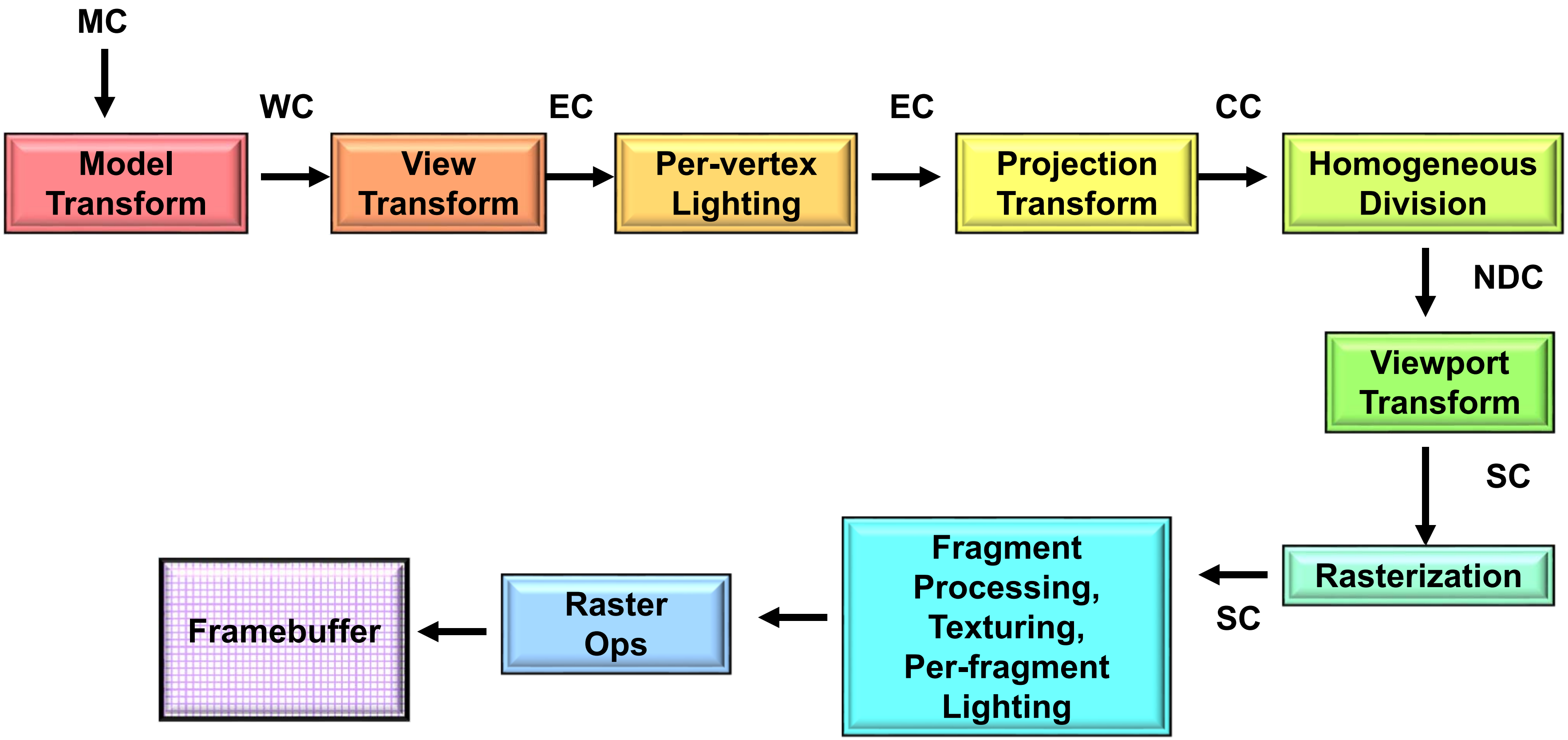
Getting Started with OpenGL Graphics Programming

Mike Bailey

mjb@cs.oregonstate.edu

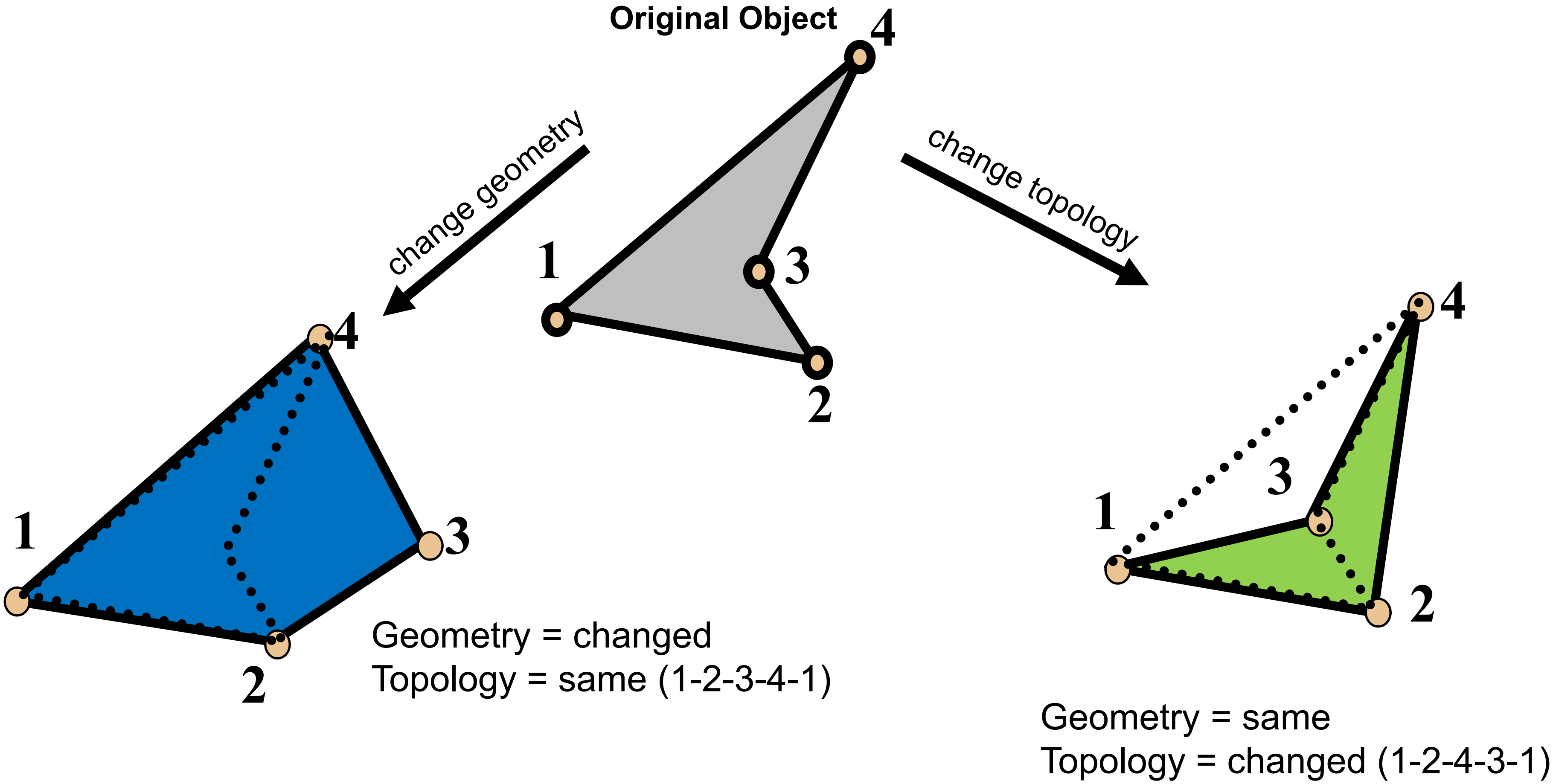
The Basic Computer Graphics Pipeline



We'll come back to this later. For now, understand that there are multiple steps to go from your 3D geometry to pixels on the screen.

MC = Model Coordinates
WC = World Coordinates
EC = Eye Coordinates
CC = Clip Coordinates
NDC = Normalized Device Coordinates
SC = Screen Coordinates

Geometry vs. Topology



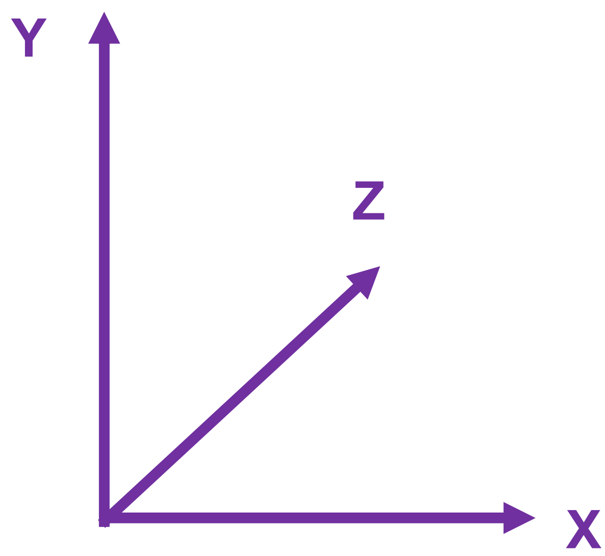
Geometry:

Where things are (e.g., coordinates)

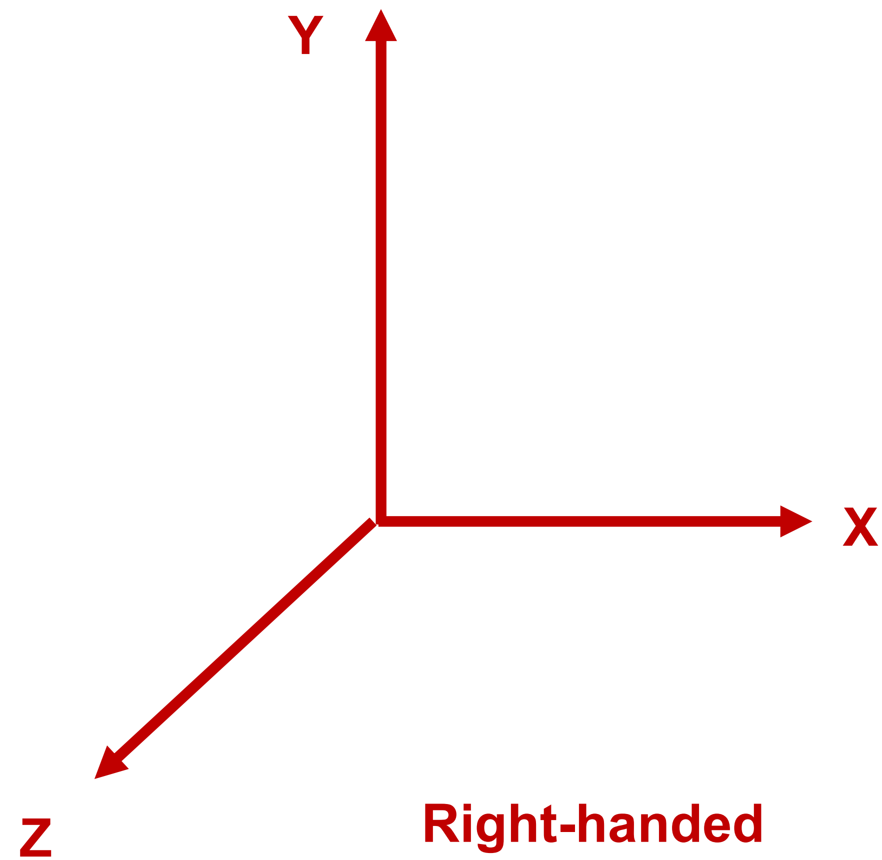
Topology:

