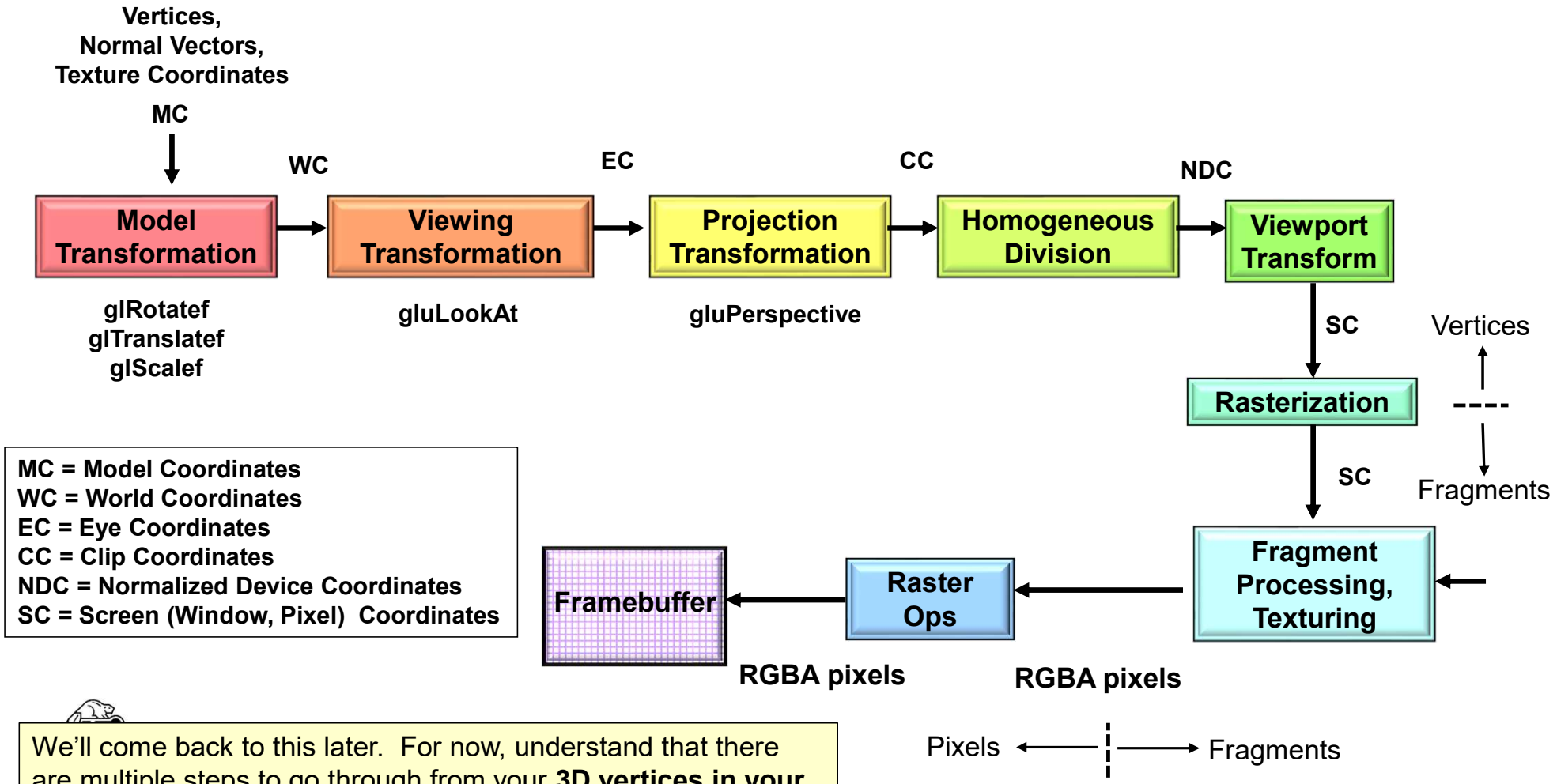
Getting Started with OpenGL Graphics Programming in C/C++

Mike Bailey

mjb@cs.oregonstate.edu

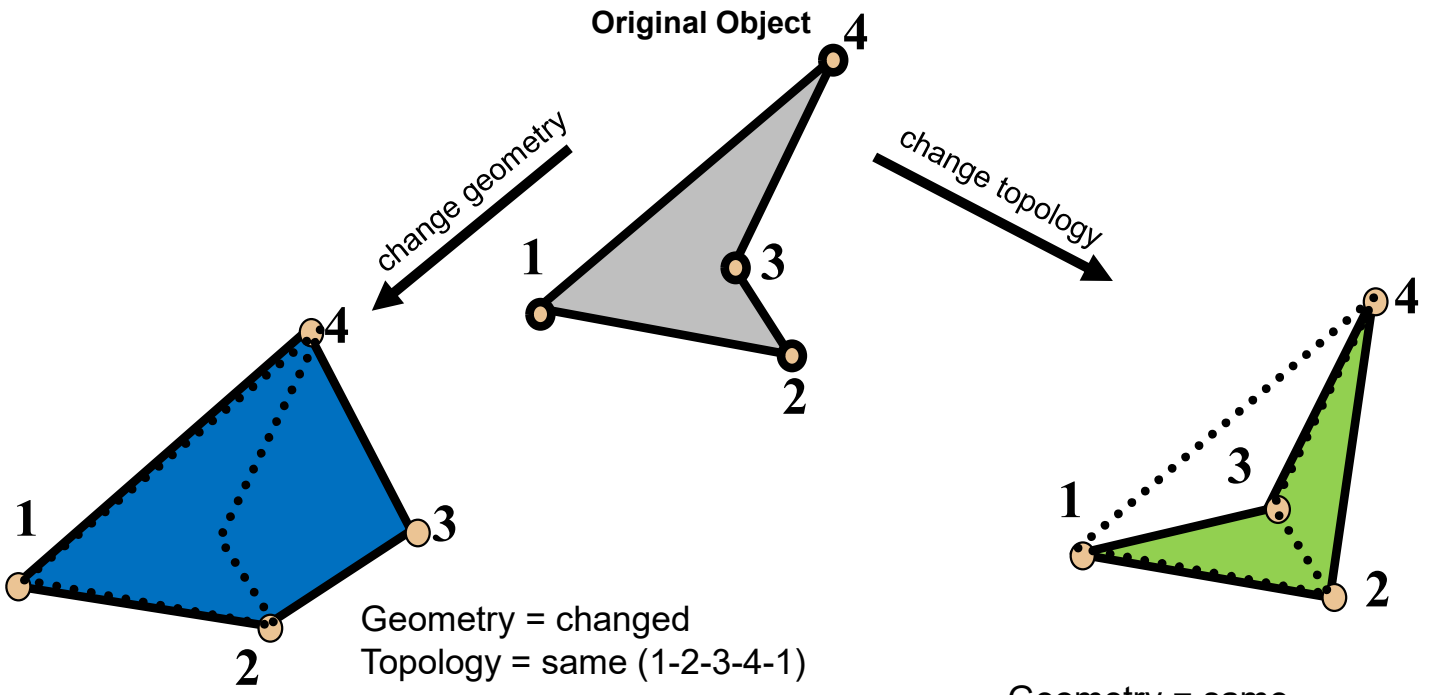
Oregon State
University
Computer Graphics

The Basic Computer Graphics Pipeline, *OpenGL-style*



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

Geometry vs. Topology



Geometry = changed
Topology = same (1-2-3-4-1)

Geometry = same
Topology = changed (1-2-4-3-1)

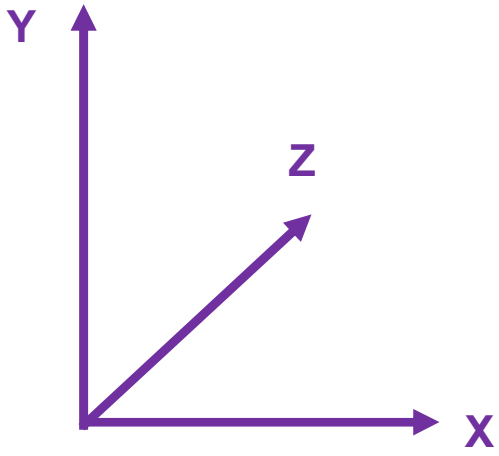
Geometry:

