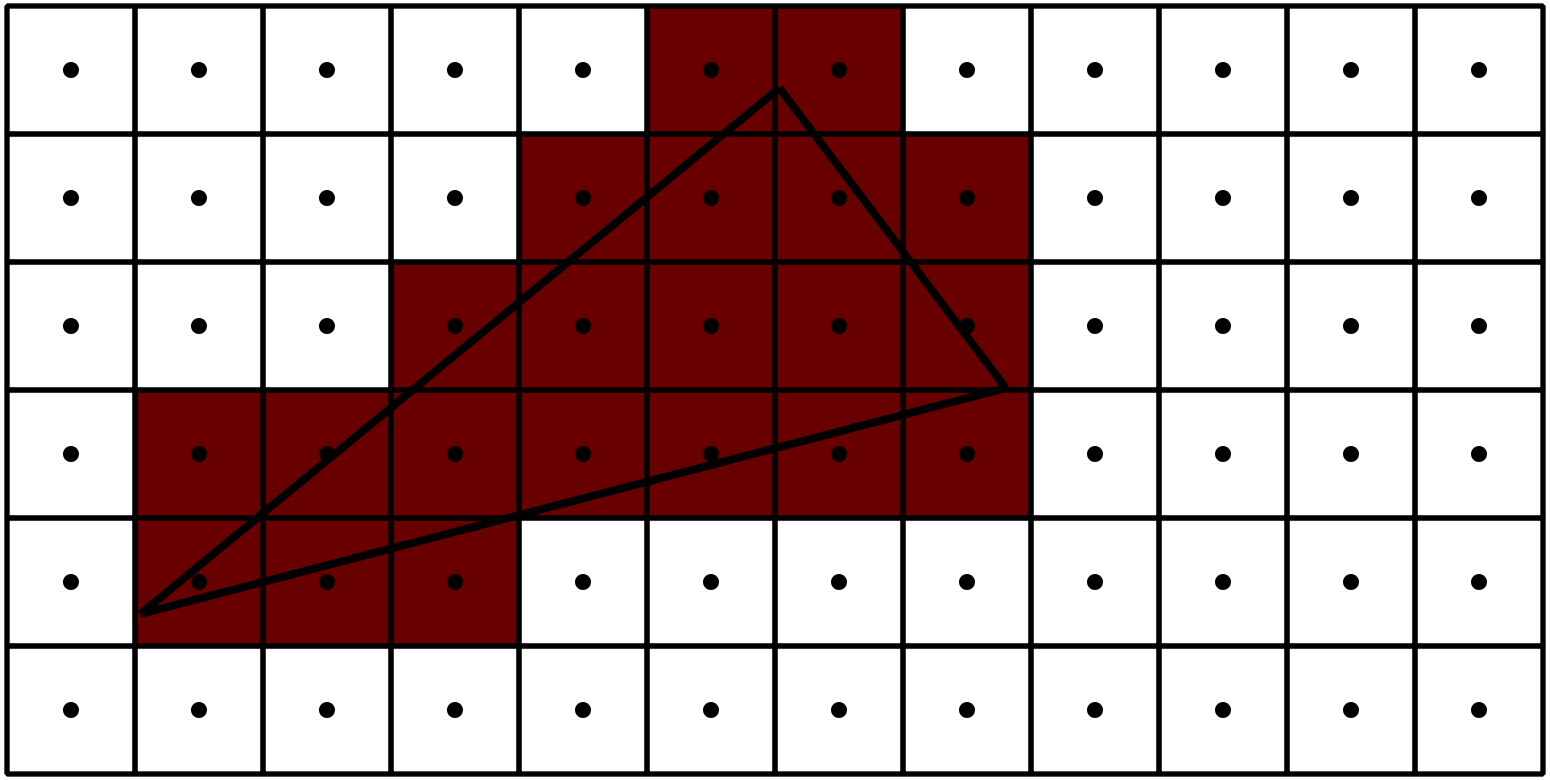
Implements point-sampled aliased rasterization