How things are connected

3D Coordinate Systems

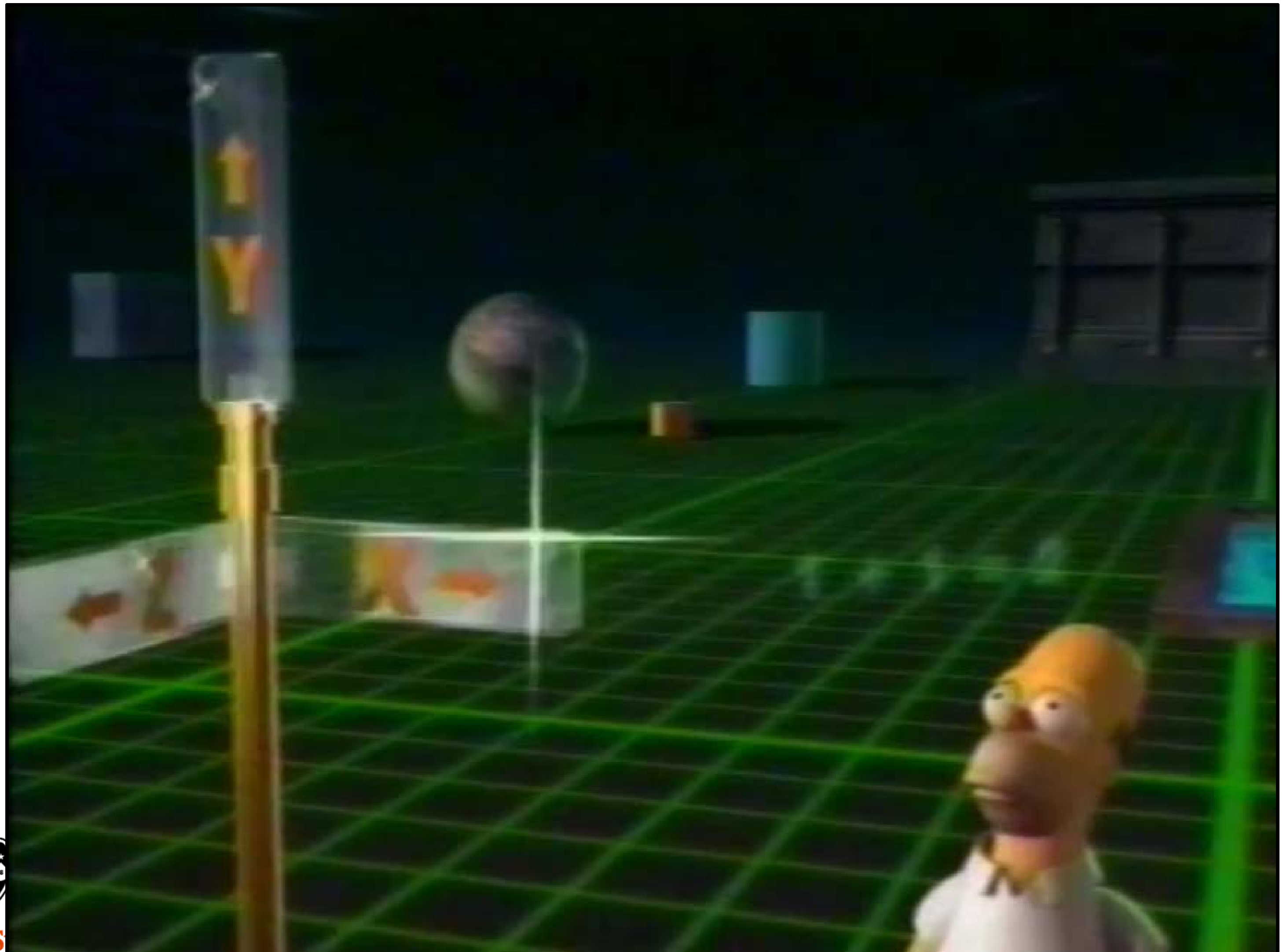


Left-handed

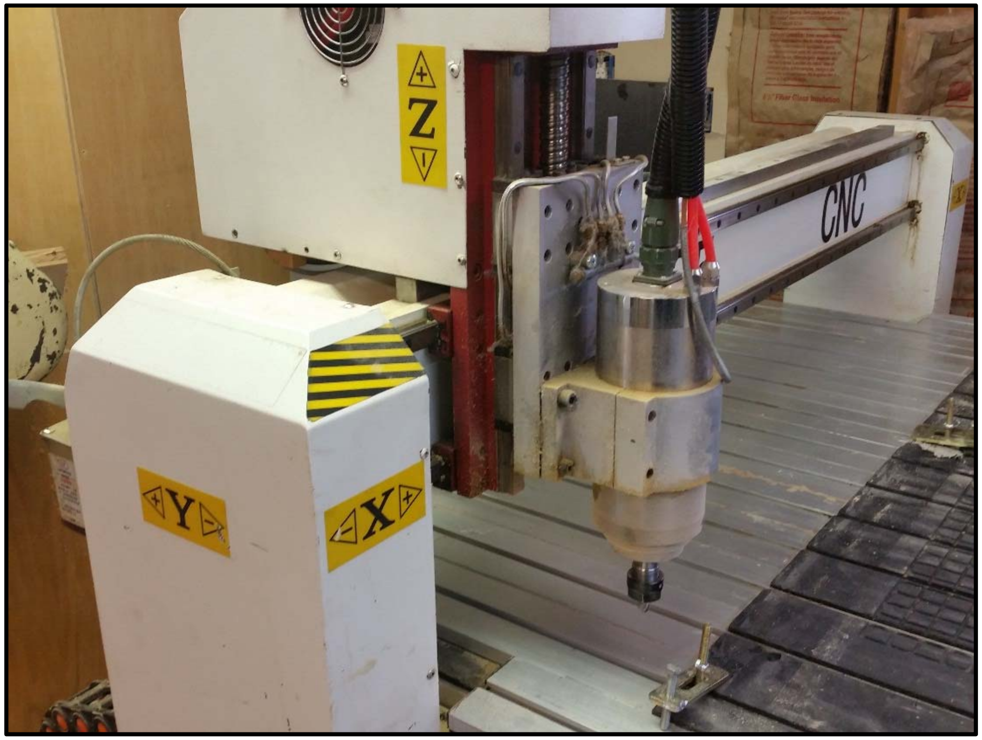


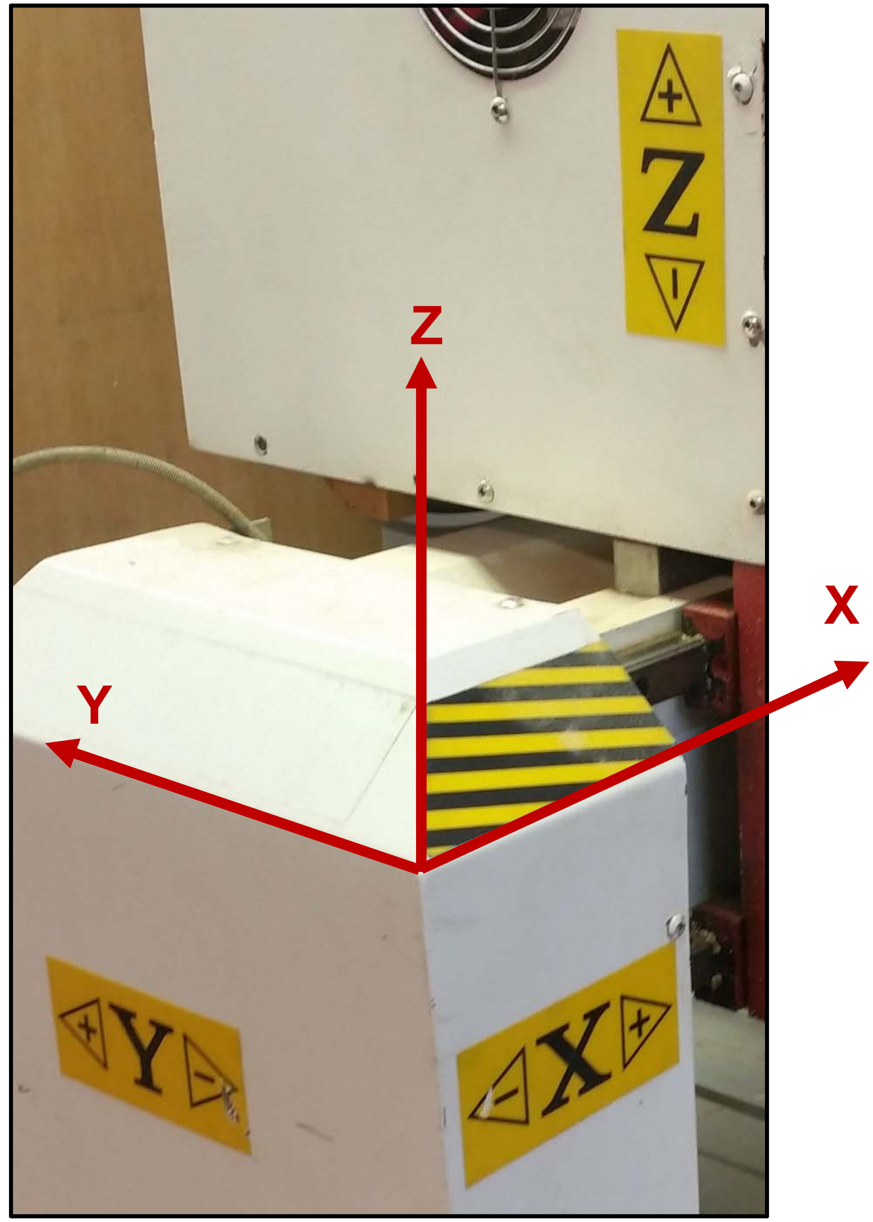
Right-handed

Since Homer Simpson uses Right-handed Coordinates, then we will too

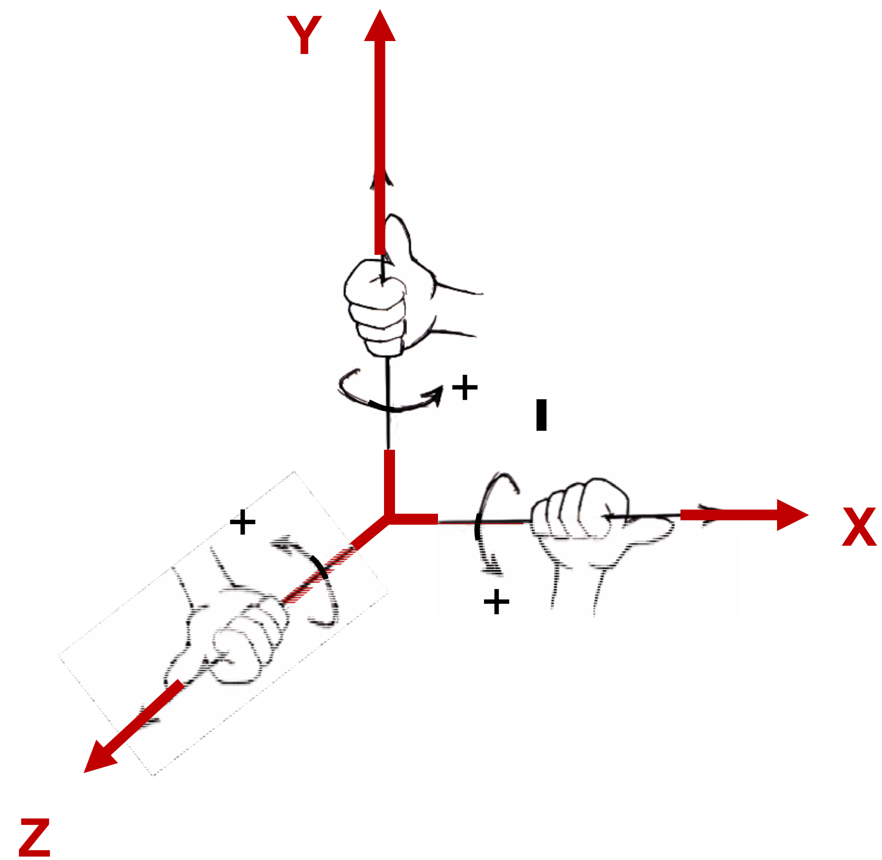


Right-handed 3D Coordinate System for a CNC Machine





Right-handed Positive Rotations



Right-Handed Coordinate System

Drawing in 3D

```
glColor3f( r, g, b );

glBegin( GL_LINE_STRIP );
        glVertex3f( x0, y0, z0 );
        glVertex3f( x1, y1, z1 );
        glVertex3f( x2, y2, z2 );
        glVertex3f( x3, y3, z3 );
        glVertex3f( x4, y4, z4 );
glEnd( );
```

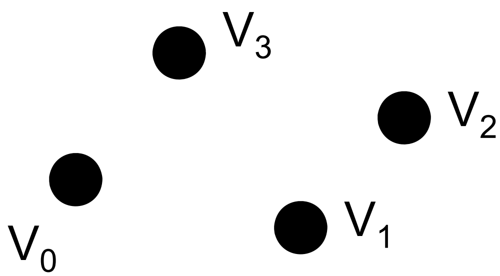
Set any display-characteristics ***state*** that you want to have in effect when you do the drawing

Begin the drawing. Use the current state's display-characteristics. Here is the topology to be used with these vertices

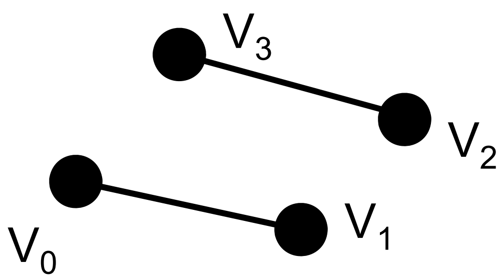
This is a wonderfully understandable way to start with 3D graphics, but it is also incredibly inefficient! We'll talk about that later…

OpenGL Topologies

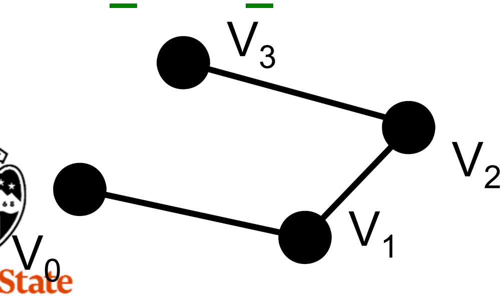
GL_POINTS



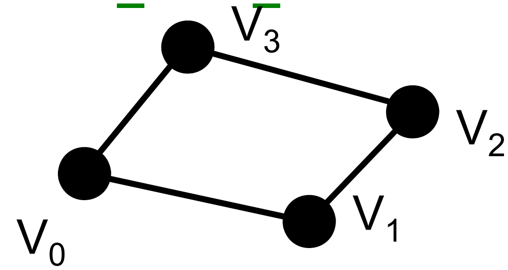
GL_LINES



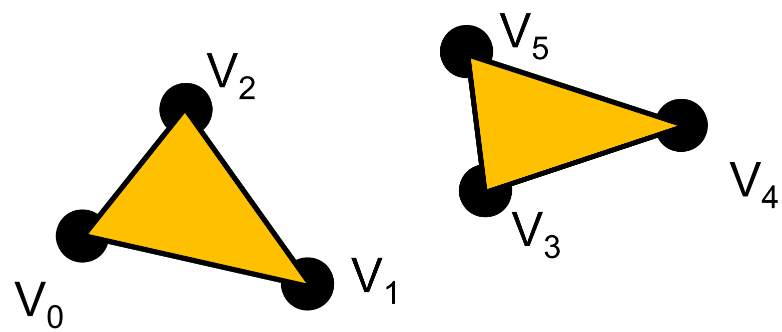
GL_LINE_STRIP



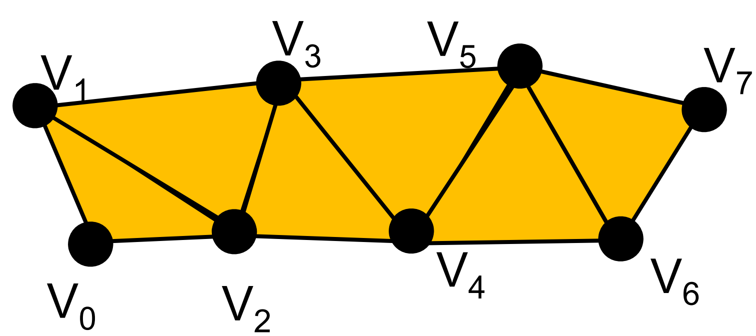
GL_LINE_LOOP



GL_TRIANGLES

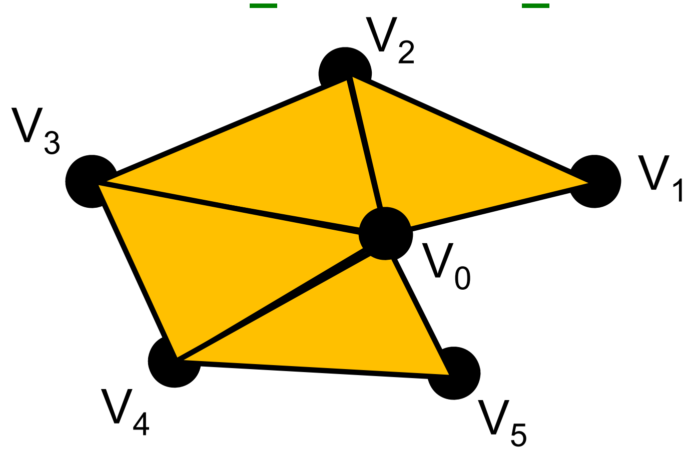


GL_TRIANGLE_STRIP

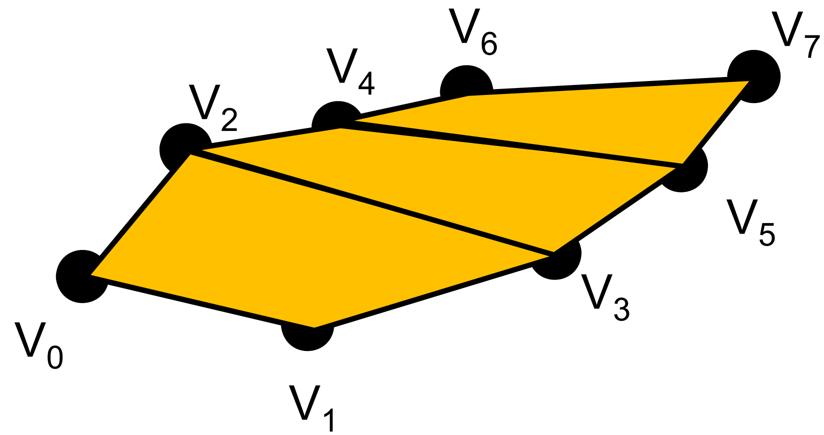


OpenGL Topologies

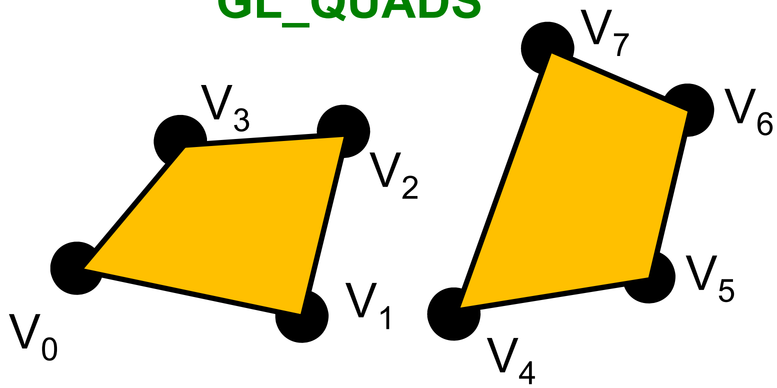
GL_TRIANGLE_FAN



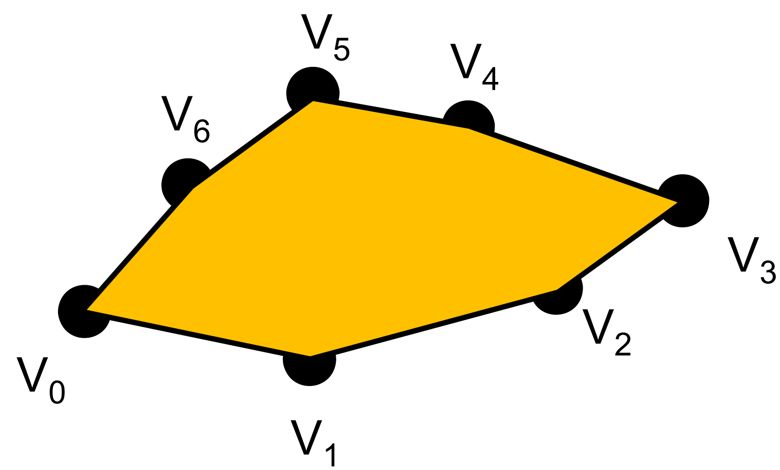
GL_QUAD_STRIP



GL_QUADS



GL_POLYGON

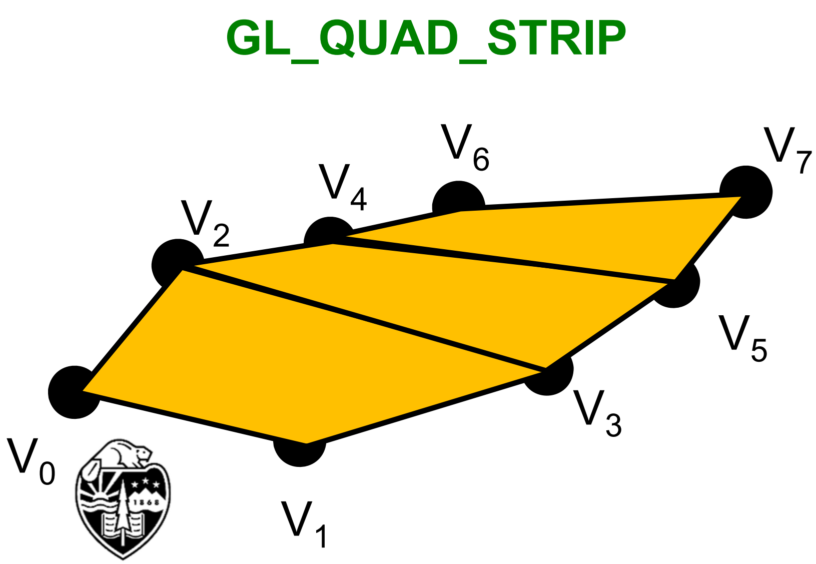


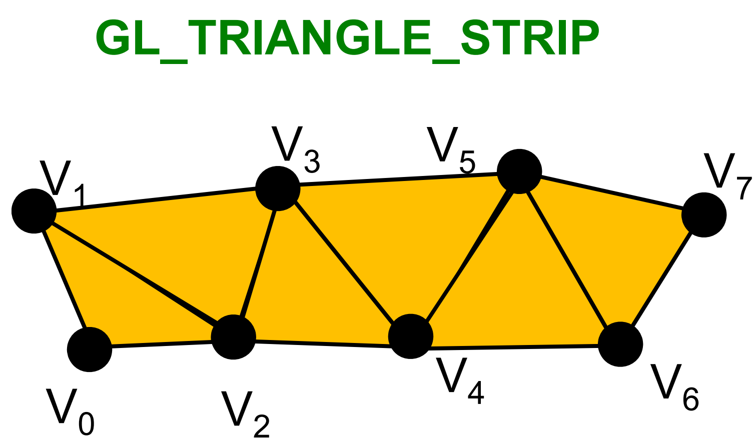
OpenGL Topologies – Polygon Requirements

Polygons must be:

- **Convex** and
- **Planar**

For that reason, GL_TRIANGLE_STRIP and GL_TRIANGLE are considered preferable to GL_QUAD_STRIP and GL_QUADS. GL_POLYGON is rarely used.