Where things are (e.g., coordinates)

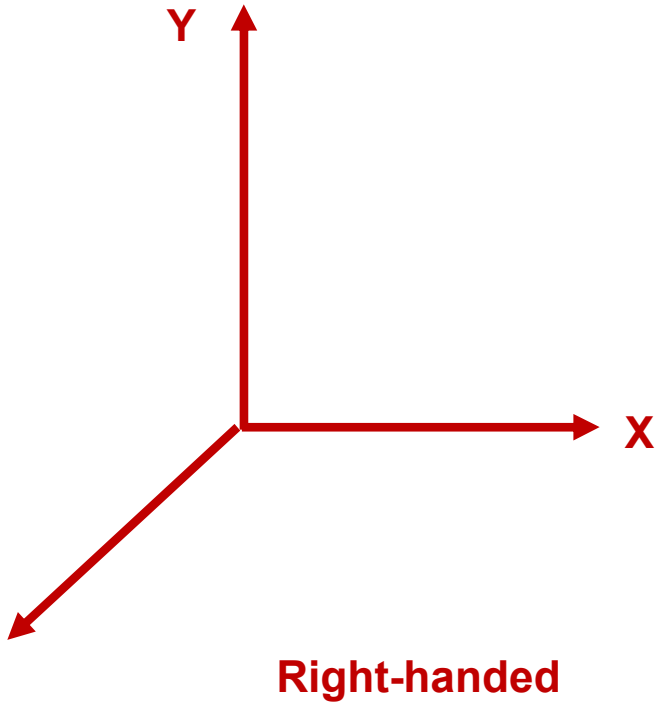
Topology:

How things are connected

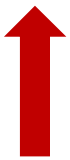
3D Coordinate Systems



Left-handed

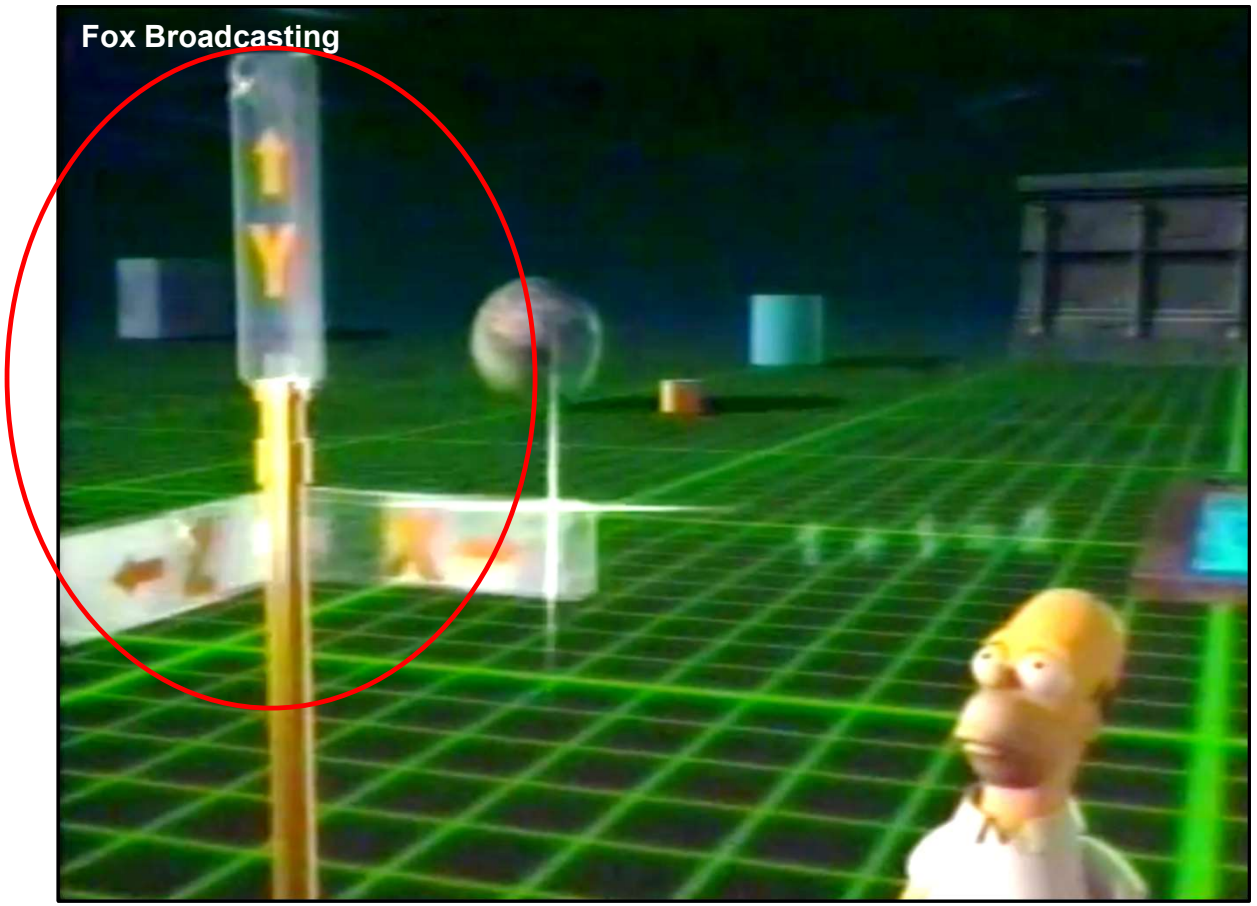


Right-handed

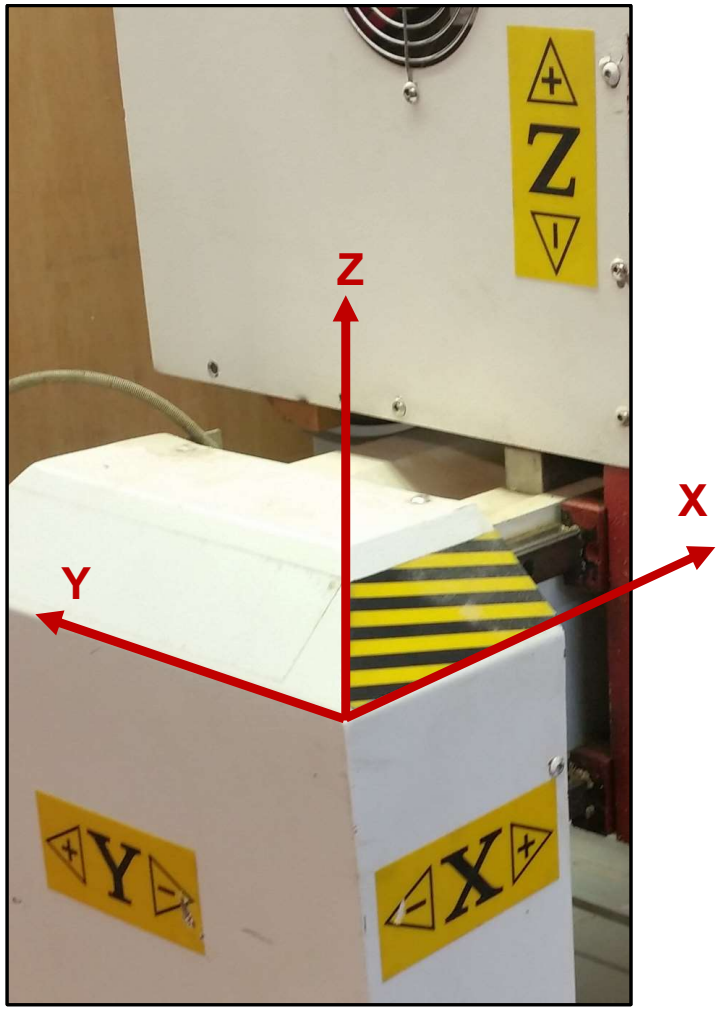


OpenGL uses this one

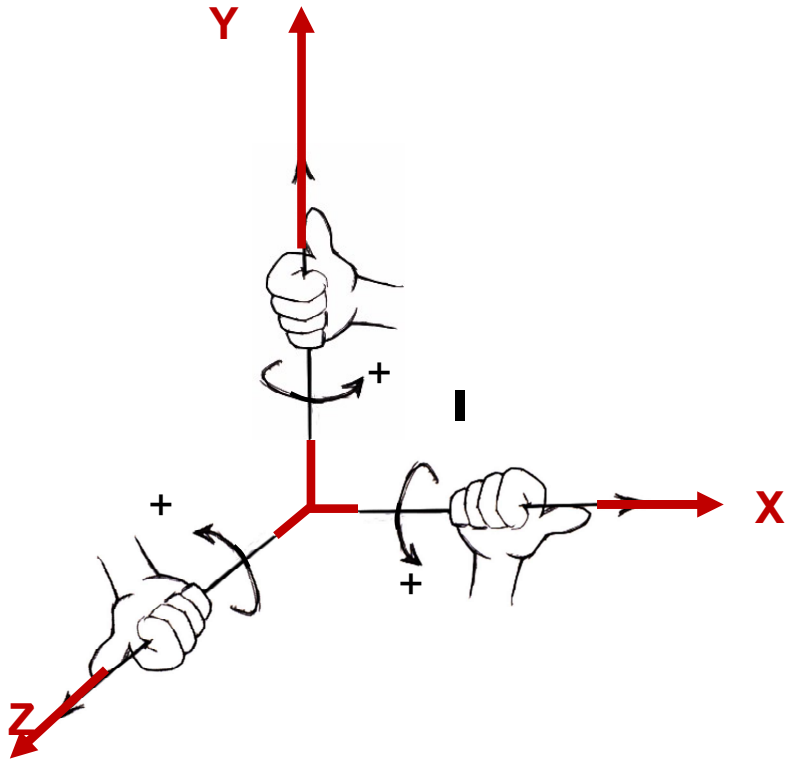
Homer Simpson uses Right-handed Coordinates. Who are we to argue with Homer Simpson? ☺



Right-handed 3D Coordinate System for a CNC Machine



Right-handed Positive Rotations



Right-Handed Coordinate System

Drawing in 3D

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

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

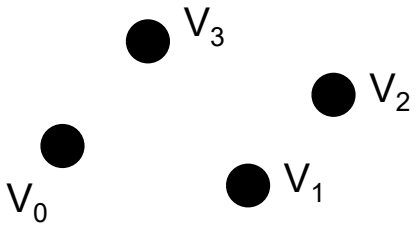
Begin the drawing. Use the current state's display-characteristics. Here is the topology to be used with these vertices

```
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( );
```

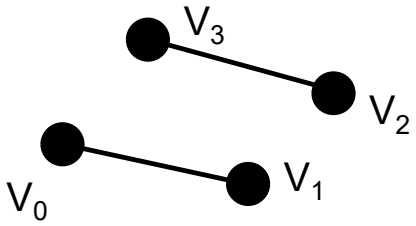
This is a wonderfully understandable way to start with 3D graphics – it is like holding a marker in your hand and sweeping out linework in the 3D air in front of you!
But it is also incredibly internally *inefficient*! We'll talk about that later and what to do about it…

OpenGL Topologies

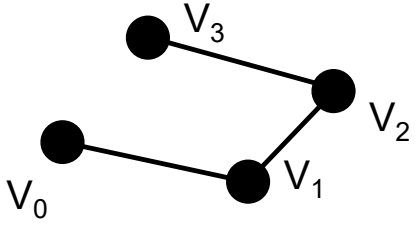
GL_POINTS



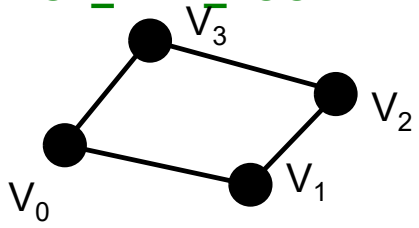
GL_LINES



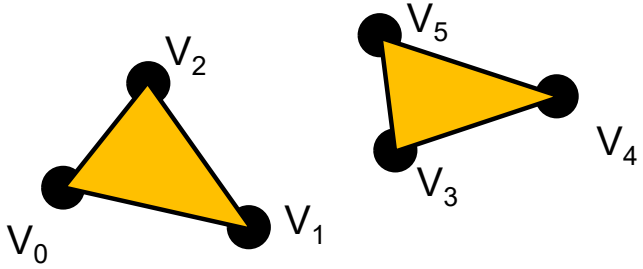
GL_LINE_STRIP



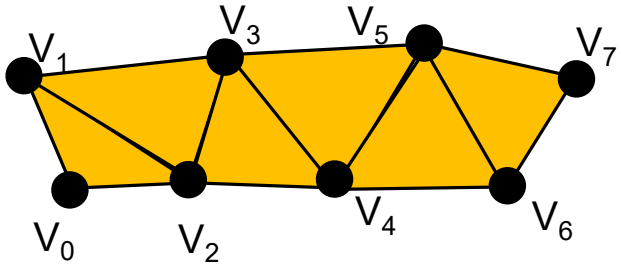
GL_LINE_LOOP



GL_TRIANGLES

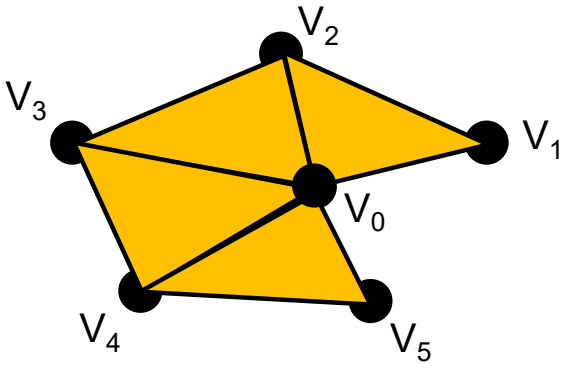


GL_TRIANGLE_STRIP

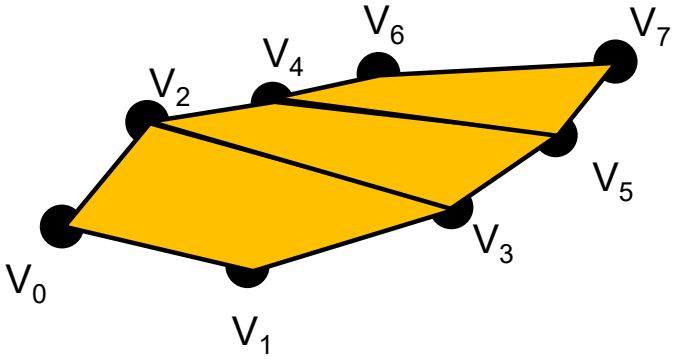


OpenGL Topologies

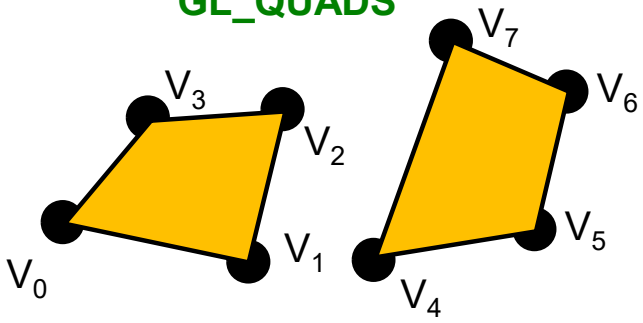
GL_TRIANGLE_FAN



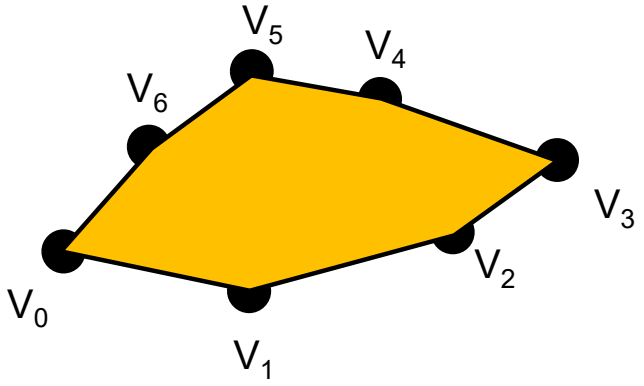
GL_QUAD_STRIP



GL_QUADS



GL_POLYGON



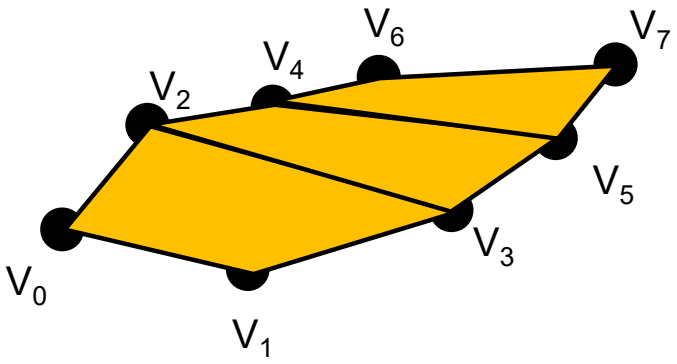
OpenGL Topologies – Polygon Requirements