Tiled fragment selection

Generate fragment if unit square intersects triangle

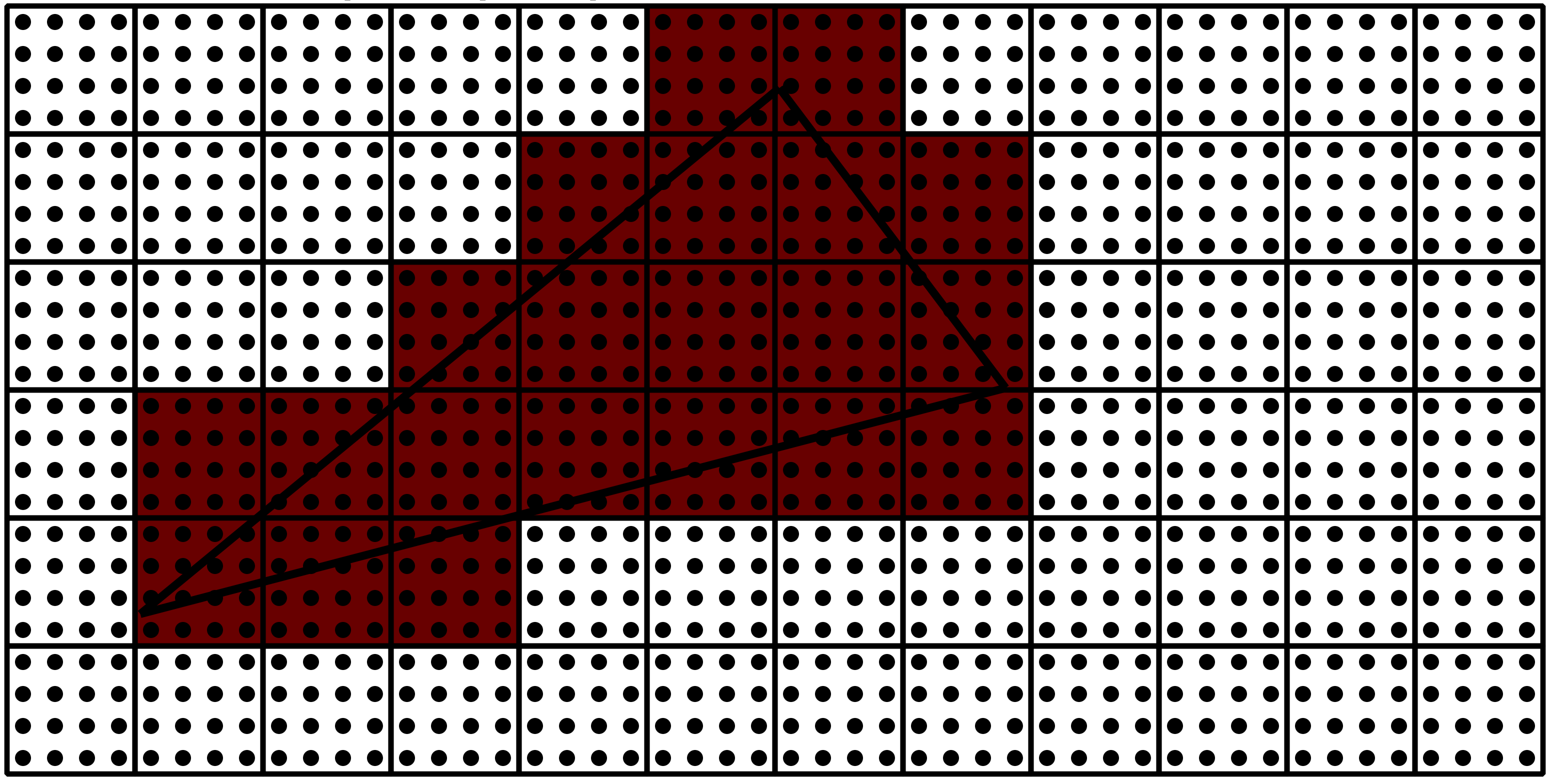
Implements multisample and tiled rasterizations