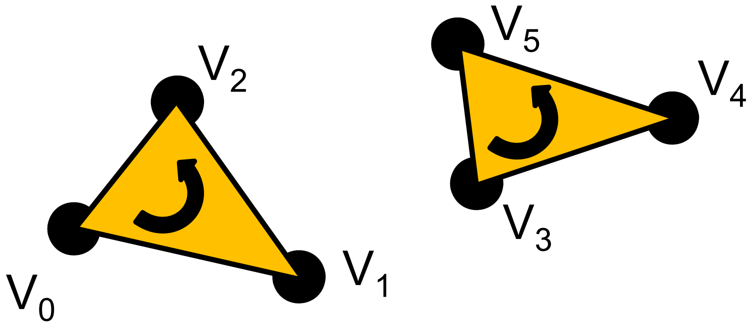


OpenGL Topologies -- Orientation

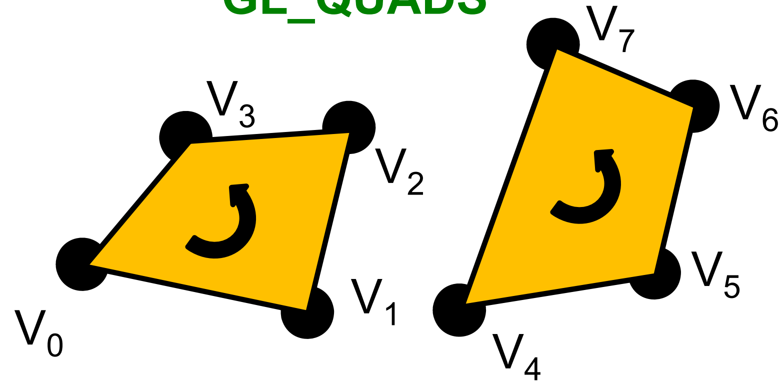
Polygons are traditionally:

- **CCW when viewed from outside the solid object**

GL_TRIANGLES



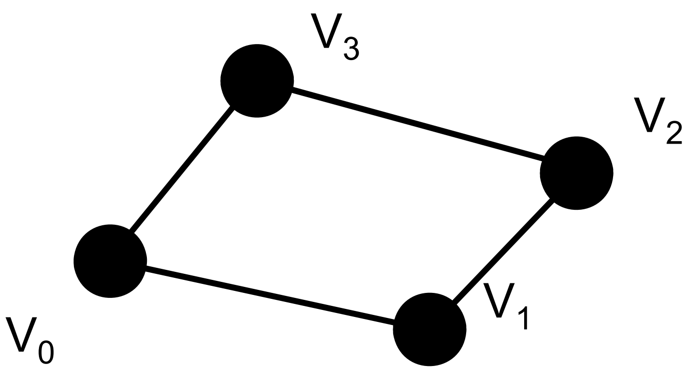
GL_QUADS



It doesn't matter much, but there is an advantage in being ***consistent***

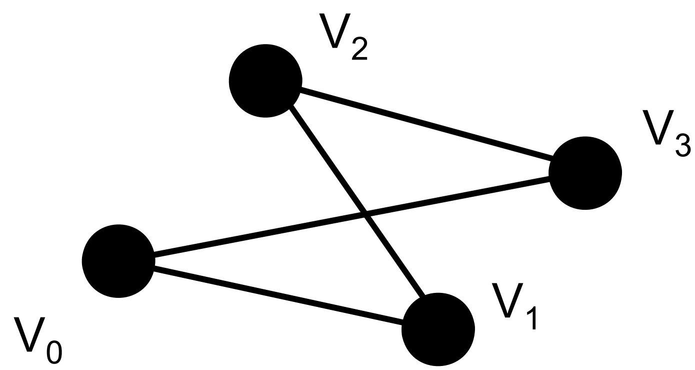
OpenGL Topologies – Vertex Order Matters

GL_LINE_LOOP



Probably what you meant to do

GL_LINE_LOOP

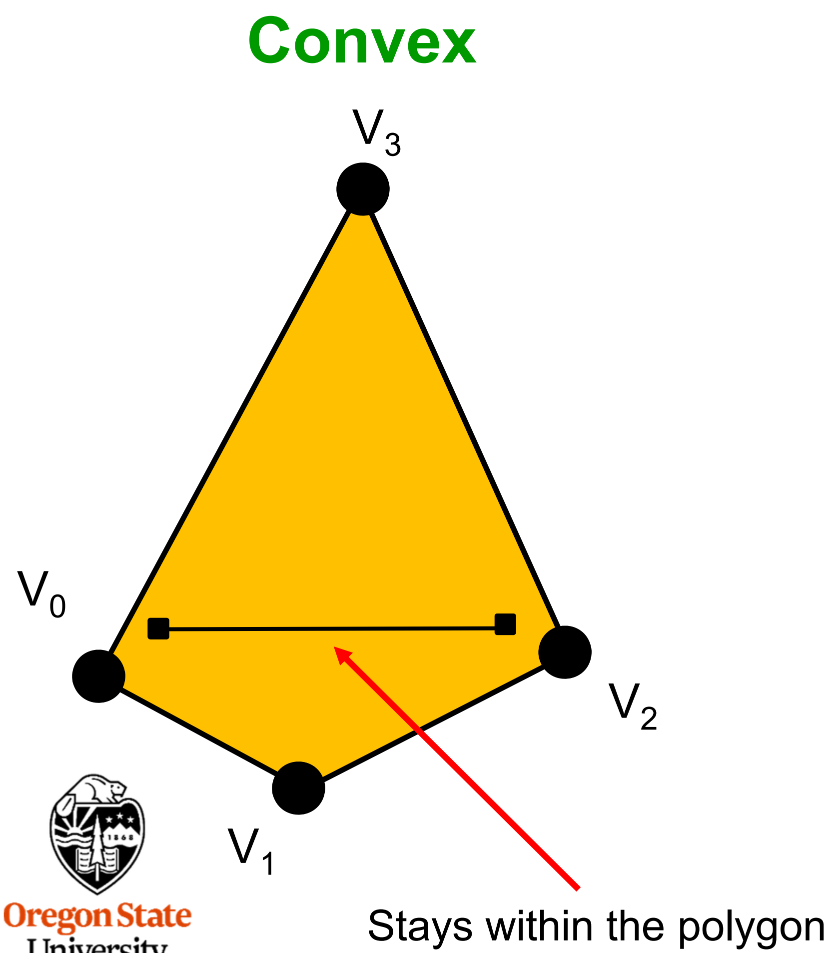


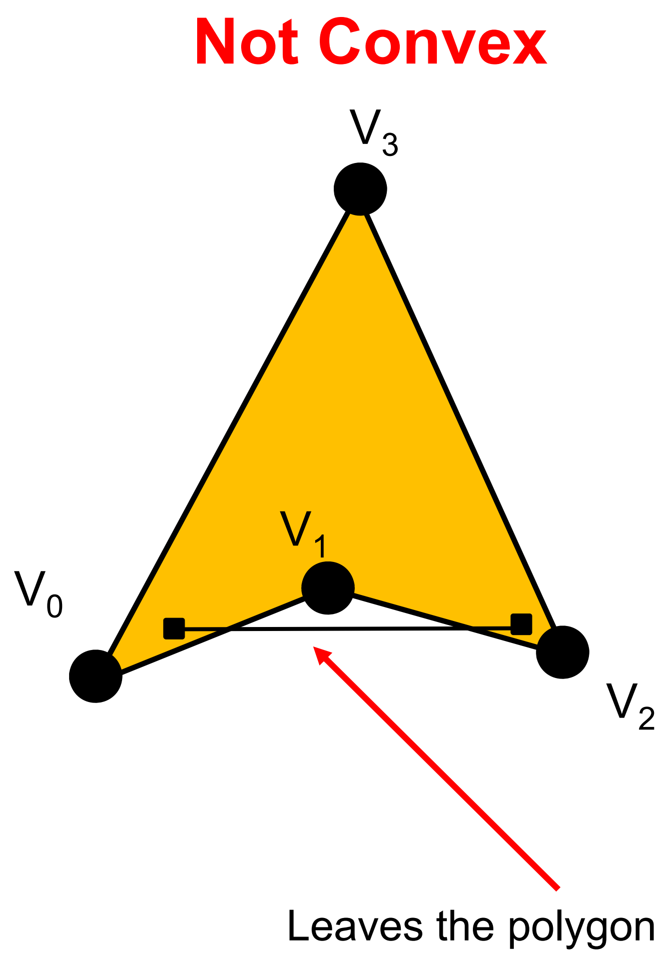
Probably not what you meant to do

This disease is referred to as “The Bowtie” ☺

What does "Convex Polygon" Mean?

We can go all mathematical here, but let's go visual instead. In a convex polygon, a line between ***any*** two points inside the polygon never leaves the inside of the polygon.

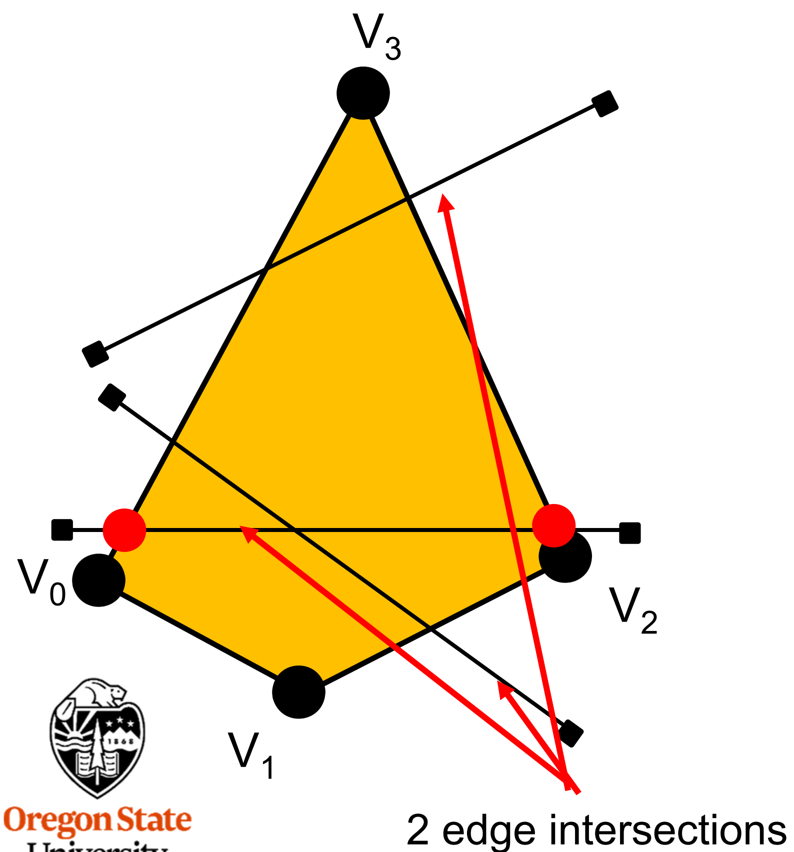




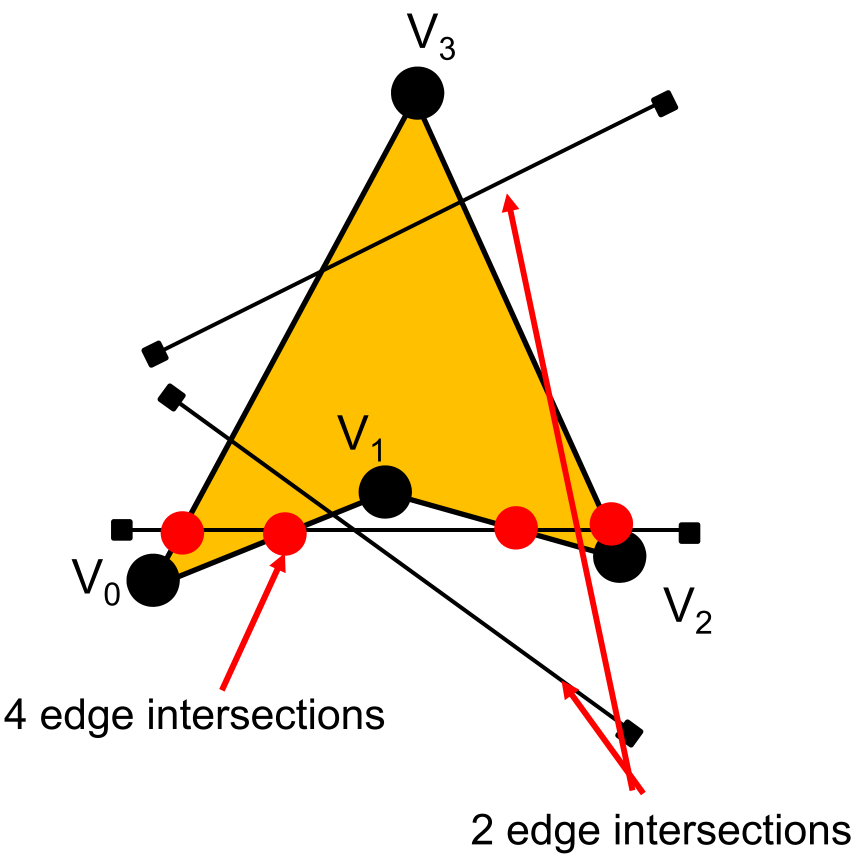
Why is there a Requirement for Polygons to be Convex?

Graphics polygon-filling hardware can be highly optimized if you know that, no matter what direction you fill the polygon in, there will be two and only two intersections between the scanline and the polygon's edges

Convex



Not Convex



Oregon State University
Computer Graphics

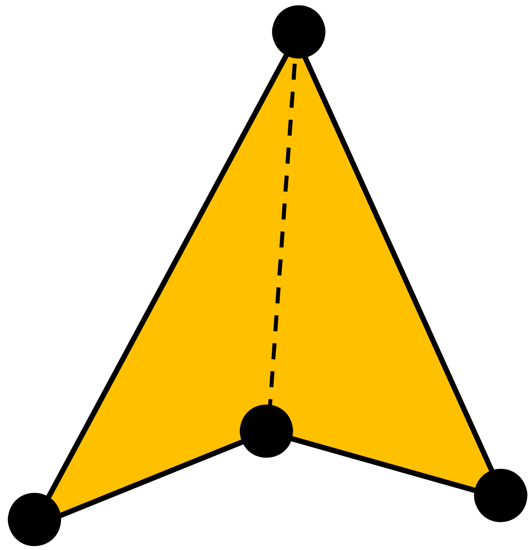
What if you need to display Polygons that are not Convex?