Polygons must be:

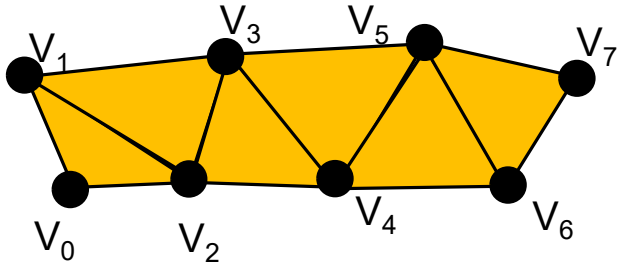
- **Convex** and
- **Planar**

GL_TRIANGLE_STRIP and GL_TRIANGLES are considered to be preferable to GL_QUAD_STRIP and GL_QUADS. GL_POLYGON is rarely used.

GL_QUAD_STRIP



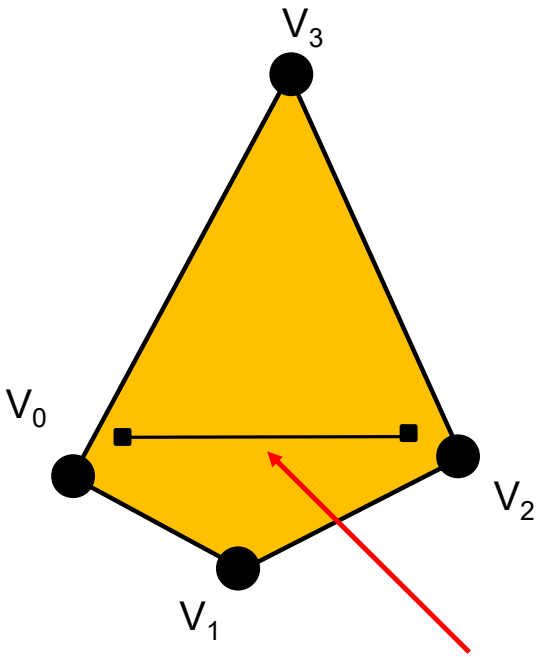
GL_TRIANGLE_STRIP



What does "Convex Polygon" Mean?

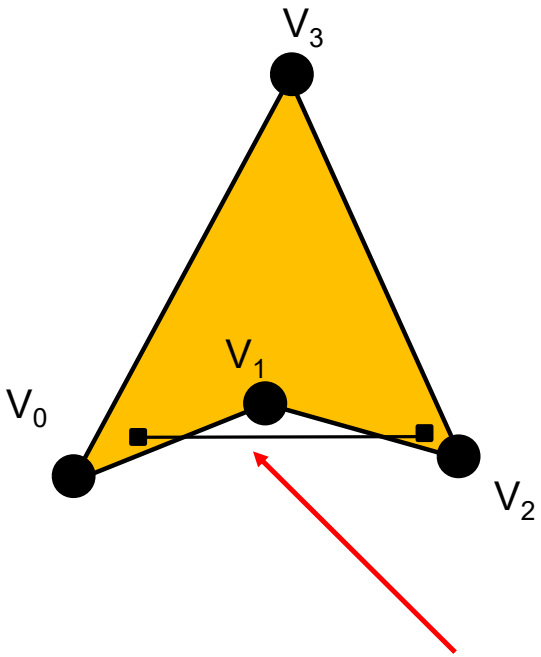
We can go all mathematical here, but let's go visual instead. In a convex polygon, a line between ***any*** two points inside the polygon never leaves the inside of the polygon.

Convex



Stays within the polygon

Not Convex

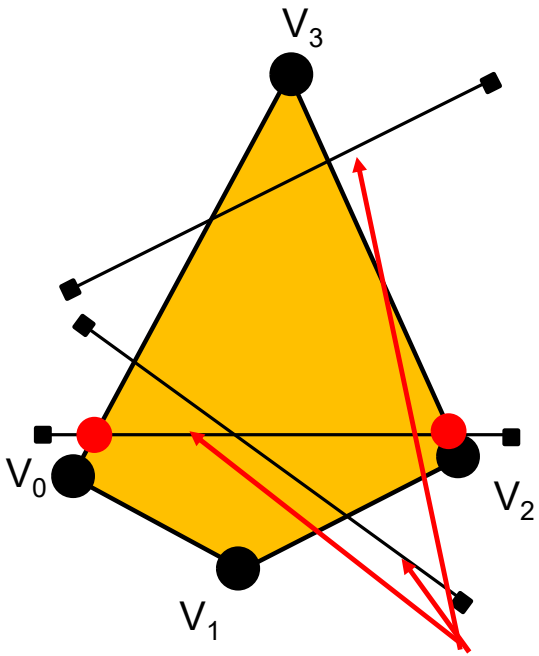


Leaves the polygon

Why is there a Requirement for Polygons to be Convex?

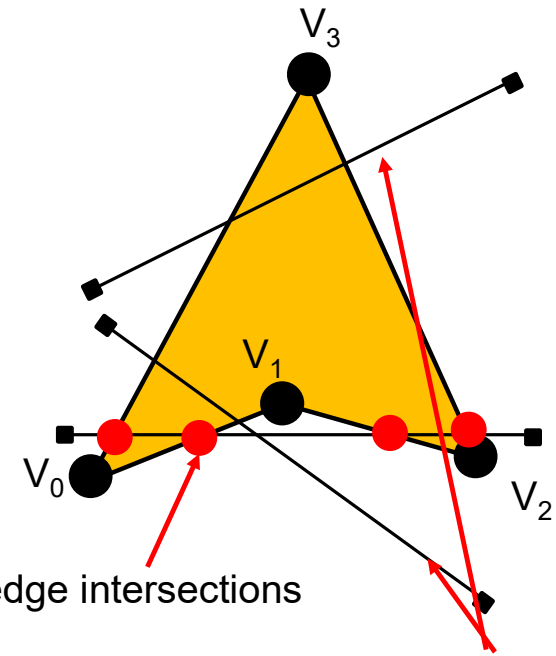
Graphics polygon-filling hardware can be highly optimized if you know that, no matter what direction you fill the polygon in, there will be two and only two intersections between the scanline and the polygon's edges

Convex



2 edge intersections

Not Convex



4 edge intersections

2 edge intersections

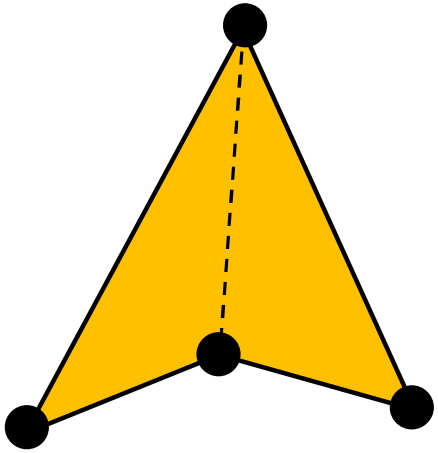
What if you need to display Polygons that are not Convex?