Tiled fragment selection

Multisample rasterization

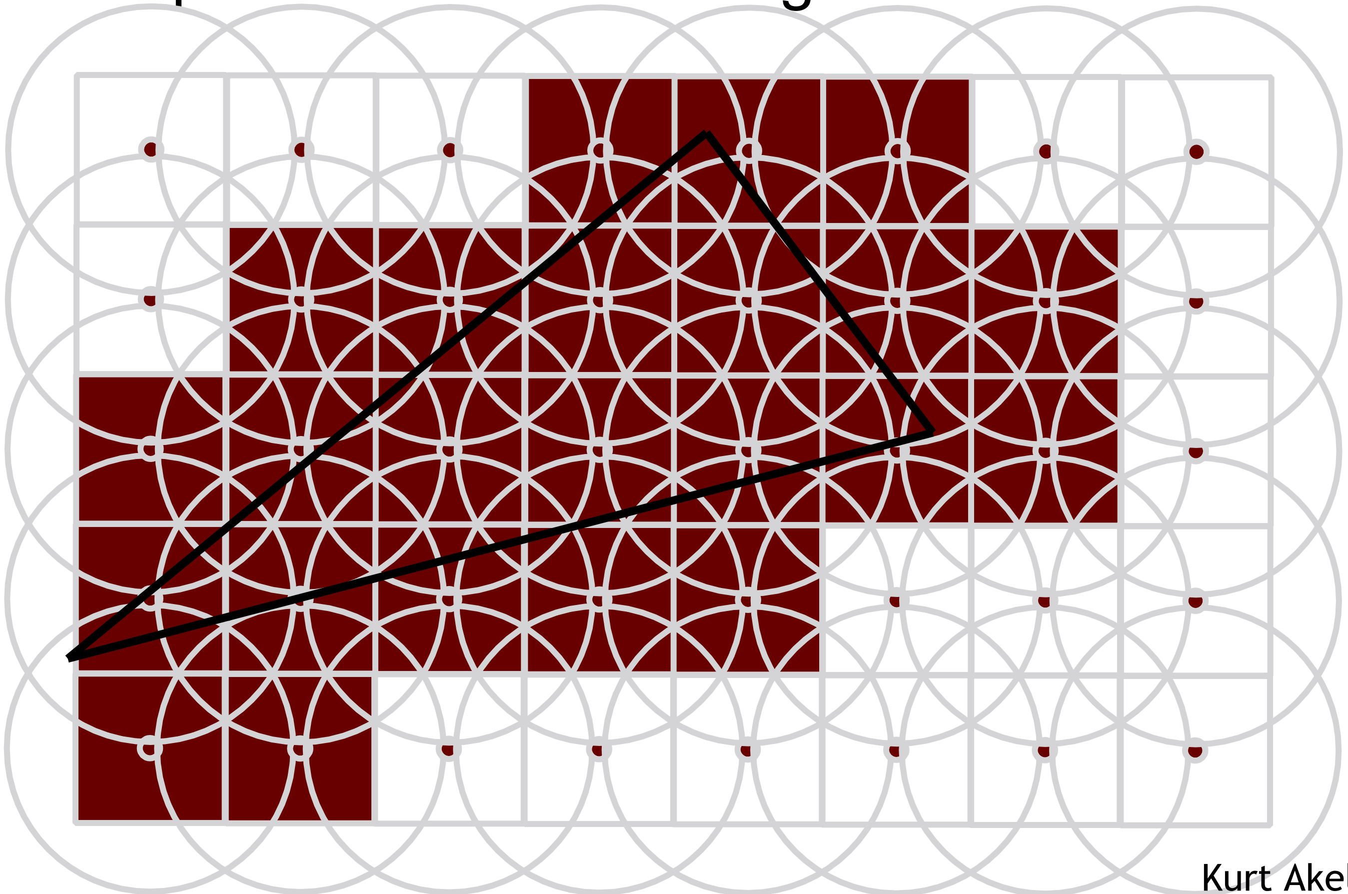
4x4 samples per pixel



Antialiased fragment selection

Generate fragment if filter function intersects triangle

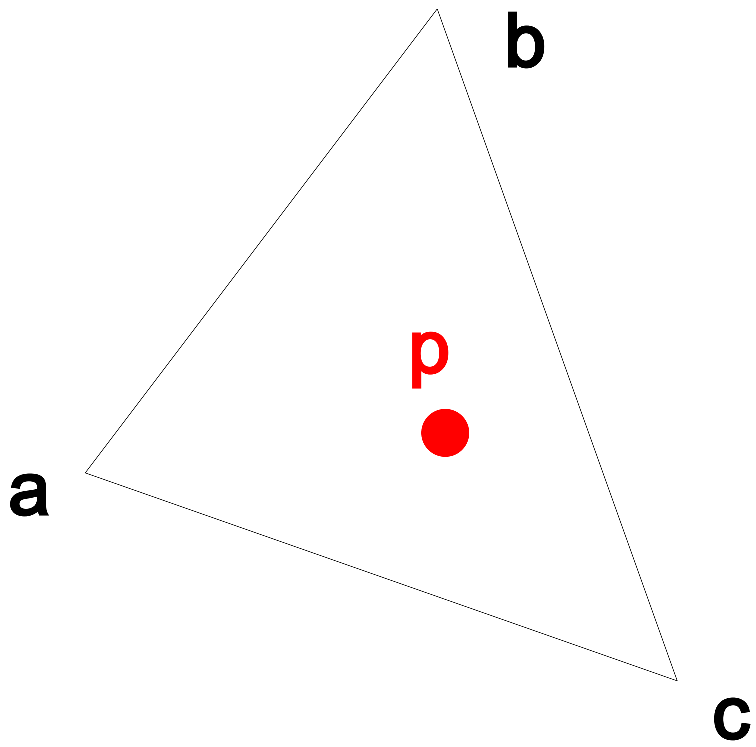
Implements pre-filtered antialiasing



Rasterization process