There are two good solutions I know of (and there are probably more):

1. OpenGL's utility (gluXxx) library has a built-in tessellation capability to break a non-convex polygon into convex polygons.

2. There is an open source library to break a non-convex polygon into convex polygons. It is called ***Polypartition***, and the source code can be found here:

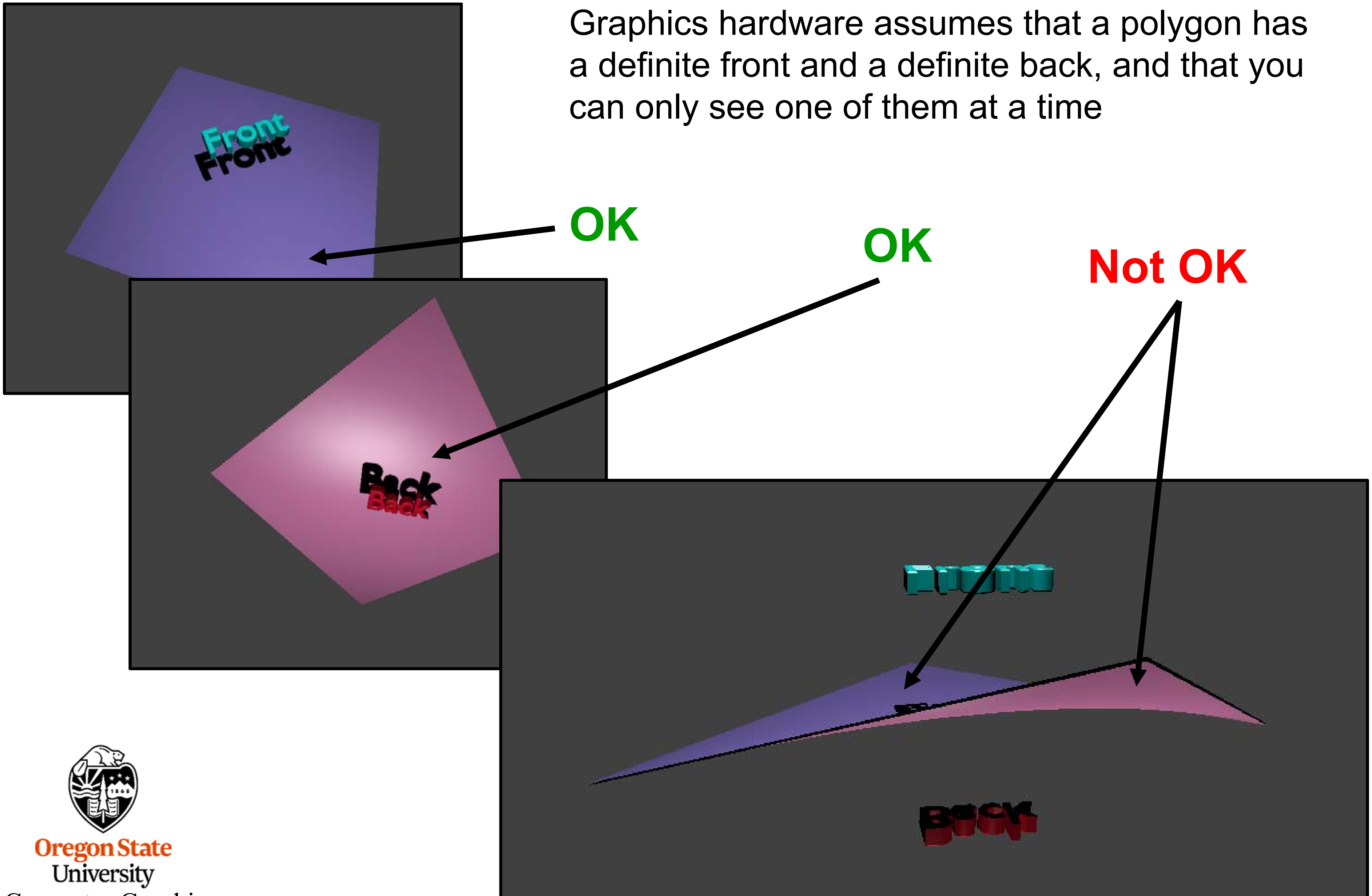
   https://github.com/ivanfratric/polypartition



If you ever need to do this, contact me. I have working code for each approach…

Why is there a Requirement for Polygons to be Planar?

Graphics hardware assumes that a polygon has a definite front and a definite back, and that you can only see one of them at a time

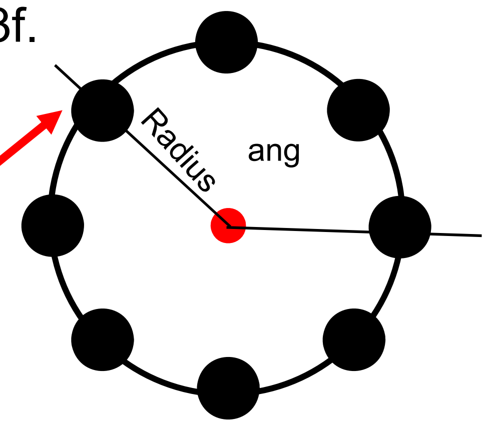


OpenGL Drawing Can Be Done Procedurally

Listing a lot of vertices explicitly gets old in a hurry

```
glColor3f( r, g, b );
glBegin( GL_LINE_LOOP );
          glVertex3f( x0, y0, 0. );
          glVertex3f( x1, y1, 0. );
          . . .
glEnd( );
```

The graphics card can’t tell how the numbers in the glVertex3f calls were produced: both explicitly listed and procedurally computed look the same to glVertex3f.



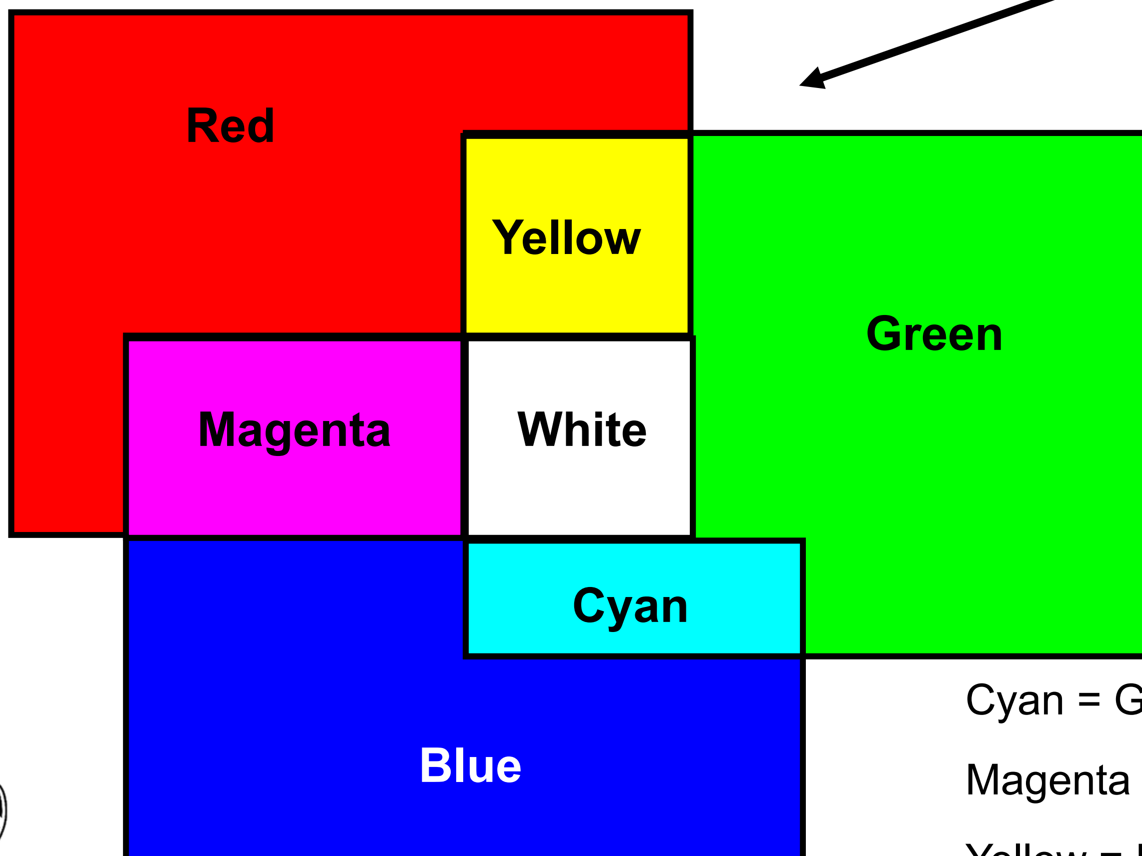

```
glColor3f( r, g, b );
float dang = 2. * M_PI / (float)( NUMSEGS – 1 );
float ang = 0.;
glBegin( GL_LINE_LOOP );
          for( int i = 0; i < NUMSEGS; i++ )
          {
                    glVertex3f( RADIUS*cos(ang),  RADIUS*sin(ang),  0. );
                    ang += dang;
          }
glEnd( );
```

Color

glColor3f(r, g, b);

0.0 ≤ r, g, b ≤ 1.0

This is referred to as **“Additive Color”**



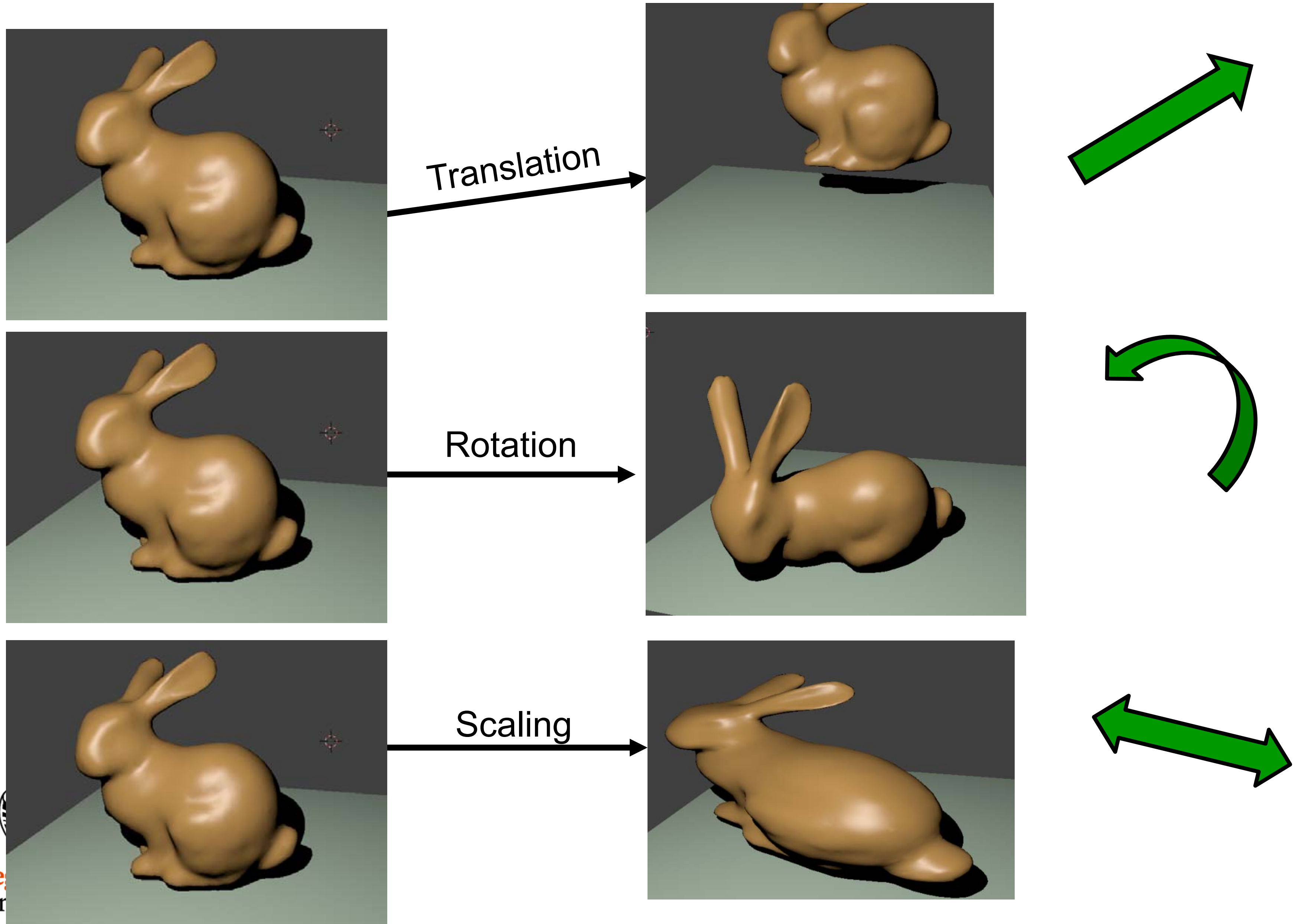
Cyan = Green + Blue

Magenta = Red + Blue

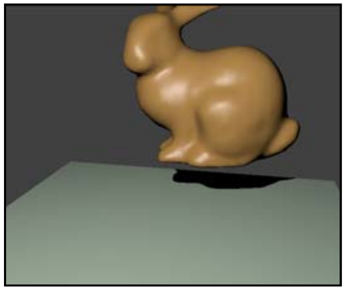
Yellow = Red + Green

White = Red + Green + Blue

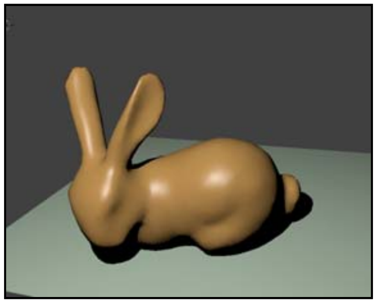
Transformations



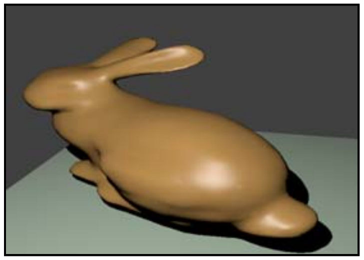
OpenGL Transformations



glTranslatef(tx, ty, tz);



glRotatef(*degrees*, ax, ay, az);



glScalef(sx, sy, sz);

Single Transformations

```
glMatrixMode( GL_MODELVIEW );
glLoadIdentity( )


glRotatef( degrees,  ax, ay, az );


glColor3f( r, g, b );
glBegin( GL_LINE_STRIP );
        glVertex3f( x0, y0, z0 );
        glVertex3f( x1, y1, z1 );
        glVertex3f( x2, y2, z2 );
        glVertex3f( x3, y3, z3 );
        glVertex3f( x4, y4, z4 );
glEnd( );
```

Compound Transformations

```
glMatrixMode( GL_MODELVIEW );
glLoadIdentity( )

glTranslatef( tx, ty, tz );            3.
glRotatef( degrees,  ax, ay, az );     2.
glScalef( sx, sy, sz );                1.

glColor3f( r, g, b );
glBegin( GL_LINE_STRIP );
        glVertex3f( x0, y0, z0 );
        glVertex3f( x1, y1, z1 );
        glVertex3f( x2, y2, z2 );
        glVertex3f( x3, y3, z3 );
        glVertex3f( x4, y4, z4 );
glEnd( );
```