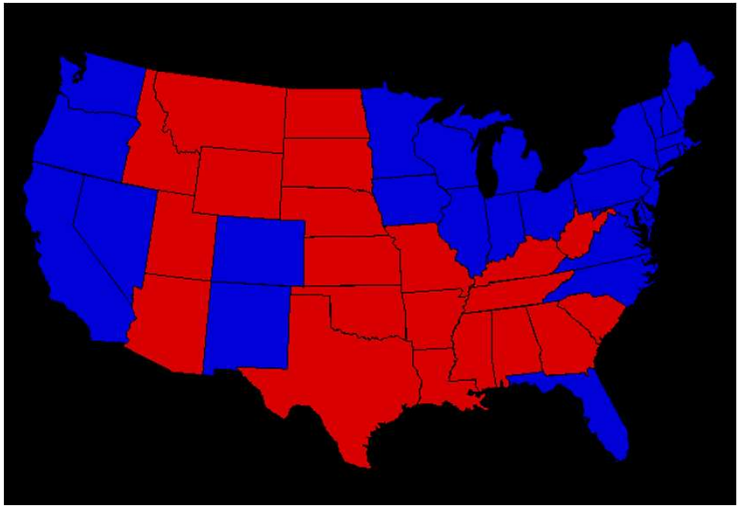
There are two good solutions I know of (and there are probably more):

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

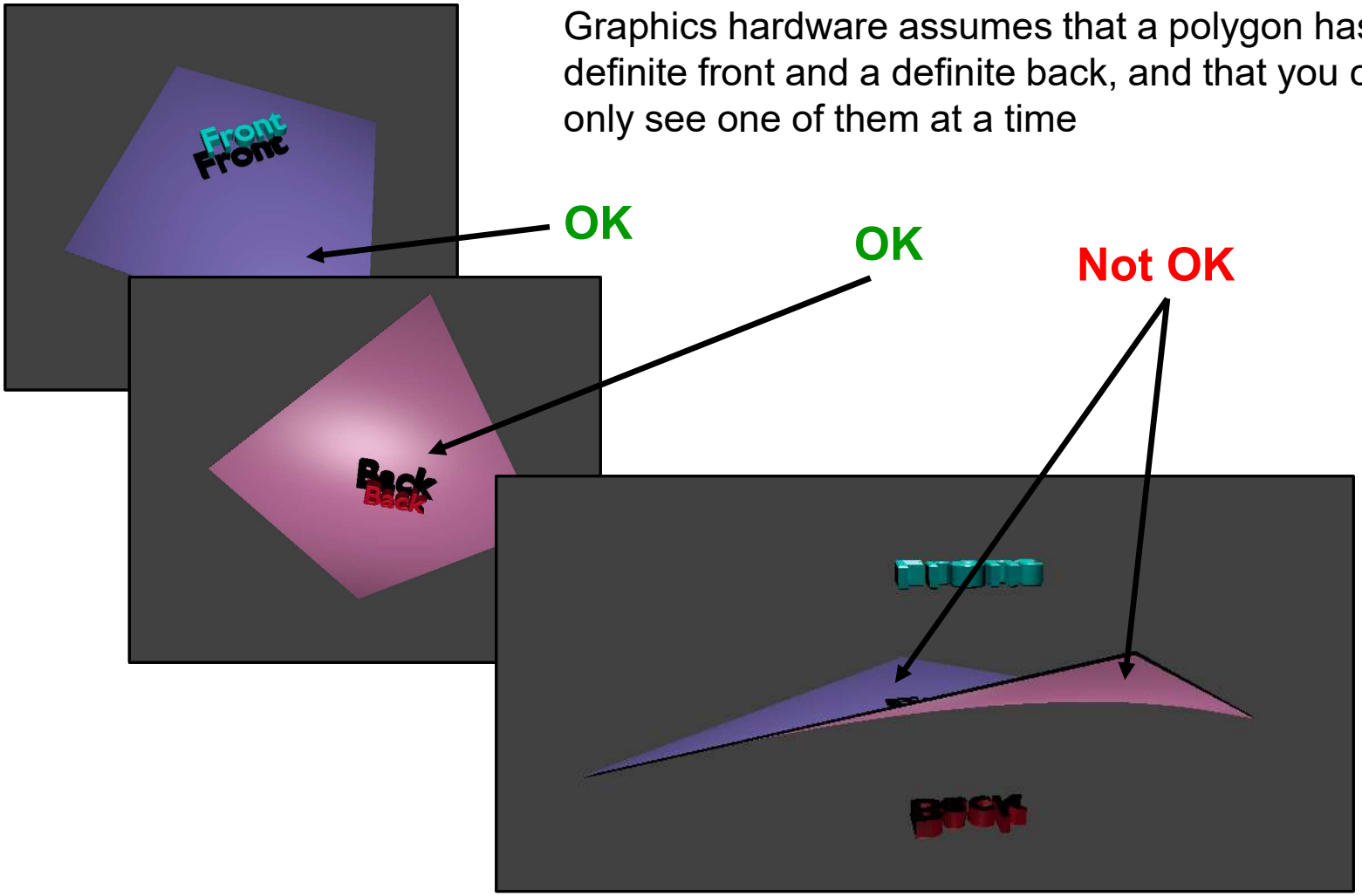




If you ever need to do this, contact me. I have working code for each approach…

Why is there a Requirement for Polygons to be Planar?

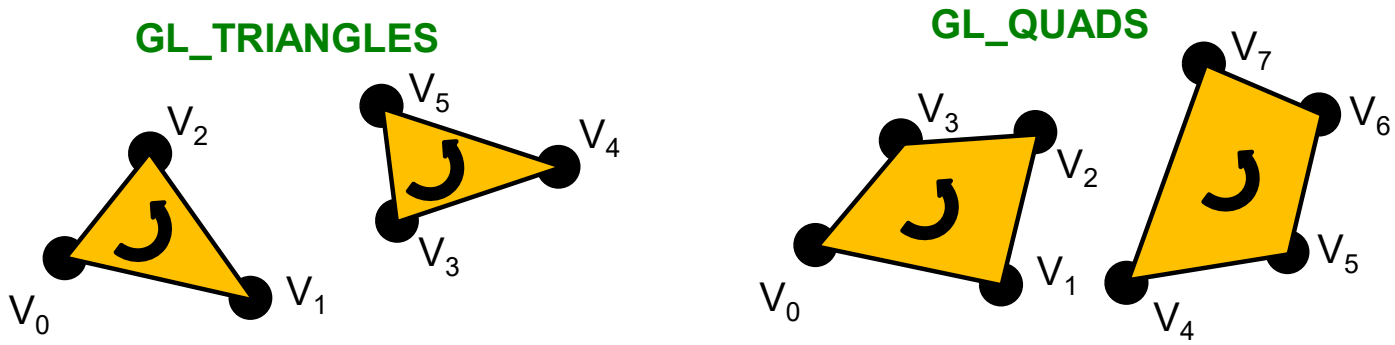
Graphics hardware assumes that a polygon has a definite front and a definite back, and that you can only see one of them at a time



OpenGL Topologies -- Orientation

Polygons are traditionally:

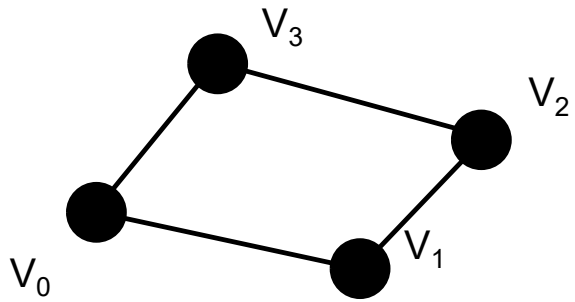
- **CCW when viewed from outside the solid object**



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

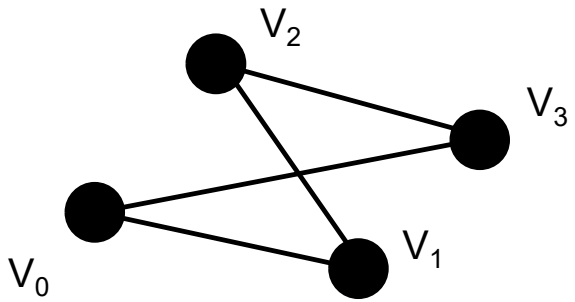
OpenGL Topologies – Vertex Order Matters

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” ☺

OpenGL Drawing Can Be Done *Procedurally*

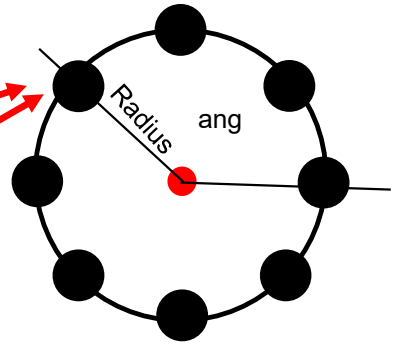
```
glColor3f( r, g, b );
glBegin( GL_LINE_LOOP );
        glVertex3f( x0, y0, 0. );
        glVertex3f( x1, y1, 0. );
        . . .
glEnd( );
```

Listing a lot of vertices explicitly gets old in a hurry

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.