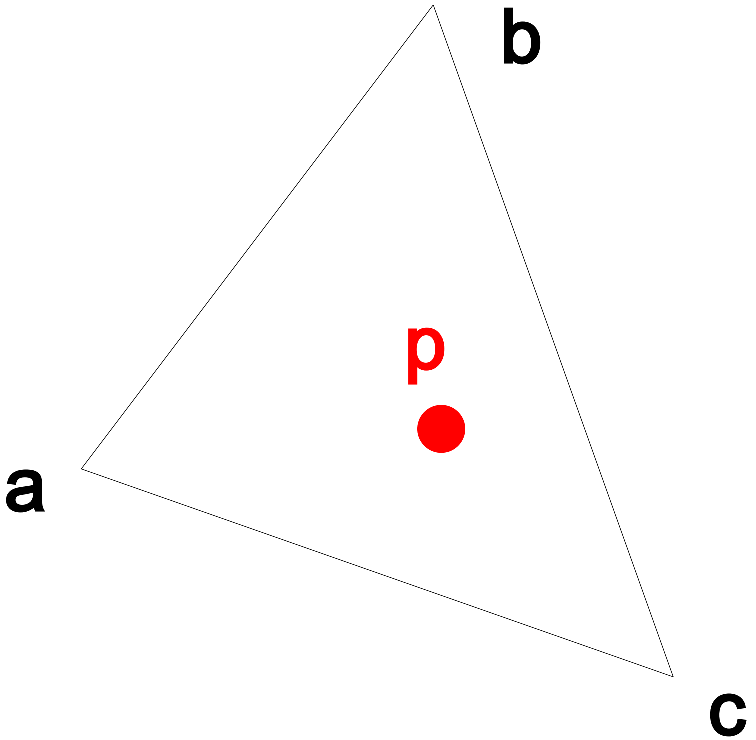
- Iterate over the intersection of the bounding rectangle and the frame buffer (for clipping)
- For each pixel (center), compute barycentric coordinates
- If they are between 0 and 1, compute interpolated values (depth, color, …) and store with generated fragment