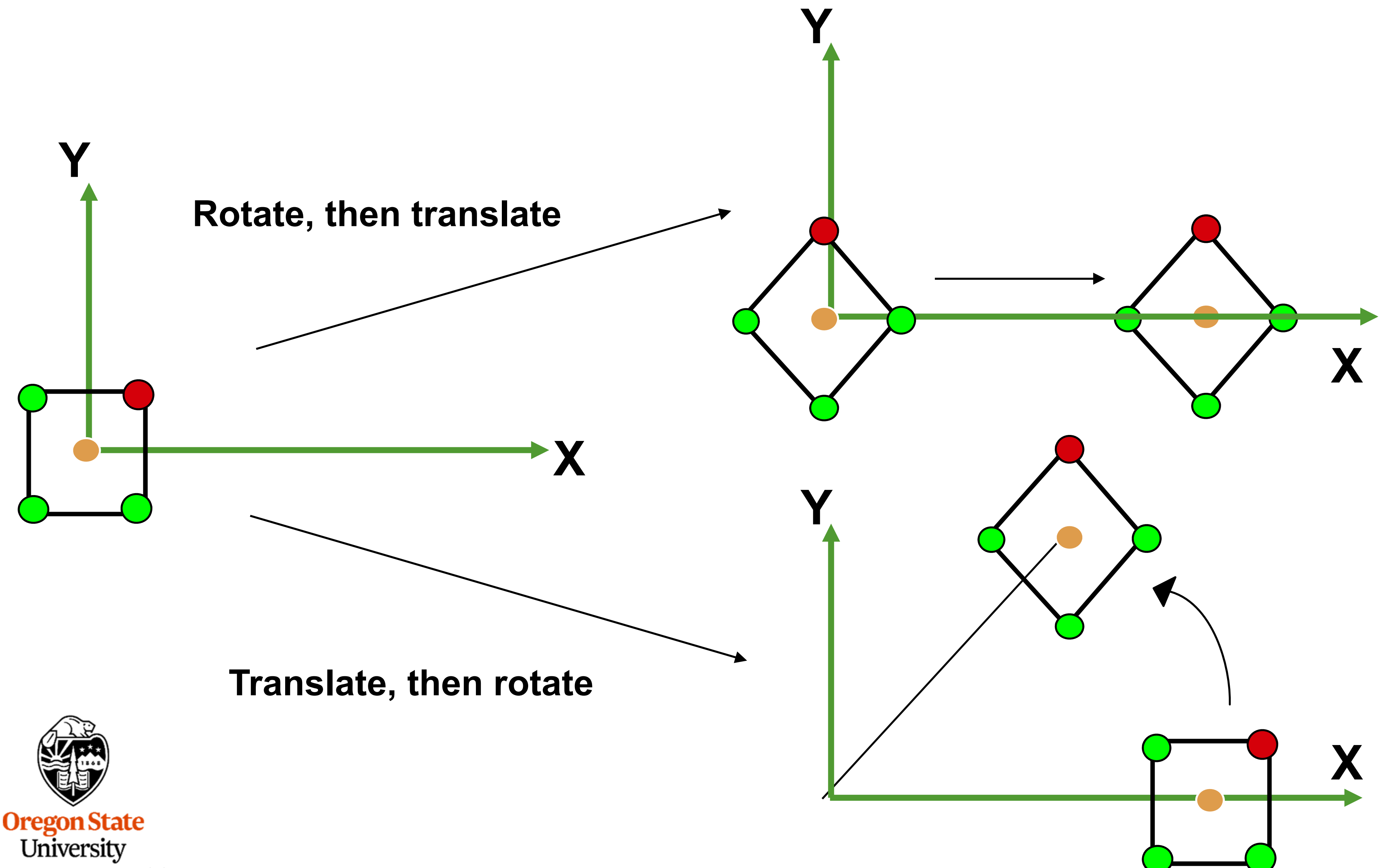
These transformations “add up”, and look like they take effect in this order

Why do the Compound Transformations Take Effect in Reverse Order?

```
3.  2.  1.
glTranslatef( tx, ty, tz );
glRotatef( degrees,  ax, ay, az );
glScalef( sx, sy, sz );
glBegin( GL_LINE_STRIP );
        glVertex3f( x0, y0, z0 );
        glVertex3f( x1, y1, z1 );
        glVertex3f( x2, y2, z2 );
        glVertex3f( x3, y3, z3 );
        glVertex3f( x4, y4, z4 );
glEnd( );
1.  2.  3.
```

Envision fully-parenthesizing what is going on. In that case, it makes perfect sense that the most recently-set transformation would take effect first.

Order Matters!
Compound Transformations are Not Commutative



The OpenGL Drawing State

The designers of OpenGL could have put lots and lots of arguments on the glVertex3f call to totally define the appearance of your drawing, like this:

glVertex3f(x, y, z,   r, g, b,   m00, …, m33,   s, t,   nx, ny, nz, linewidth, …   );

Yuch! *That* would have been ugly. Instead, they decided to let you create a "current drawing state". You set all of these characteristics first, then they take effect when you do the drawing. They continue to remain in effect for future drawing calls, until you change them.

You must set the transformations before you expect them to take effect!

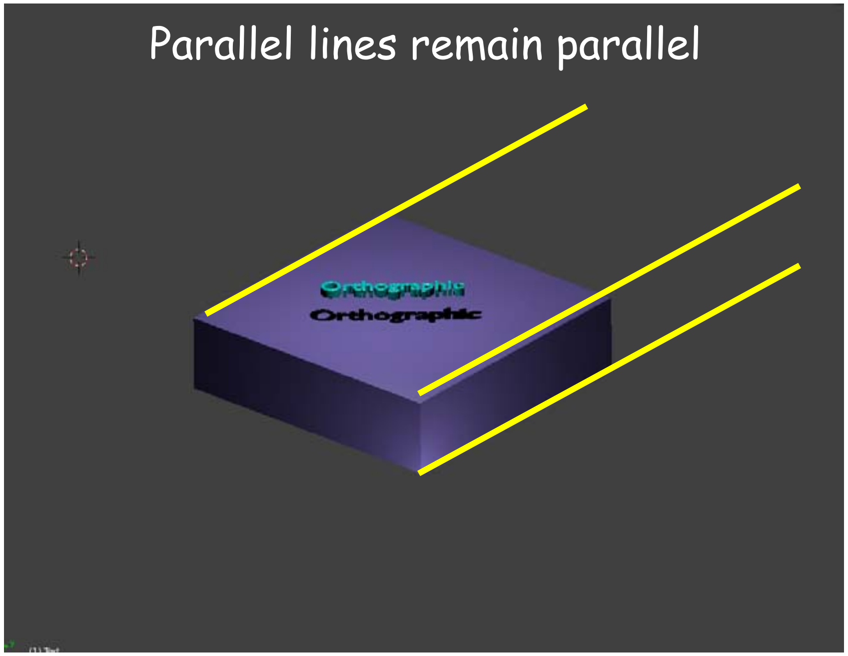
Set the state first

Use the state second

```
glMatrixMode( GL_MODELVIEW );
glLoadIdentity( )

glTranslatef( tx, ty, tz );
glRotatef( degrees,  ax, ay, az );
glScalef( sx, sy, sz );

glColor3f( r, g, b );
glBegin( GL_LINE_STRIP );
        glVertex3f( x0, y0, z0 );
        glVertex3f( x1, y1, z1 );
        glVertex3f( x2, y2, z2 );
        glVertex3f( x3, y3, z3 );
        glVertex3f( x4, y4, z4 );
glEnd( );
```

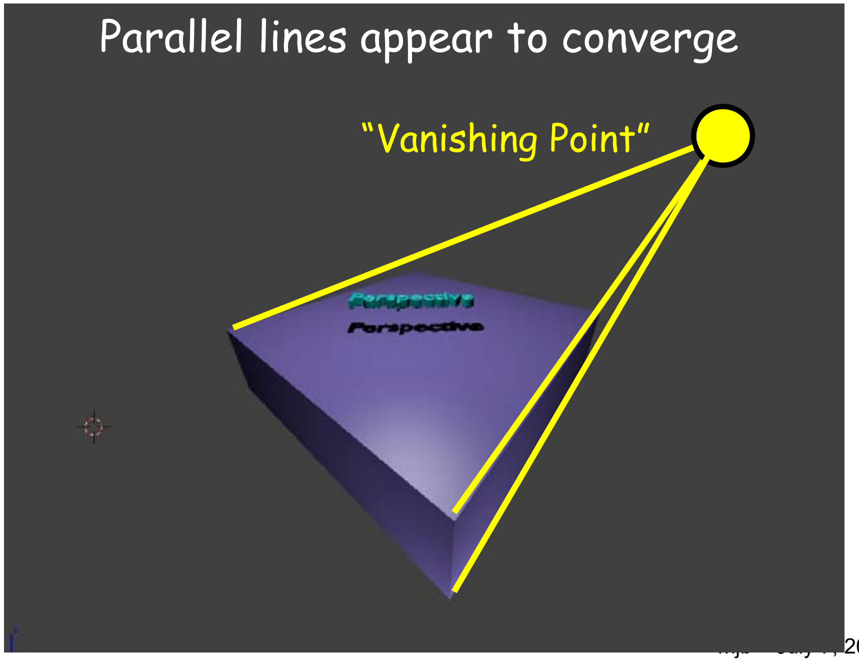
Projecting an Object from 3D into 2D

Orthographic (or Parallel) projection



glOrtho(xl, xr, yb, yt, zn, zf);

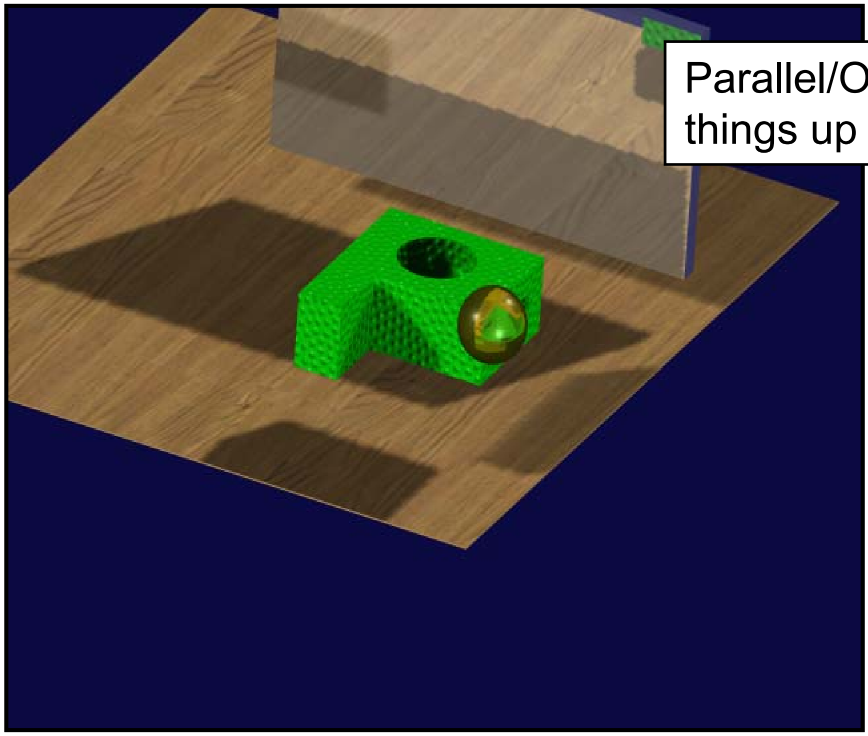
Perspective projection



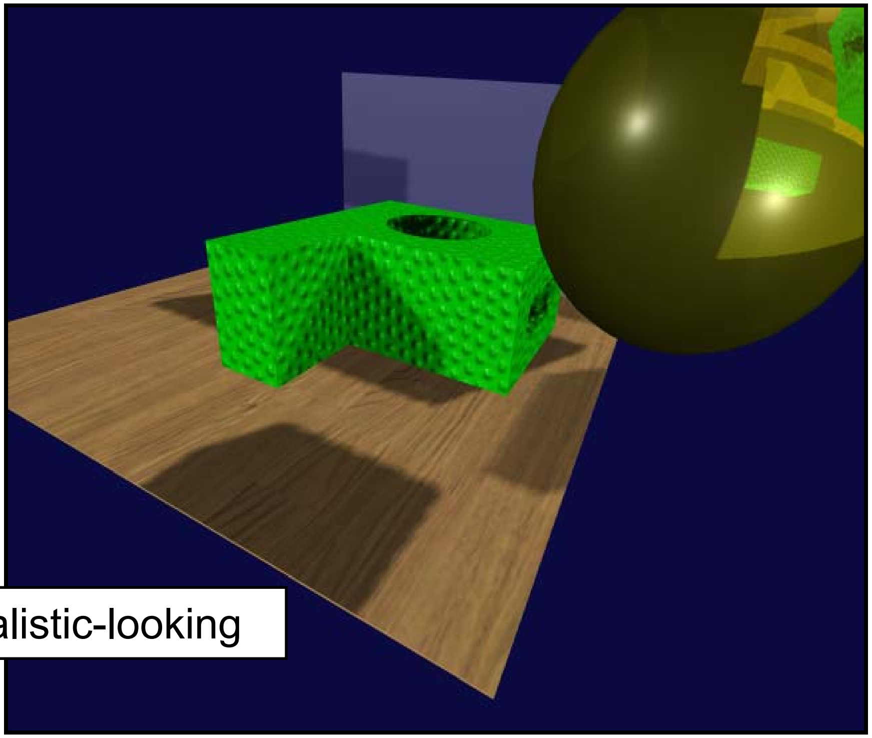
gluPerspective(fovy, aspect, zn, zf);

Oregon State
University
Computer Graphics

Projecting on Object from 3D to 2D



Parallel/Orthographic is good for lining things up and comparing sizes



Perspective is more realistic-looking

OpenGL Projection Functions

```
glMatrixMode( GL_PROJECTION );
glLoadIdentity( );
```

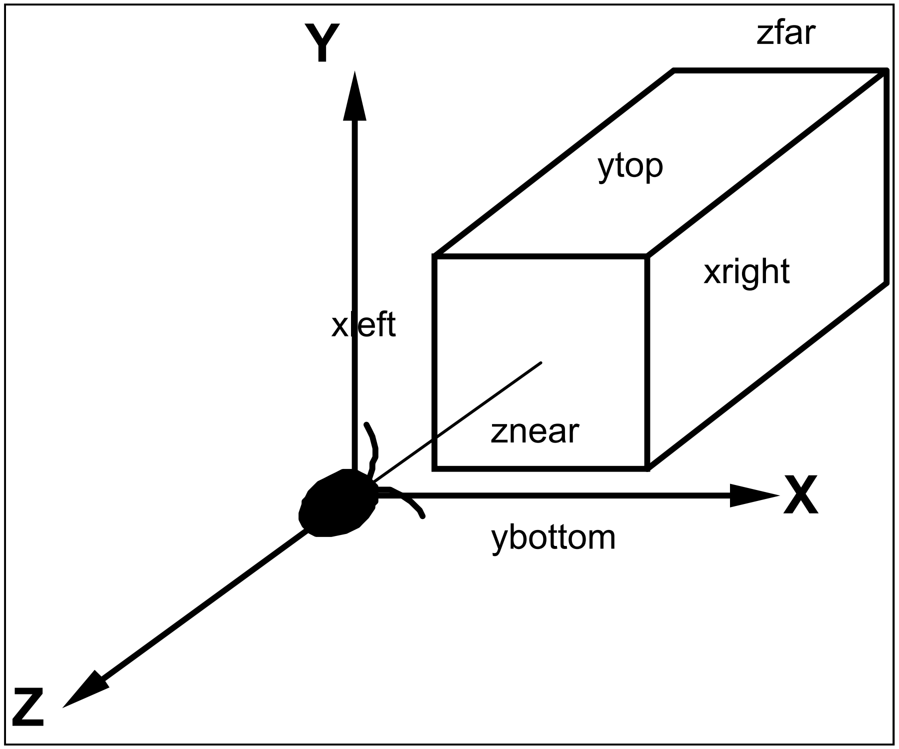
glOrtho(xl, xr, yb, yt, zn, zf); **gluPerspective(fovy, aspect, zn, zf);**

Use one of these, but not both!

```
glMatrixMode( GL_MODELVIEW );
glLoadIdentity( );

gluLookAt( ex, ey, ez,     lx, ly, lz,     ux, uy, uz );

glTranslatef( tx, ty, tz );
glRotatef( degrees,  ax, ay, az );
glScalef( sx, sy, sz );

glColor3f( r, g, b );
glBegin( GL_LINE_STRIP );
        glVertex3f( x0, y0, z0 );
        glVertex3f( x1, y1, z1 );
        glVertex3f( x2, y2, z2 );
        glVertex3f( x3, y3, z3 );
        glVertex3f( x4, y4, z4 );
glEnd( );
```