Draw a circle using lines:

```
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( );
```



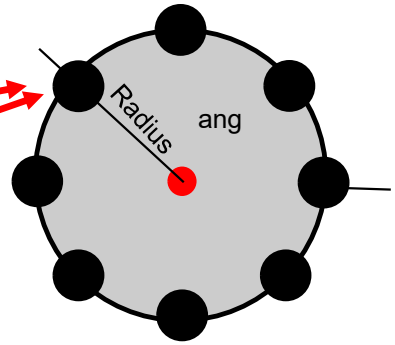
OpenGL Drawing Can Be Done *Procedurally*

The graphics card can't tell how the numbers in the glVertex3f calls were produced.

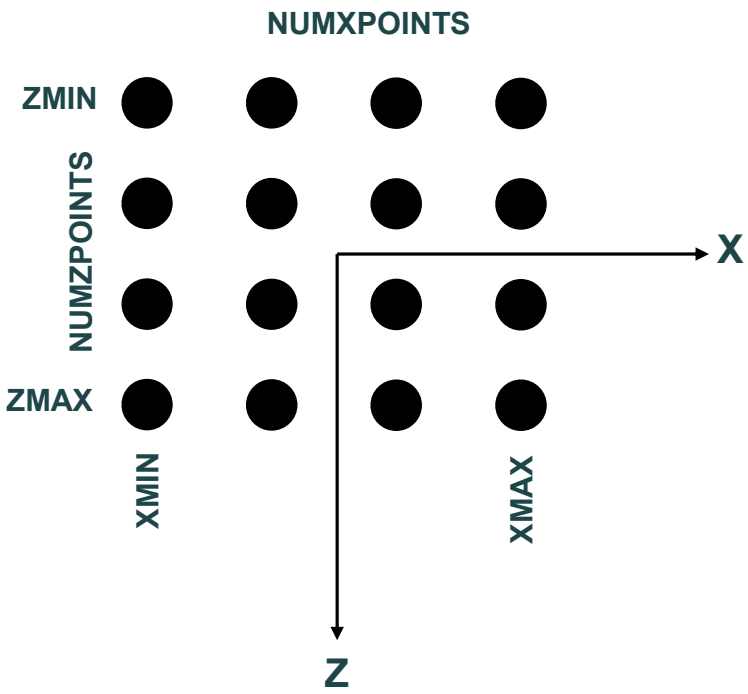
Both explicitly-listed and procedurally-computed look the same to glVertex3f.

Draw a circle using polygons:

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



OpenGL Drawing Can Be Done *Procedurally*



Draw a grid:

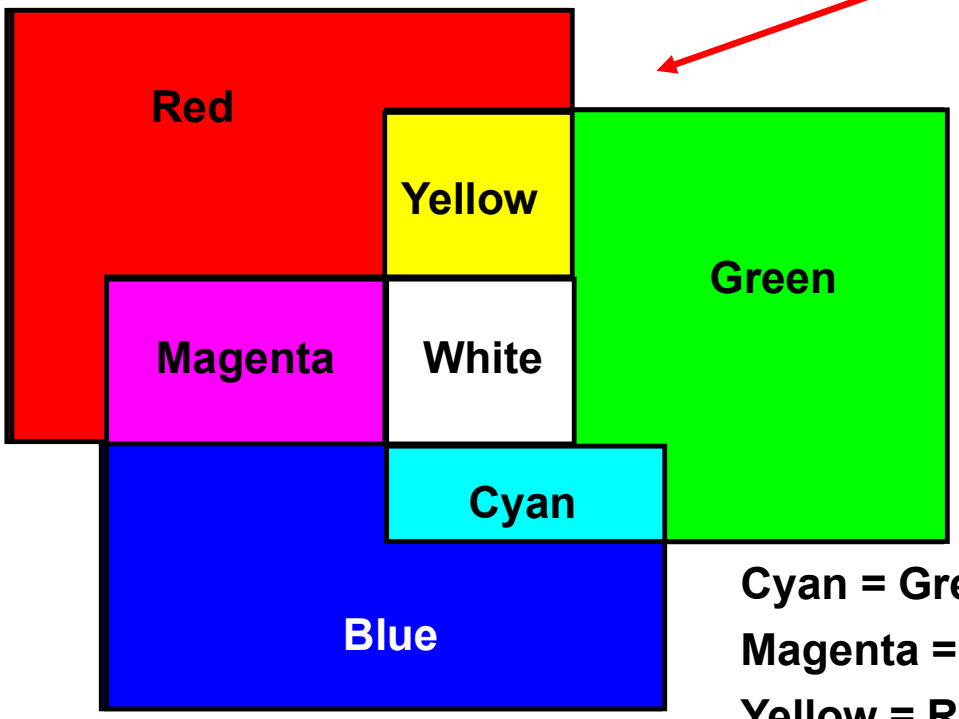
```
glColor3f( r, g, b );
float dx = (XMAX-XMIN) / (float)( NUMXPOINTS – 1 );
float dz = (ZMAX-ZMIN) / (float)( NUMZPOINTS – 1 );
float z = ZMIN;
for( int i = 0; i < NUMZPOINTS; i++ )
{
        float x = XMIN;
        glBegin( GL_LINE_STRIP );
        for( int j = 0; j < NUMXPOINTS; j++ )
        {
                float y = 0;
                // could do something in here to vary y:
                glVertex3f( x, y, z );
                x += dx;
        }
        glEnd( );
        z += dz;
}
```

Color

glColor3f(r, g, b);