Barycentric coordinates



$$p = \alpha\, a + \beta\, b + \gamma\, c \quad \text{such that} \quad \alpha+\beta+\gamma = 1$$

Barycentric coordinates

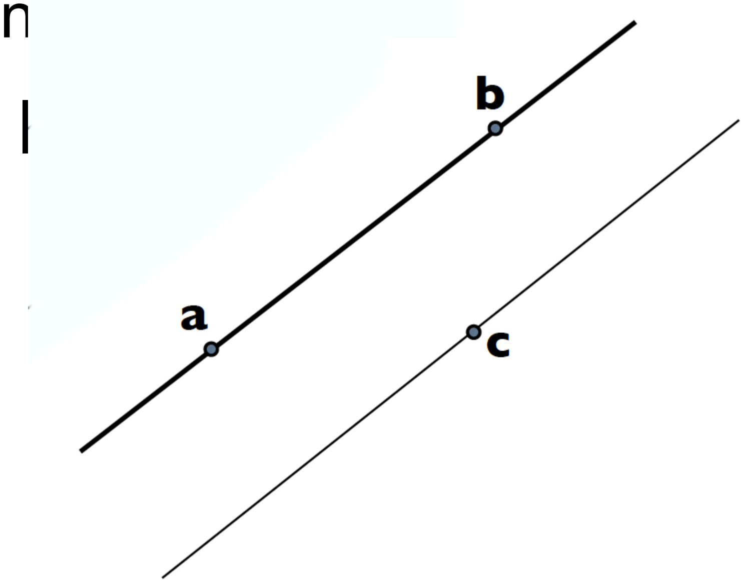


$$p = a + \beta\,(b-a) + \gamma\,(c-a) \qquad \text{since } \alpha = 1-\beta-\gamma$$

Barycentric coordinates

- How to compute β and γ ?

- $f_{ab}(x, y) = (y_a - y_b)x + (x_b - x_a)y + x_a y_b - x_b y_a$
- represents how far (x,y) is to the lin
- $f_{ab}(x, y) = 0$ if (x,y) is on the l
- $f_{ab}(x,y) / f_{ab}(x_c,y_c) = \gamma$
  (ratio to map this
  distance to [0,1])



Barycentric coordinates

$$g = \frac{(y_a - y_b)x + (x_b - x_a)y + x_a y_b - x_b y_a}{(y_a - y_b)x_c + (x_b - x_a)y_c + x_a y_b - x_b y_a}$$

$$b = \frac{(y_c - y_a)x + (x_a - x_c)y + x_c y_a - x_a y_c}{(y_c - y_a)x_b + (x_a - x_c)y_b + x_c y_a - x_a y_c}$$

$$a = 1 - b - g$$

Interpolating Values

- Vertices come with many attributes (i.e. colors, normal depth) which need to be interpolated to each fragment
- Simplest method is to use barycentric interpolation
- $v_p = \alpha \ v_a + \beta \ v_b + \gamma \ v_c$

HW 1 Q& A

Appendix:

Let’s See Some Code!