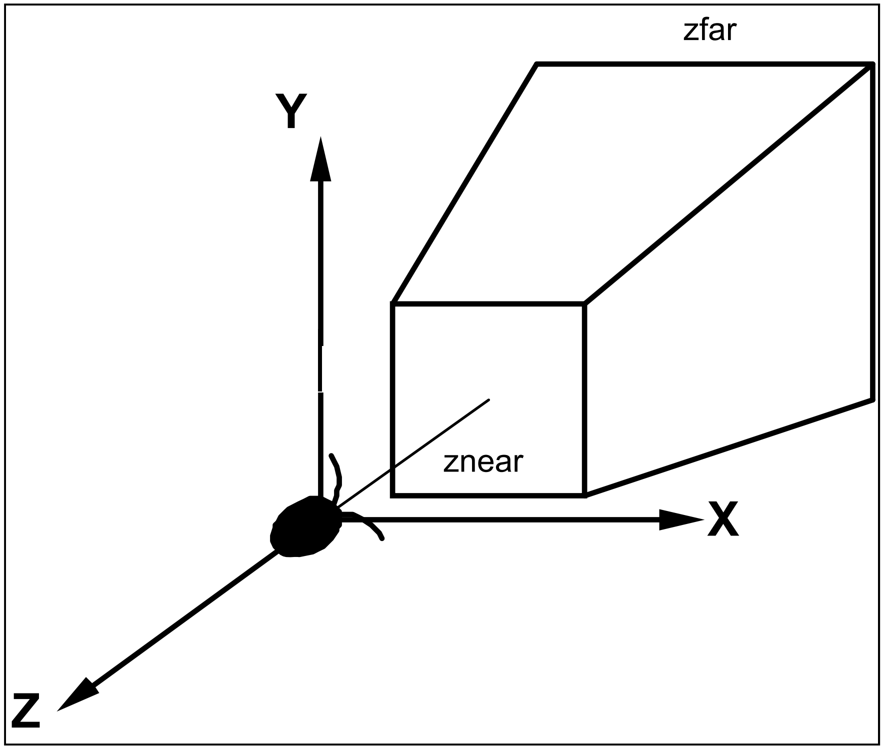
How the Viewing Volumes Look from the Outside

glOrtho(xl, xr, yb, yt, zn, zf);



Parallel/Orthographic

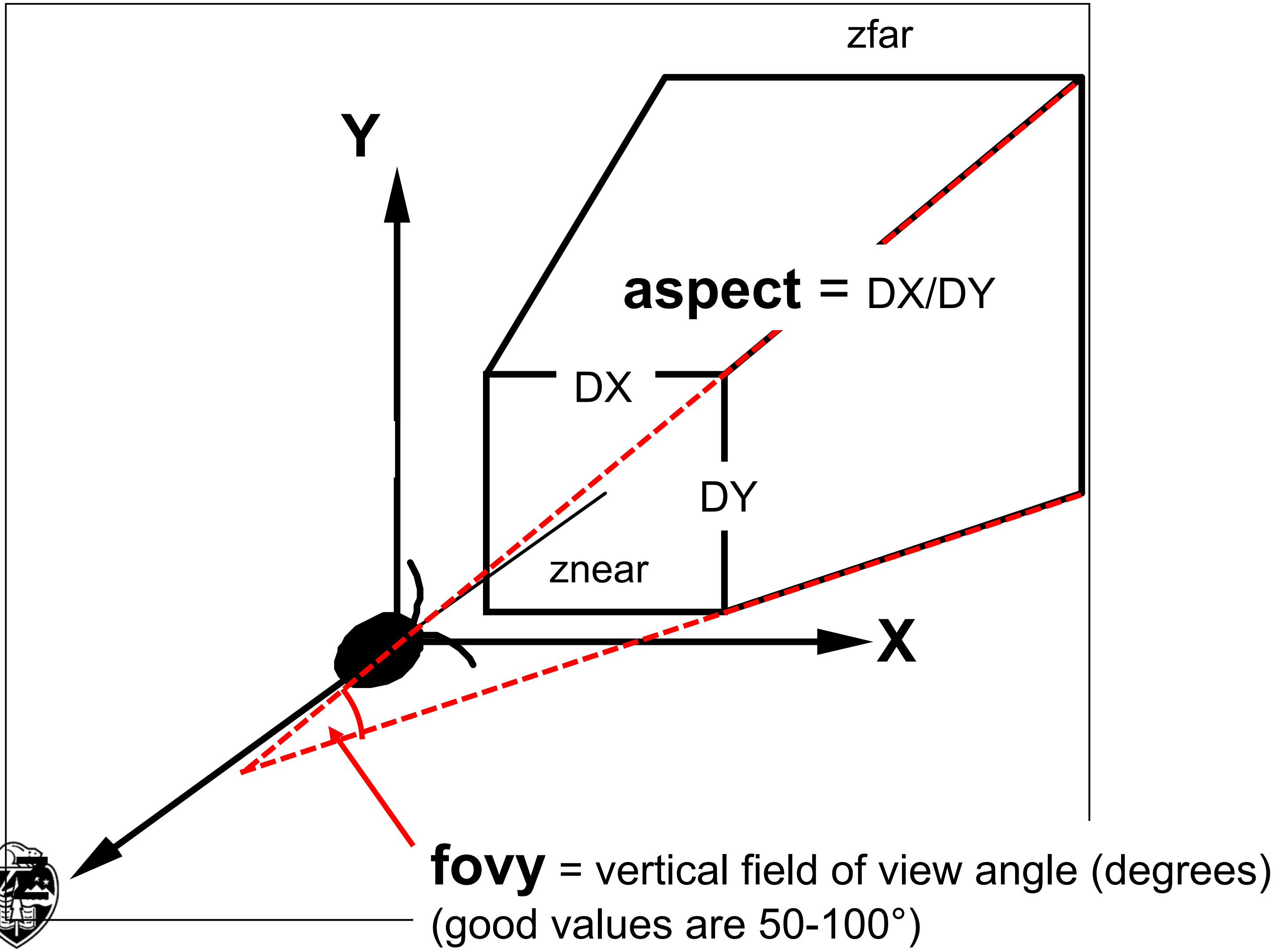
gluPerspective(fovy, aspect, zn, zf);



Perspective

The Perspective Viewing Frustum

gluPerspective(fovy, aspect, zn, zf);



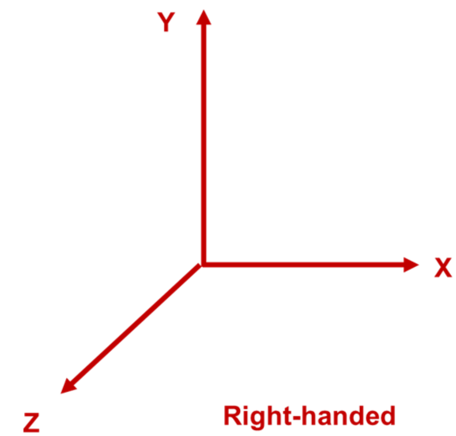
Arbitrary Viewing

```
glMatrixMode( GL_PROJECTION );
glLoadIdentity( );
gluPerspective( fpvy, aspect, zn, zf );

glMatrixMode( GL_MODELVIEW );
glLoadidentity( );
```

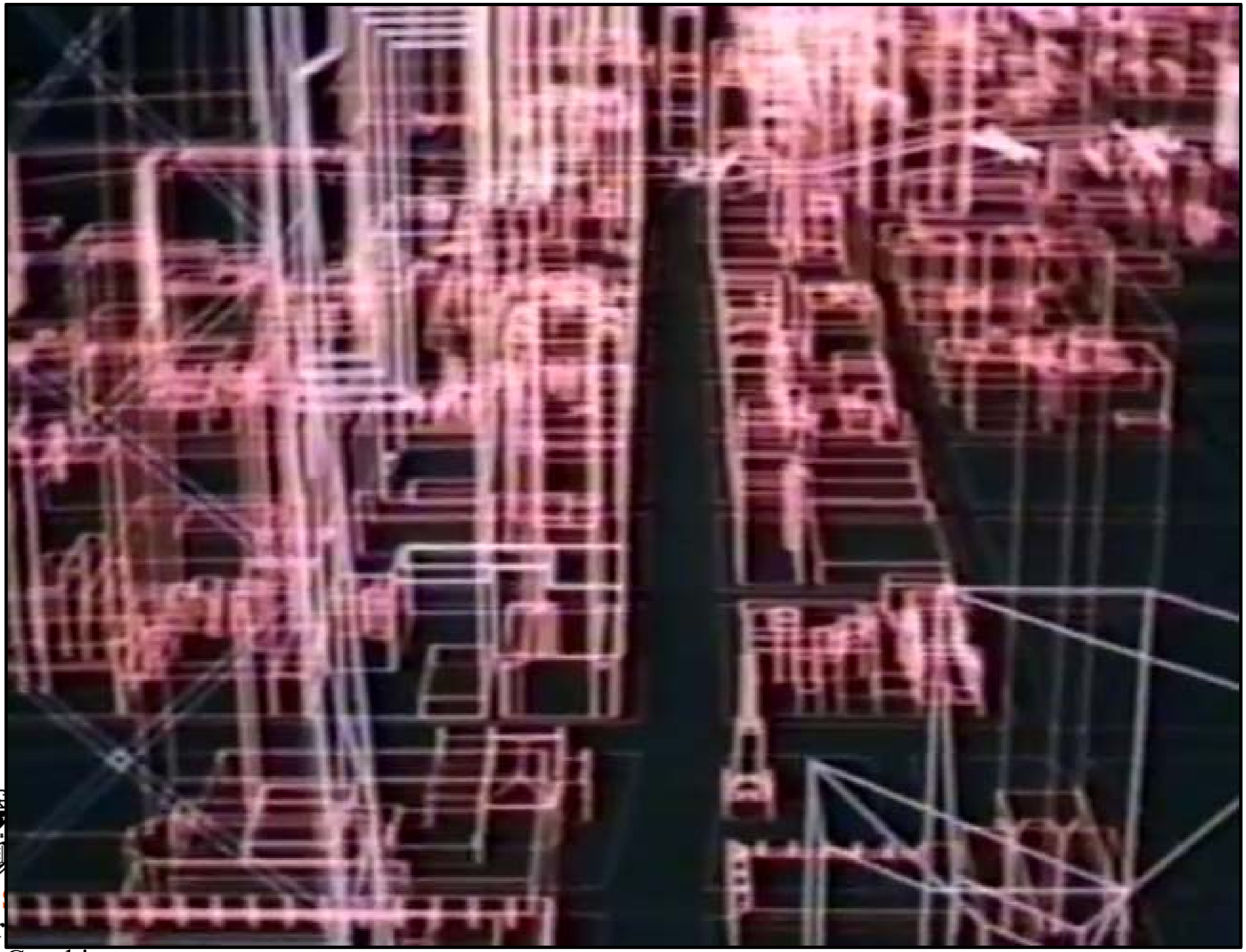
Eye Position / Look-at Position / Up vector

gluLookAt(ex, ey, ez, lx, ly, lz, ux, uy, uz);

```
glTranslatef( tx, ty, tz );
glRotatef( degrees,  ax, ay, az );
glScalef( sx, sy, sz );

glColor3f( r, g, b );
glBegin( GL_LINE_STRIP );
        glVertex3f( x0, y0, z0 );
        glVertex3f( x1, y1, z1 );
        glVertex3f( x2, y2, z2 );
        glVertex3f( x3, y3, z3 );
        glVertex3f( x4, y4, z4 );
glEnd( );
```



Chicago Fly-through:
Changing Eye, Look, and Up



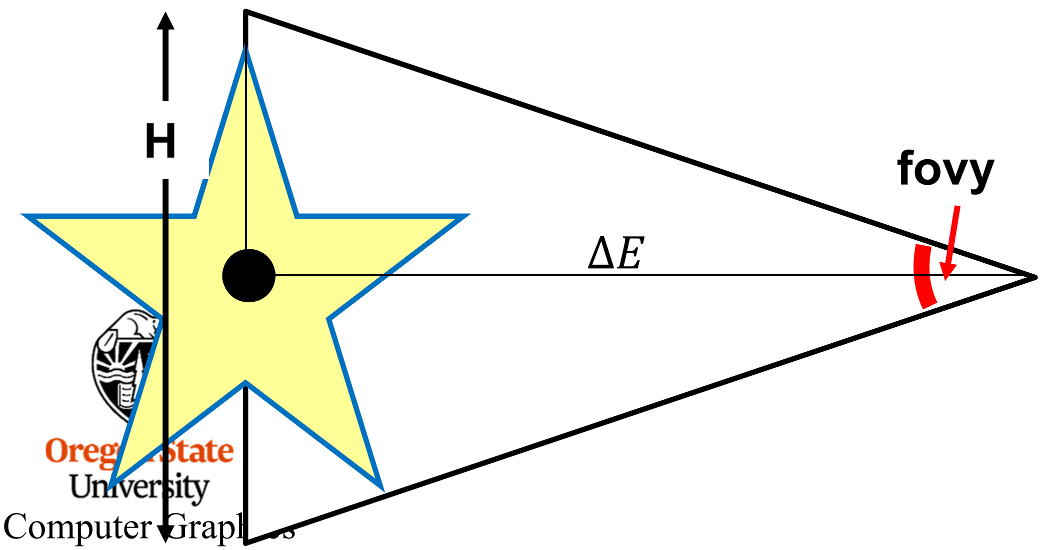
How Can You Be Sure You See Your Scene?

gluPerspective(fovy, aspect, zn, zf);

gluLookAt(ex, ey, ez, lx, ly, lz, ux, uy, uz);

Here's a good way to start:

1. Set **lx,ly,lz** to be the average of all the vertices
2. Set **ux,uy,uz** to be 0.,1.,0.
3. Set **ex=lx** and **ey=ly**
4. Now, you change *ΔE* or *fovy* so that the object fits in the viewing volume:



$$tan(\frac{fovy}{2}) = \frac{H/2}{\Delta E}$$

Giving:

$$fovy = 2arctan\left[\frac{H}{2\Delta E}\right]$$

or:

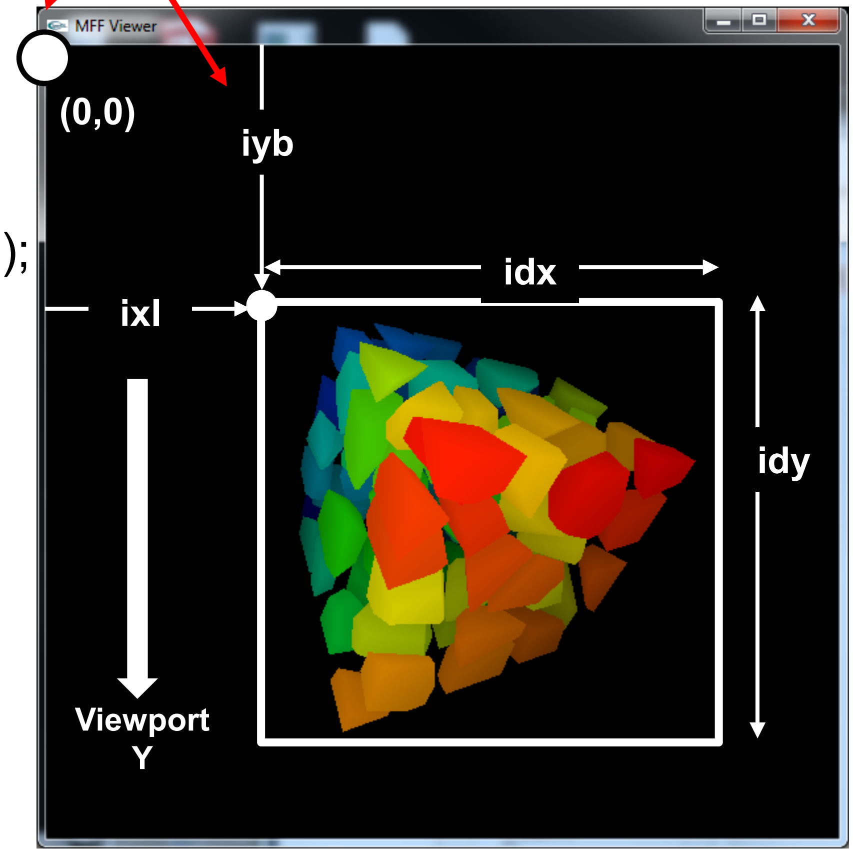
$$\Delta E = \frac{H}{2tan(\frac{fovy}{2})}$$

Be sure the y:x aspect ratios match!!

Specifying a Viewport

Viewports use the upper-left corner as (0,0) and their Y goes down

```
glViewport( ixl, iyb, idx, idy );

glMatrixMode( GL_PROJECTION );
gluPerspective( fovy, aspect, zn, zf );

glMatrixMode( GL_MODELVIEW );
gluLookAt( ex, ey, ez,     lx, ly, lz,     ux, uy, uz );
glTranslatef( tx, ty, tz );
glRotatef( degrees,  ax, ay, az );
glScalef( sx, sy, sz );

glColor3f( r, g, b );
glBegin( GL_LINE_STRIP );
        glVertex3f( x0, y0, z0 );
        glVertex3f( x1, y1, z1 );
        glVertex3f( x2, y2, z2 );
        glVertex3f( x3, y3, z3 );
        glVertex3f( x4, y4, z4 );
glEnd( );
```



Note: setting the viewport is not part of setting either the Modelview or the Projection transformations.