0.0 ≤ r, g, b ≤ 1.0

This is referred to as **"Additive Color"**



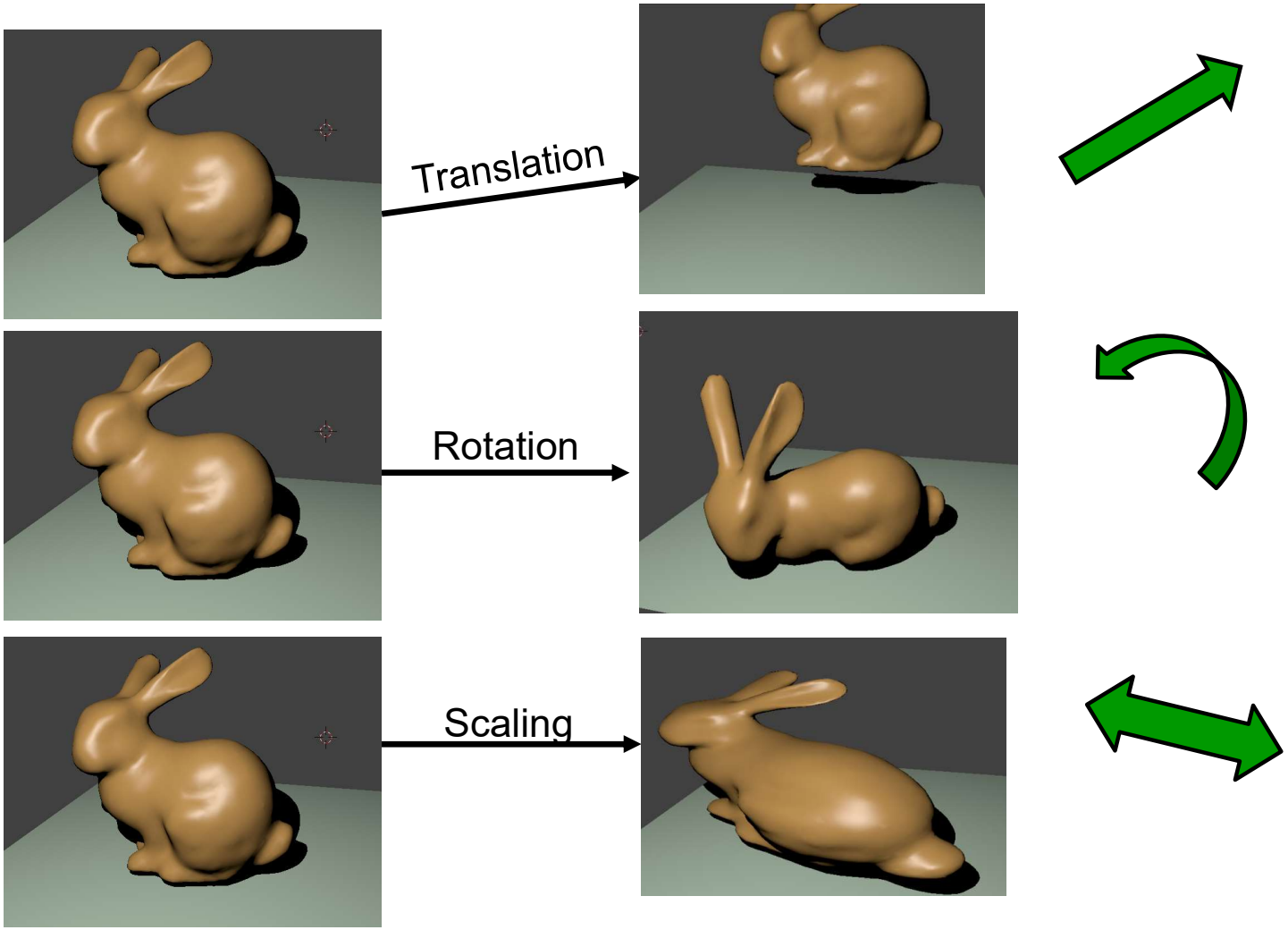
Cyan = Green + Blue

Magenta = Red + Blue

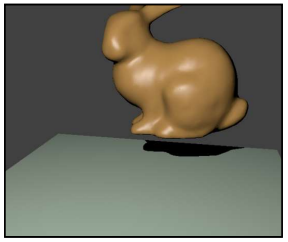
Yellow = Red + Green

White = Red + Green + Blue

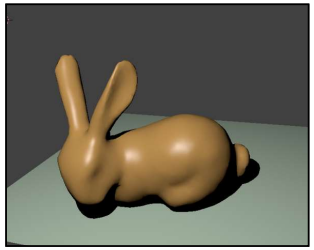
Transformations



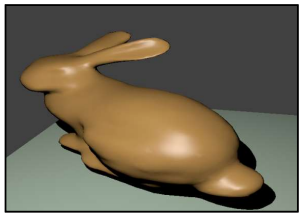
OpenGL Transformations



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( );
```

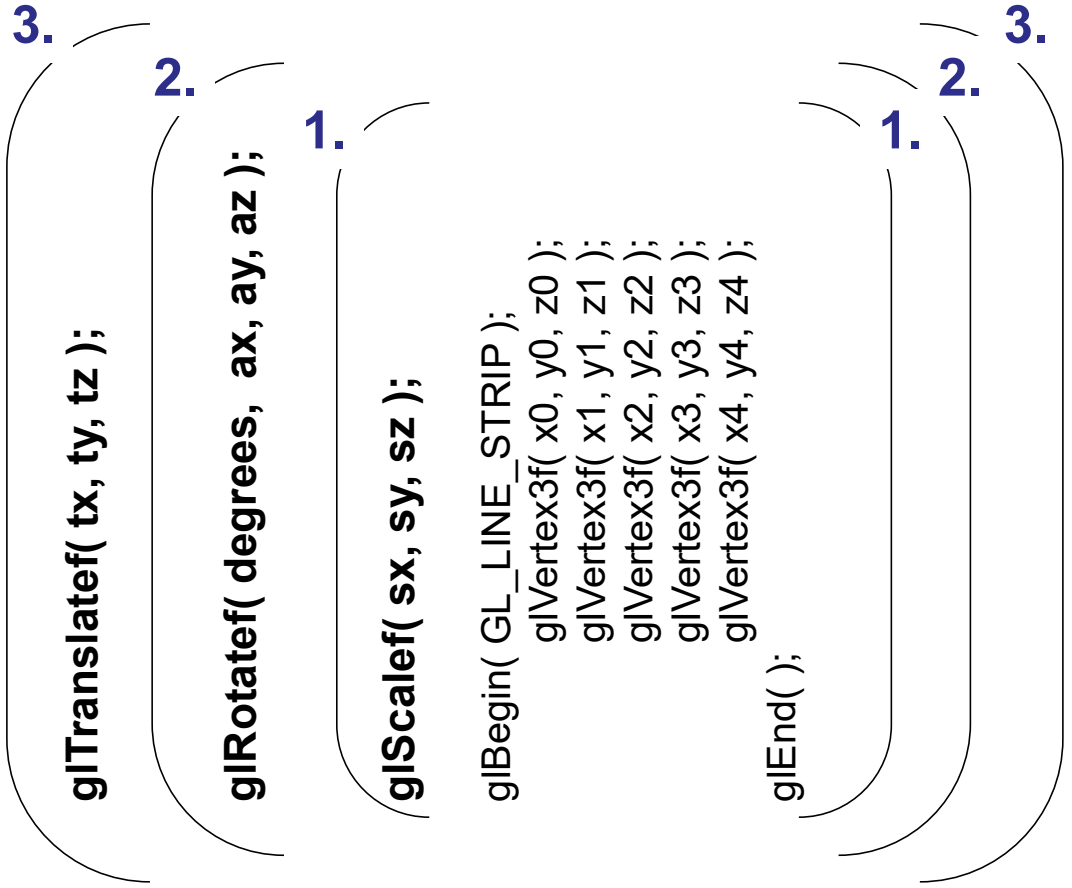
Compound Transformations

```
glMatrixMode( GL_MODELVIEW );
glLoadIdentity( )