GLUT

- Cross Platform Windowing Library for OpenGL
  - Sets up window and associated buffers for you
  - Gives simple input handling functions

GLUT functions

- void glutInit(int argc, char** argv)
  - Takes arguments of main
- glutInitDisplayMode(unsigned int mode) [optional]
  - Tells GLUT which buffers to allocate
  - Ex. glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH)
- int glutCreateWindow(char* title)
  - Puts window on screen
- void glutDisplayFunc(void (*func) (void))
  - Takes callback to function where rendering happens
- void glutMainLoop()
  - Start rendering!

Simple Square

```
/* File: simple_glut.cpp */
#include <GL/glut.h>
void display(){

	//Set what color to clear to
	glClearColor (0.0, 0.0, 0.0, 0.0);
	//Clear the color buffer
	glClear (GL_COLOR_BUFFER_BIT);
	//Set The color
	glColor3f (1.0, 1.0, 1.0);

	//Set the current Projection Matrix
	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();
	glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);

	//Specify Geometry
	glBegin(GL_POLYGON);
	  glVertex3f (0.25, 0.25, 0.0);
	  glVertex3f (0.75, 0.25, 0.0);
	  glVertex3f (0.75, 0.75, 0.0);
	  glVertex3f (0.25, 0.75, 0.0);
	glEnd();
	//Flush Graphics buffer
	glFlush();

}

int main(int argc, char** argv){
	glutInit(&argc,argv);
	glutCreateWindow("Simple Glut");
	glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
	glutDisplayFunc(display);
	glutMainLoop();
}
```

```
/* File: modelview_example.cpp */
void display(){
          //Set what color to clear to
          glClearColor (0.0, 0.0, 0.0, 0.0);
          //Clear the color buffer
          glClear (GL_COLOR_BUFFER_BIT);
          //Set The color
          glColor3f (1.0, 1.0, 1.0);


          //Set the current Projection Matrix
          glMatrixMode(GL_PROJECTION);
          glLoadIdentity();
          glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);


          //Tile Squares
          glMatrixMode(GL_MODELVIEW);
          for(int i = 0; i< 3; i++){
                    for(int j = 0; j<3; j++){
                              glLoadIdentity();
                              glTranslatef(0.2f+ 0.3f*i,0.2f+ 0.3f*j,0);
                              DrawSquare();
                    }
          }
          //Flush Graphics buffer
          glFlush();
}

int main(int argc, char** argv){
          glutInit(&argc,argv);
          glutCreateWindow("Translation");
          glutDisplayFunc(display);
          glutMainLoop();
          return 0;
}
```

Remember, glTranslate is a multiplication, so don't forget glLoadIdentity()

GLU convenience functions

- GLU is set of useful, timesaving OpenGL utilities

  Ex. #include <GL/glu.h>
- void gluOrtho2D(left, right, bottom, top)

  Specifies a rectangular clipping region
- void gluLookAt(eyex, eyey,eyez,
  atx,aty,atz,
  upx,upy,upz)

  Specifies a perspective frustem

  Camera positioned at “eye”

  Camera looking at “at”

  Up vector is “up”

```
/* File: cube_example.cpp */
#include <GL/glut.h>
void display(void)
{
          glClearColor (0.0, 0.0, 0.0, 0.0);
          glClear (GL_COLOR_BUFFER_BIT);
          glColor3f (1.0, 1.0, 1.0);
          glLoadIdentity ();             /* clear the matrix */
          /* viewing transformation  */
          gluLookAt (0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
          glScalef (1.0, 2.0, 1.0);      /* modeling transformation */
          glutWireCube (1.0);
          glFlush ();
}

//Called whenever the GLUT window is reshaped
void reshape (int w, int h)
{
   glViewport (0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode (GL_PROJECTION);
   glLoadIdentity ();
   gluLookAt (0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
   glMatrixMode (GL_MODELVIEW);
}

int main(int argc, char** argv)
{
   glutInit(&argc, argv);
   glutCreateWindow ("Cube Example");
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   glutMainLoop();
   return 0;
}
```