Saving and Restoring the Current Transformation

```
glViewport( ixl, iyb,  idx, idy );

glMatrixMode( GL_PROJECTION );
glLoadidentity( );
gluPerspective( fovy, aspect,  zn, zf );

glMatrixMode( GL_MODELVIEW );
glLoadidentity( );
gluLookAt( ex, ey, ez,     lx, ly, lz,     ux, uy, uz );
glTranslatef( tx, ty, tz );
glPushMatrix( );
glRotatef( degrees,  ax, ay, az );
glScalef( sx, sy, sz );

glColor3f( r, g, b );
glBegin( GL_LINE_STRIP );
        glVertex3f( x0, y0, z0 );
        glVertex3f( x1, y1, z1 );
        glVertex3f( x2, y2, z2 );
        glVertex3f( x3, y3, z3 );
        glVertex3f( x4, y4, z4 );
glEnd( );
glPopMatrix( );
    . . .
```

sample.cpp Program Structure

- #includes
- Consts and #defines
- Global variables
- Function prototypes
- Main program
- InitGraphics function
- Display callback
- Keyboard callback

#includes

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

#define _USE_MATH_DEFINES
#include <math.h>

#ifdef WIN32
#include <windows.h>
#pragma warning(disable:4996)
#include "glew.h"
#endif

#include <GL/gl.h>
#include <GL/glu.h>
#include "glut.h"
```

consts and #defines


```
const char *WINDOWTITLE = { "OpenGL / GLUT Sample -- Joe Graphics" };
const char *GLUITITLE   = { "User Interface Window" };
const int GLUITRUE  = { true  };
const int GLUIFALSE = { false };
#define ESCAPE          0x1b
const int INIT_WINDOW_SIZE = { 600 };
const float BOXSIZE = { 2.f };
const float ANGFACT = { 1. };
const float SCLFACT = { 0.005f };
const float MINSCALE = { 0.05f };
const int LEFT   = { 4 };
const int MIDDLE = { 2 };
const int RIGHT  = { 1 };
enum Projections
{
	ORTHO,
	PERSP
};
enum ButtonVals
{
	RESET,
	QUIT
};
enum Colors
{
	RED,
	YELLOW,
	GREEN,
	CYAN,
	BLUE,
	MAGENTA,
	WHITE,
	BLACK
};
```

consts are always preferred over #defines. But, Visual Studio does not allow consts to be used in case statements or as array sizes.

Initialized Global Variables

```
const GLfloat BACKCOLOR[ ] = { 0., 0., 0., 1. };
const GLfloat AXES_WIDTH   = { 3. };
char * ColorNames[ ] =
{
        "Red",
        "Yellow",
        "Green",
        "Cyan",
        "Blue",
        "Magenta",
        "White",
        "Black"
};
const GLfloat Colors[ ][3] =
{
        { 1., 0., 0. },         // red
        { 1., 1., 0. },         // yellow
        { 0., 1., 0. },         // green
        { 0., 1., 1. },         // cyan
        { 0., 0., 1. },         // blue
        { 1., 0., 1. },         // magenta
        { 1., 1., 1. },         // white
        { 0., 0., 0. },         // black
};
const GLfloat   FOGCOLOR[4] = { .0, .0, .0, 1. };
const GLenum  FOGMODE       = { GL_LINEAR };
const GLfloat   FOGDENSITY  = { 0.30f };
const GLfloat   FOGSTART      = { 1.5 };
const GLfloat   FOGEND          = { 4. };
```

Global Variables

```
int       ActiveButton;      // current button that is down
GLuint    AxesList;          // list to hold the axes
int       AxesOn;            // != 0 means to draw the axes
int       DebugOn;           // != 0 means to print debugging info
int       DepthCueOn;        // != 0 means to use intensity depth cueing
GLuint    BoxList;           // object display list
int       MainWindow;        // window id for main graphics window
float     Scale;             // scaling factor
int       WhichColor;        // index into Colors[ ]
int       WhichProjection;   // ORTHO or PERSP
int       Xmouse, Ymouse;    // mouse values
float     Xrot, Yrot;        // rotation angles in degrees
```

Function Prototypes

```
void    Animate( );
void    Display( );
void    DoAxesMenu( int );
void    DoColorMenu( int );
void    DoDepthMenu( int );
void    DoDebugMenu( int );
void    DoMainMenu( int );
void    DoProjectMenu( int );
void    DoRasterString( float, float, float, char * );
void    DoStrokeString( float, float, float, float, char * );
float   ElapsedSeconds( );
void    InitGraphics( );
void    InitLists( );
void    InitMenus( );
void    Keyboard( unsigned char, int, int );
void    MouseButton( int, int, int, int );
void    MouseMotion( int, int );
void    Reset( );
void    Resize( int, int );
void    Visibility( int );

void    Axes( float );
void    HsvRgb( float[3], float [3] );
```

Main Program

```
int
main( int argc, char *argv[ ] )
{
	// turn on the glut package:
	// (do this before checking argc and argv since it might
	// pull some command line arguments out)

	glutInit( &argc, argv );

	// setup all the graphics stuff:

	InitGraphics( );

	// create the display structures that will not change:

	InitLists( );

	// init all the global variables used by Display( ):
	// this will also post a redisplay

	Reset( );

	// setup all the user interface stuff:

	InitMenus( );

	// draw the scene once and wait for some interaction:
	// (this will never return)
	glutSetWindow( MainWindow );
	glutMainLoop( );

	// this is here to make the compiler happy:

	return 0;
}
```

InitGraphics( ), I

```
void
InitGraphics( )
{
	// request the display modes:
	// ask for red-green-blue-alpha color, double-buffering, and z-buffering:

	glutInitDisplayMode( GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH );

	// set the initial window configuration:

	glutInitWindowPosition( 0, 0 );
	glutInitWindowSize( INIT_WINDOW_SIZE, INIT_WINDOW_SIZE );

	// open the window and set its title:

	MainWindow = glutCreateWindow( WINDOWTITLE );
	glutSetWindowTitle( WINDOWTITLE );

	// set the framebuffer clear values:

	glClearColor( BACKCOLOR[0], BACKCOLOR[1], BACKCOLOR[2], BACKCOLOR[3] );

	glutSetWindow( MainWindow );
	glutDisplayFunc( Display );
	glutReshapeFunc( Resize );
	glutKeyboardFunc( Keyboard );
	glutMouseFunc( MouseButton );
	glutMotionFunc( MouseMotion );
	glutTimerFunc( -1, NULL, 0 );
	glutIdleFunc( NULL );
```

InitGraphics( ), II

```
#ifdef WIN32

        GLenum err = glewInit( );
         if( err != GLEW_OK )
         {
                fprintf( stderr, "glewInit Error\n" );
         }

#endif


}
```

Display( ), I

```
void
Display( )
{
	// set which window we want to do the graphics into:

	glutSetWindow( MainWindow );

	// erase the background:

	glDrawBuffer( GL_BACK );
	glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
	glEnable( GL_DEPTH_TEST );

	// specify shading to be flat:

	glShadeModel( GL_FLAT );

	// set the viewport to a square centered in the window:

	GLsizei vx = glutGet( GLUT_WINDOW_WIDTH );
	GLsizei vy = glutGet( GLUT_WINDOW_HEIGHT );
	GLsizei v = vx < vy ? vx : vy;			// minimum dimension
	GLint xl = ( vx - v ) / 2;
	GLint yb = ( vy - v ) / 2;
	glViewport( xl, yb,  v, v );
```

Display( ), II

```
// set the viewing volume:
// remember that the Z clipping  values are actually
// given as DISTANCES IN FRONT OF THE EYE

glMatrixMode( GL_PROJECTION );
glLoadIdentity( );
if( WhichProjection == ORTHO )
	glOrtho( -3., 3.,     -3., 3.,     0.1, 1000. );
else
	gluPerspective( 90., 1.,        0.1, 1000. );

// place the objects into the scene:

glMatrixMode( GL_MODELVIEW );
glLoadIdentity( );

// set the eye position, look-at position, and up-vector:

gluLookAt( 0., 0., 3.,     0., 0., 0.,     0., 1., 0. );

// rotate the scene:

glRotatef( (GLfloat)Yrot, 0., 1., 0. );
glRotatef( (GLfloat)Xrot, 1., 0., 0. );

// uniformly scale the scene:

if( Scale < MINSCALE )
	Scale = MINSCALE;
glScalef( (GLfloat)Scale, (GLfloat)Scale, (GLfloat)Scale );
```

Display( ), III

```
// set the fog parameters:

if( DepthCueOn != 0 )
{
        glFogi( GL_FOG_MODE, FOGMODE );
        glFogfv( GL_FOG_COLOR, FOGCOLOR );
        glFogf( GL_FOG_DENSITY, FOGDENSITY );
        glFogf( GL_FOG_START, FOGSTART );
        glFogf( GL_FOG_END, FOGEND );
        glEnable( GL_FOG );
}
else
{
        glDisable( GL_FOG );
}

// possibly draw the axes:

if( AxesOn != 0 )
{
        glColor3fv( &Colors[WhichColor][0] );
        glCallList( AxesList );
}

// draw the current object:

glCallList( BoxList );
```

Replay the graphics commands from a previously-stored Display List.

Display Lists have their own noteset.

Display( ), IV

```
// draw some gratuitous text that just rotates on top of the scene:

glDisable( GL_DEPTH_TEST );
glColor3f( 0., 1., 1. );
DoRasterString( 0., 1., 0., "Text That Moves" );


// draw some gratuitous text that is fixed on the screen:
// the projection matrix is reset to define a scene whose
// world coordinate system goes from 0-100 in each axis
// this is called "percent units", and is just a convenience
// the modelview matrix is reset to identity as we don't
// want to transform these coordinates

glDisable( GL_DEPTH_TEST );
glMatrixMode( GL_PROJECTION );
glLoadIdentity( );
gluOrtho2D( 0., 100.,     0., 100. );
glMatrixMode( GL_MODELVIEW );
glLoadIdentity( );
glColor3f( 1., 1., 1. );
DoRasterString( 5., 5., 0., "Text That Doesn't" );

// swap the double-buffered framebuffers:
```

glutSwapBuffers( );

```
// be sure the graphics buffer has been sent:
// note: be sure to use glFlush( ) here, not glFinish( ) !

glFlush( );
}
```

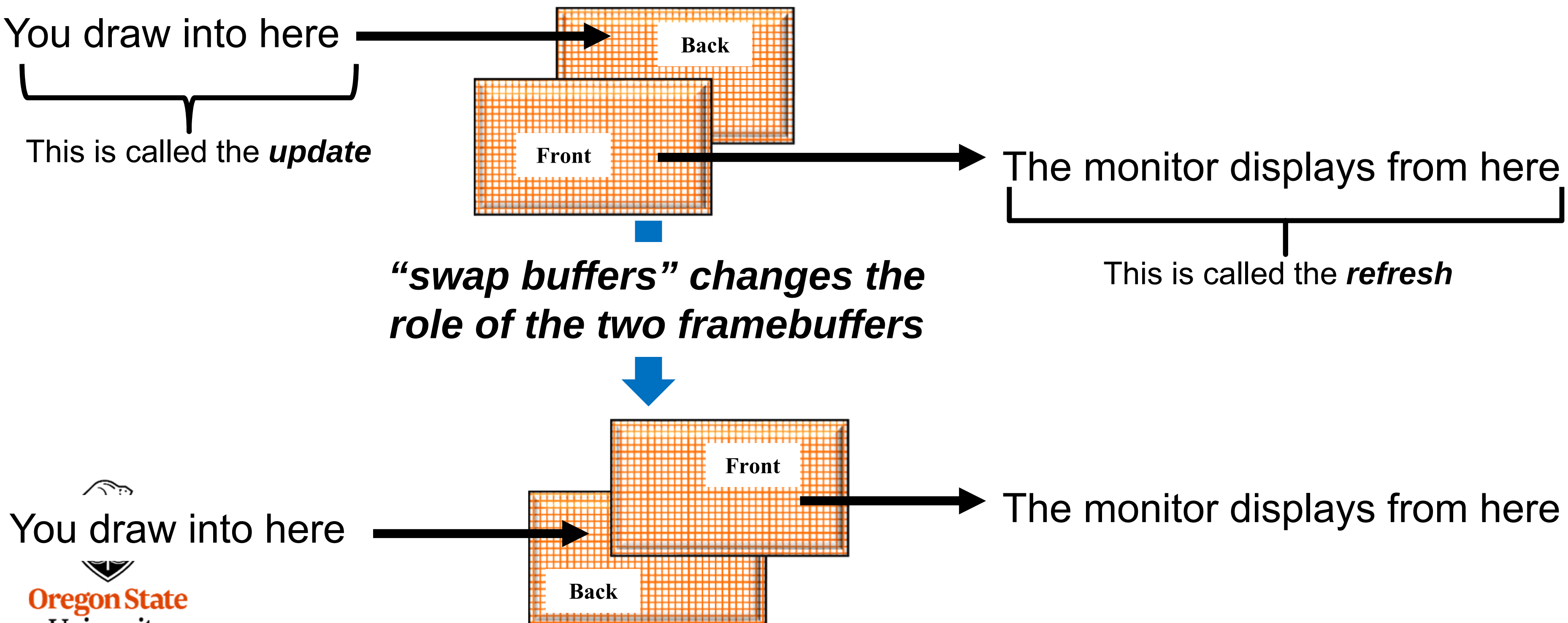
(x,y,z), to be *translated* by the ModelView matrix

glutSwapBuffers( )

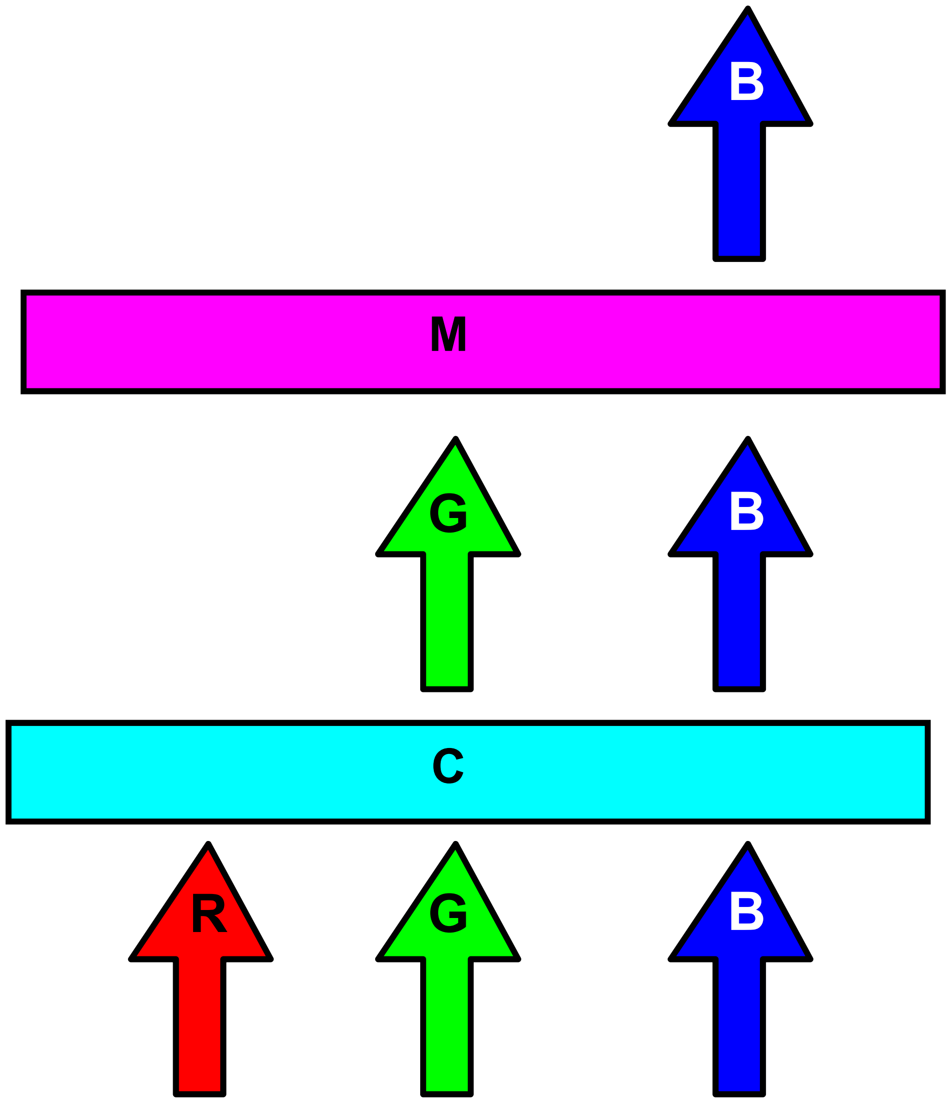
```
// swap the double-buffered framebuffers:

glutSwapBuffers( );
```

```
glutInitDisplayMode( GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH );
```

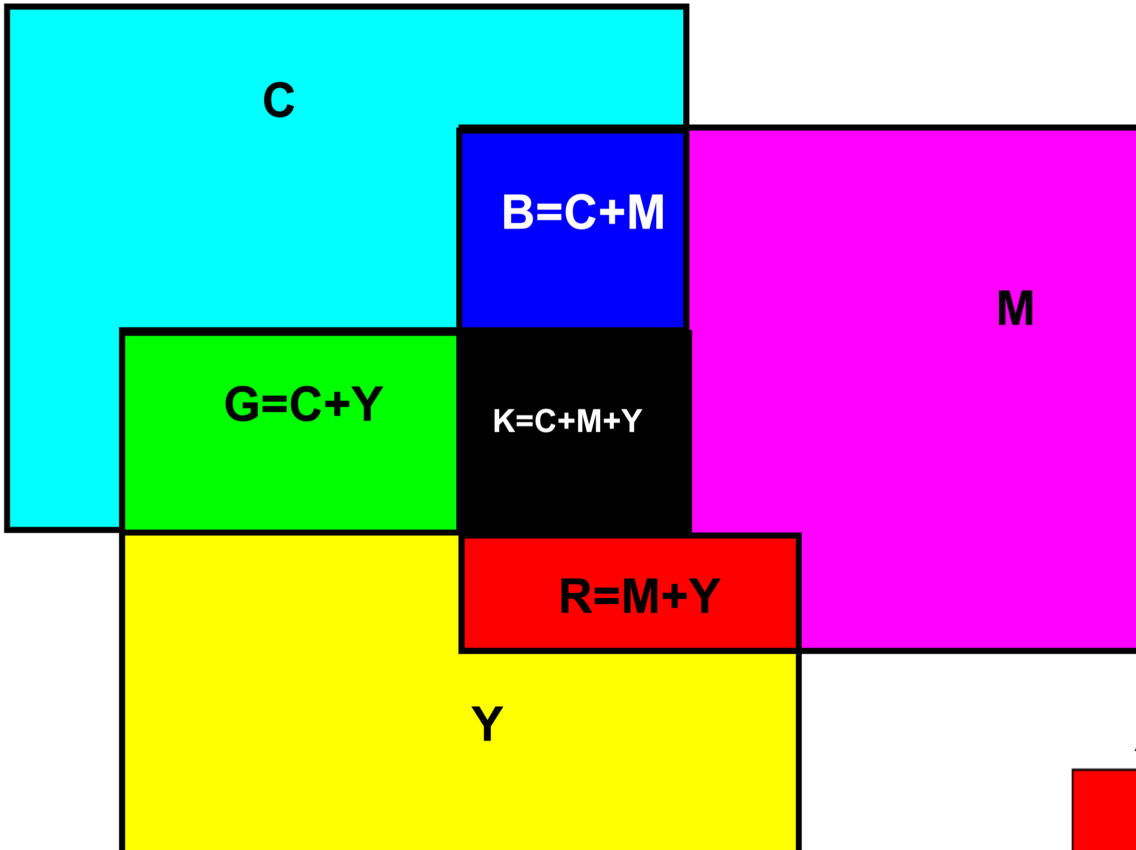
```
glDrawBuffer( GL_BACK );
```



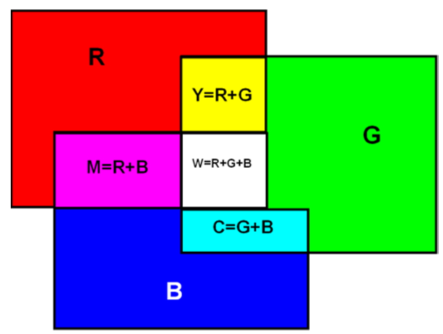
Extra Topics:
(Don’t need to get started with OpenGL programming)
Subtractive Colors (CMYK)



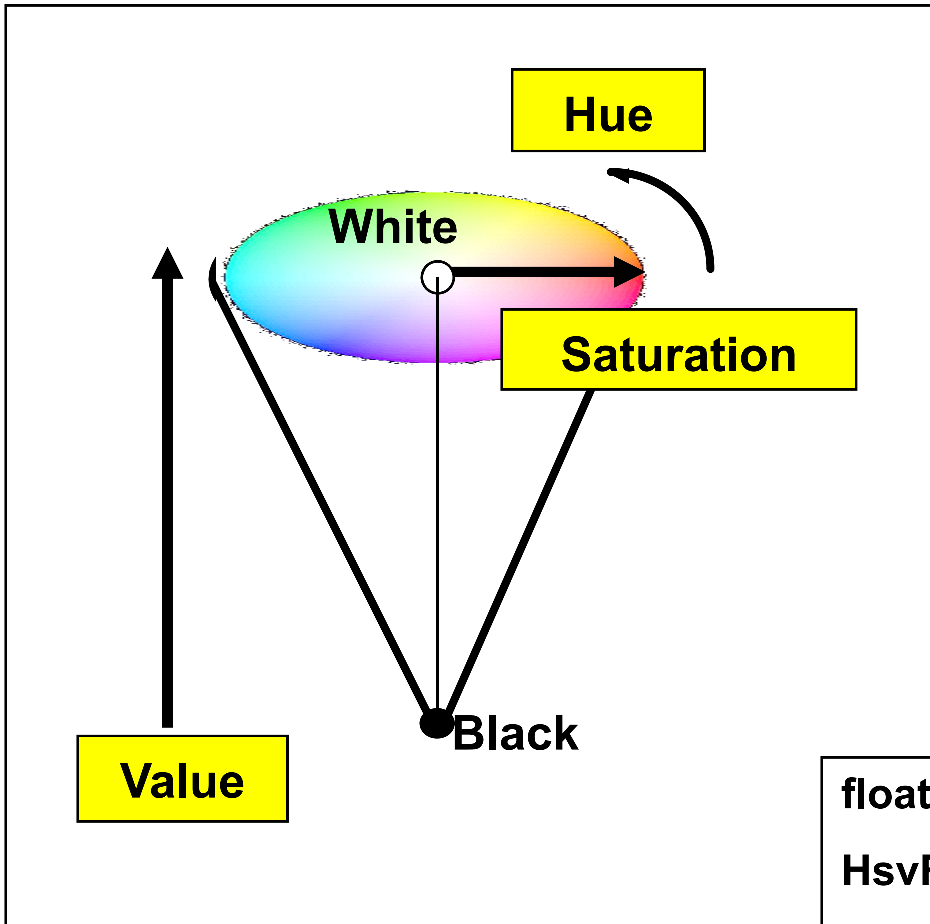
Sidebar: Subtractive Colors (CMYK)

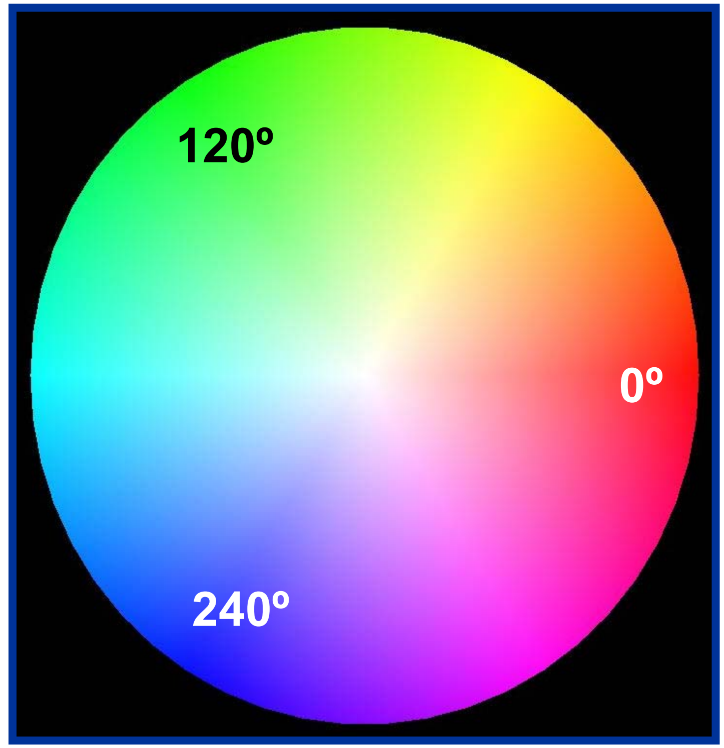


Additive Colors:



Sidebar: Hue-Saturation-Value (HSV) -- Another way to specify additive color



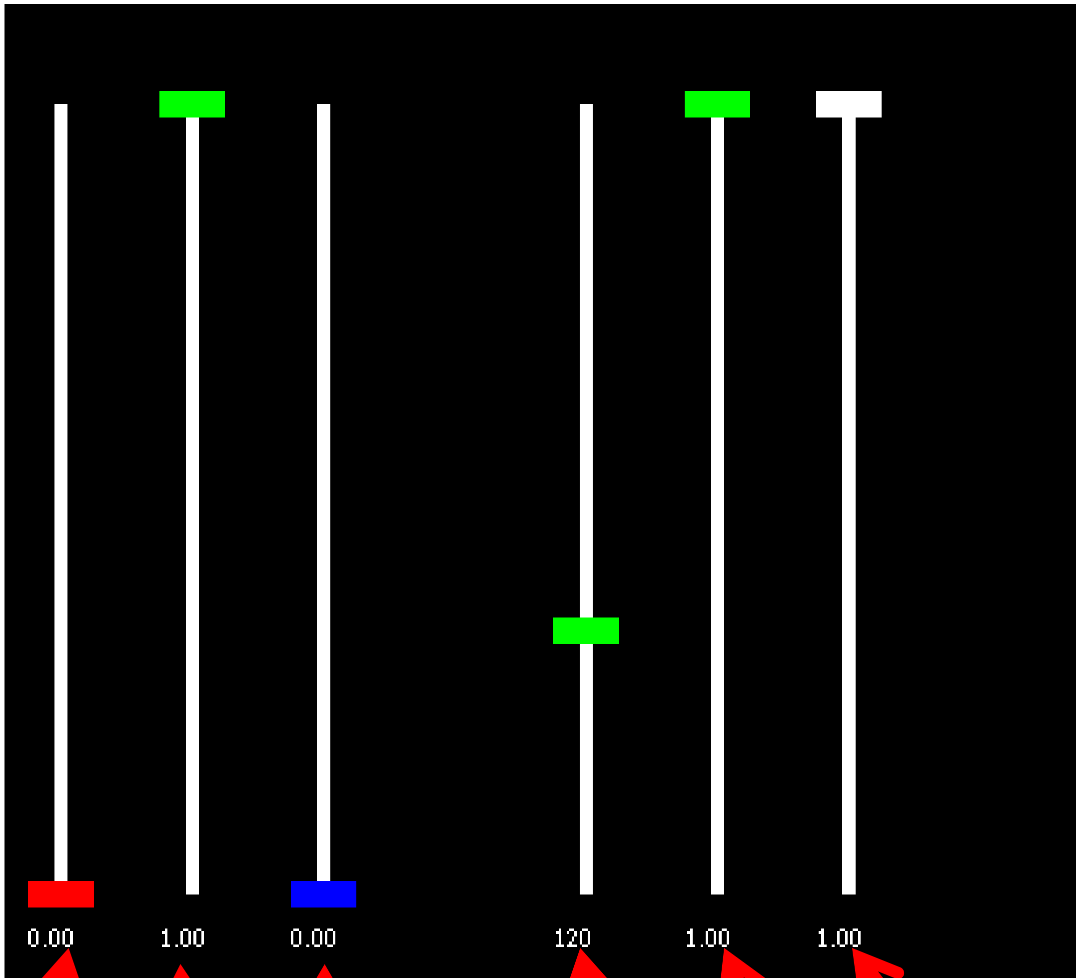


The HsvRgb function is in your sample code

```
float hsv[3], rgb[3];
HsvRgb( hsv, rgb );
glColor3fv( rgb );
```

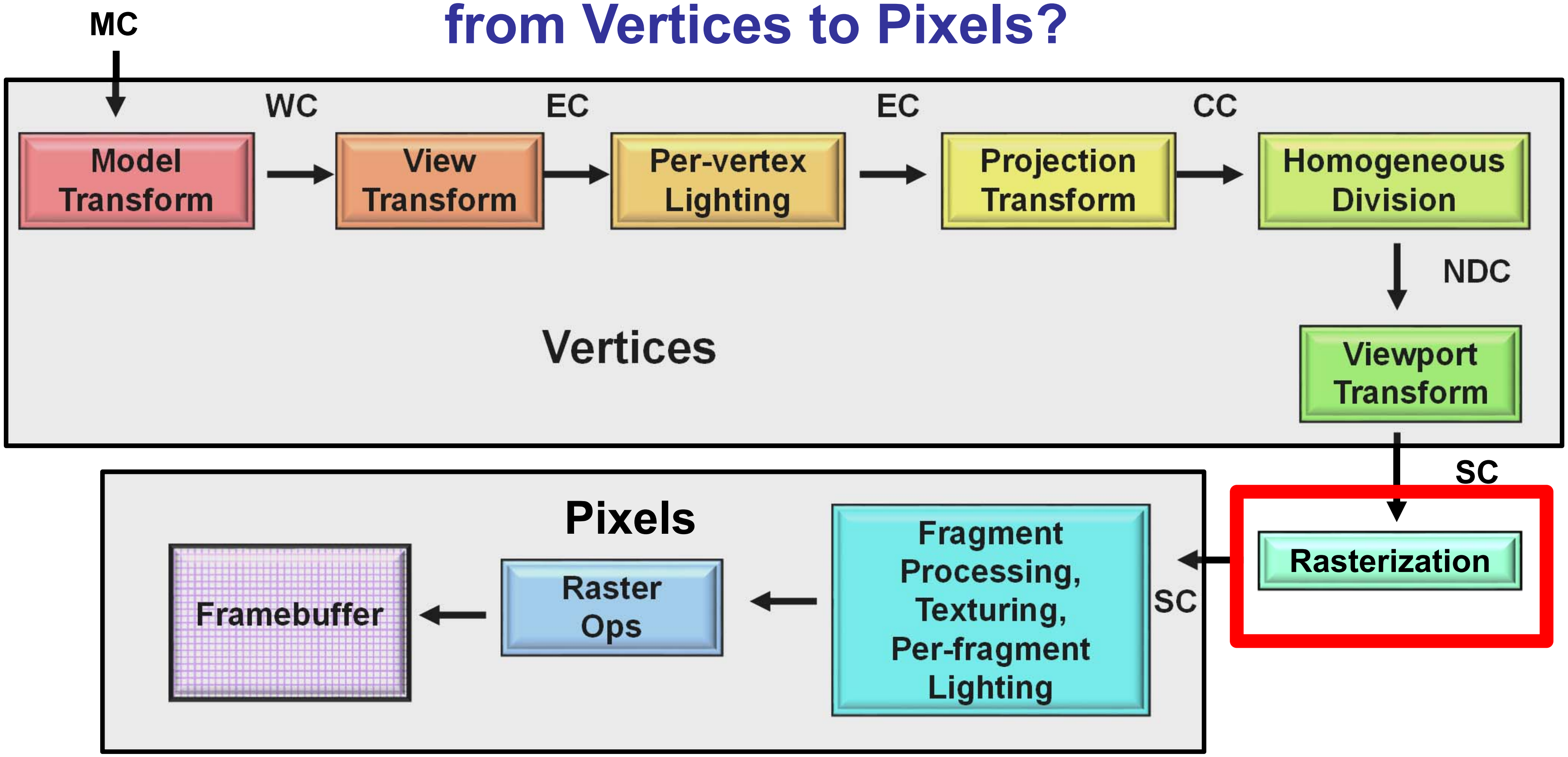
$0. \le s, v, r, g, b \le 1.$
$0. \le h \le 360.$

The OSU *ColorPicker* Program



Red, Green, Blue **Hue, Saturation, Value**

Sidebar: How Did We Make the Transition from Vertices to Pixels?



Vertices

MC = Model Coordinates
WC = World Coordinates
EC = Eye Coordinates
CC = Clip Coordinates
NDC = Normalized Device Coordinates

Pixels

SC = Screen Coordinates

Sidebar: How Did We Make the Transition from Vertices to Pixels?

There is a piece of hardware called the **Rasterizer**. Its job is to interpolate a line or polygon, defined by vertices, into a collection of **fragments**. Think of it as filling in squares on graph paper.

A fragment is a "pixel-to-be". In computer graphics, "pixel" is defined as having its full RGBA already computed. A fragment does not yet but all of the information needed to compute the RGBA is there.

A fragment is turned into a pixel by the **fragment processing** operation.

In CS 457/557, you will do some pretty snazzy things with your own fragment processing code!