glTranslatef( tx, ty, tz );
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( );
```

3.
2.
1.

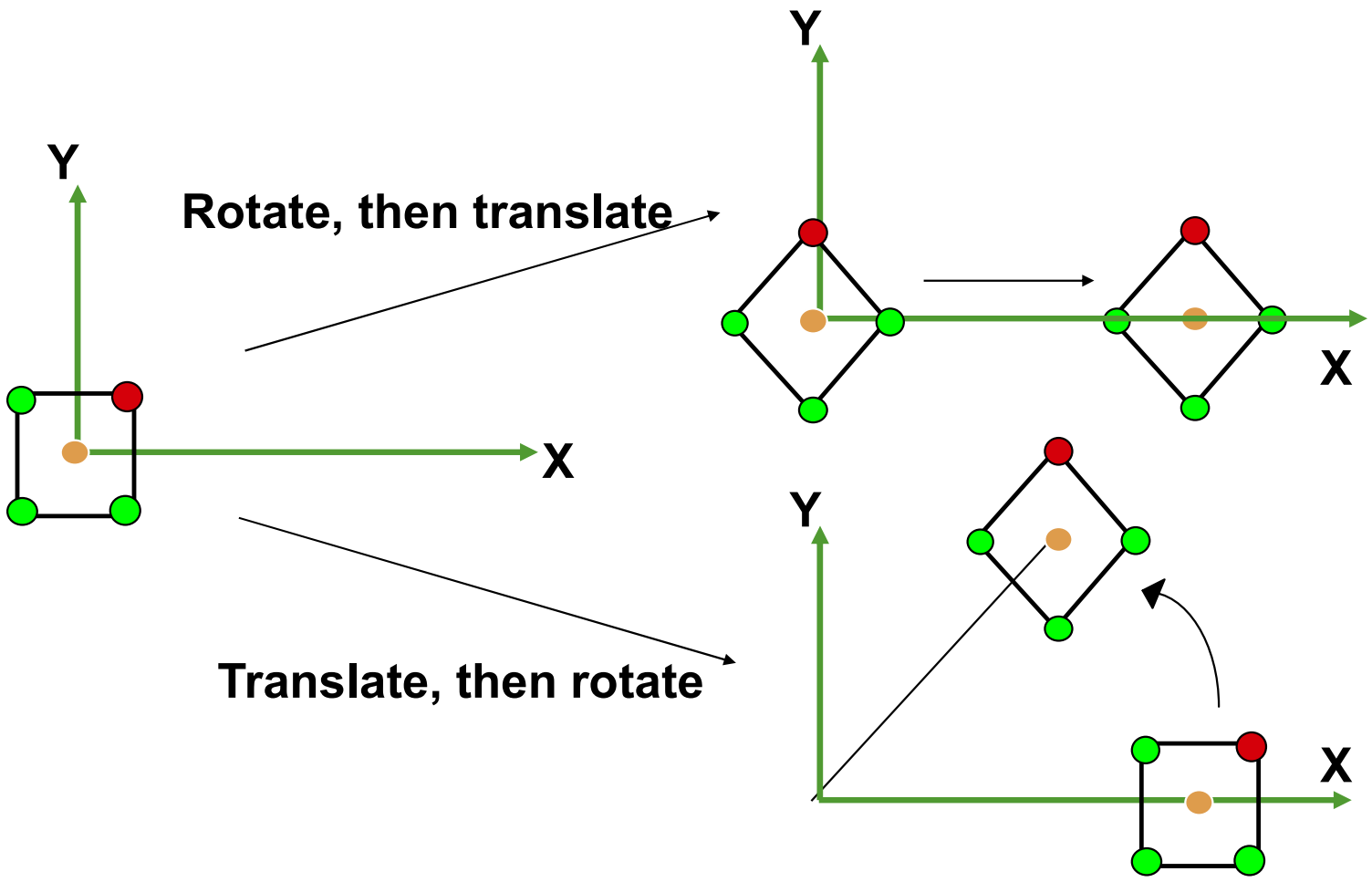
These transformations "add up", and take effect in this order

Why do the Compound Transformations Take Effect in Reverse Order?



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 and colors before you can expect them to take effect!

1. Set the state

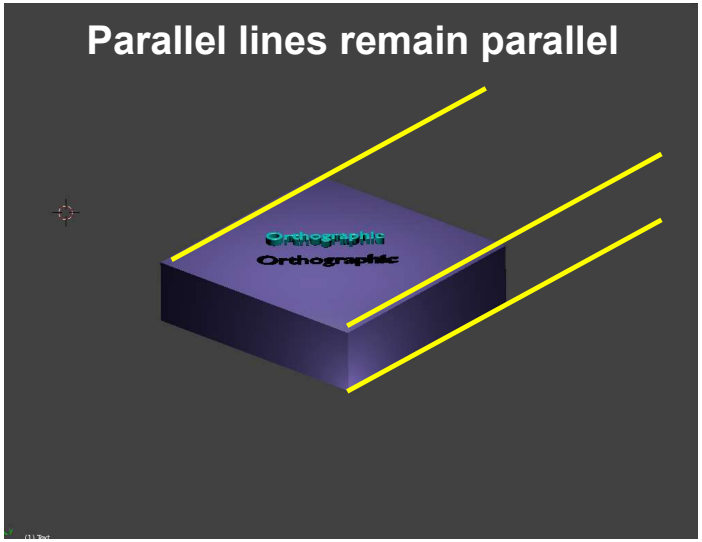
2. Draw with that state

```
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

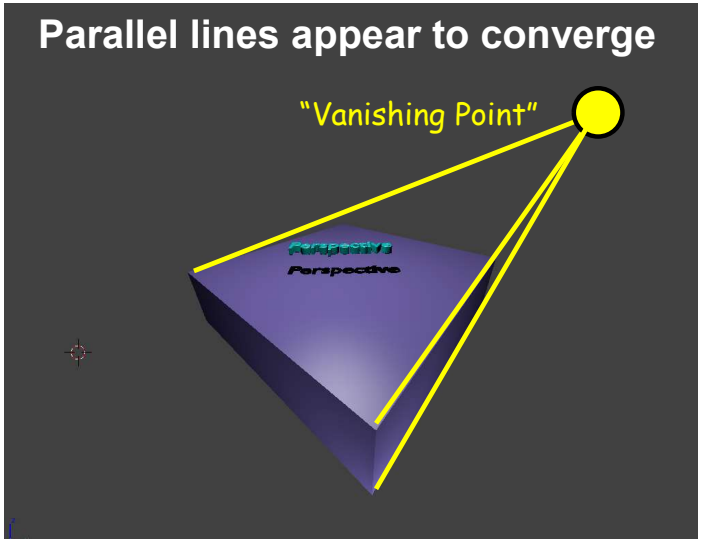
Orthographic (or Parallel) projection

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



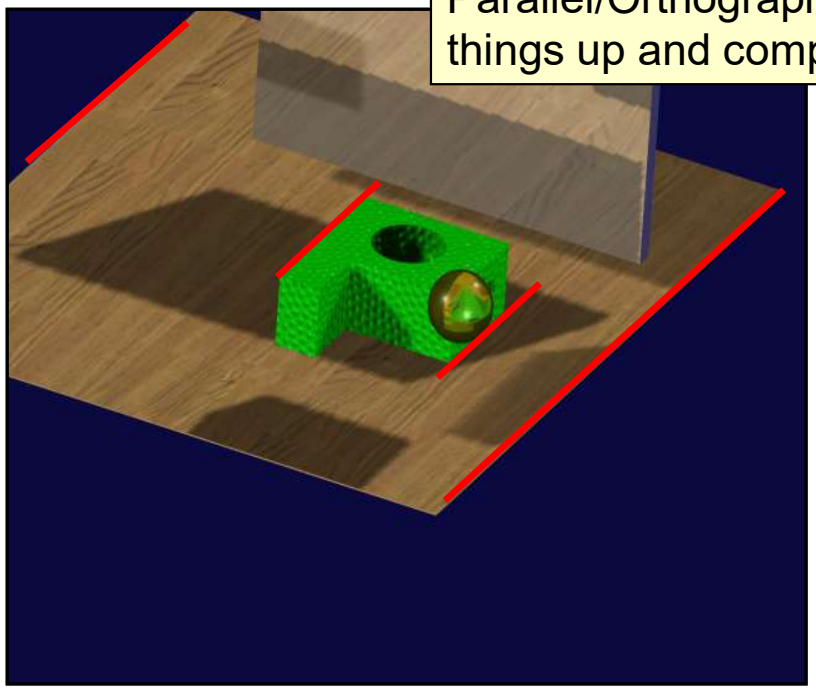
Perspective projection

gluPerspective(fovy, aspect, zn, zf);

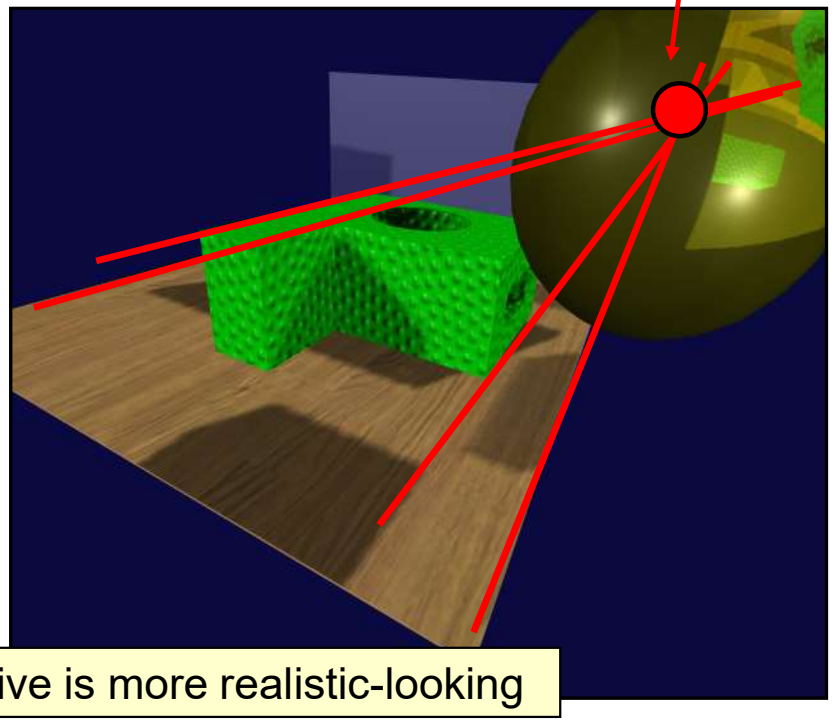


Projecting on Object from 3D to 2D

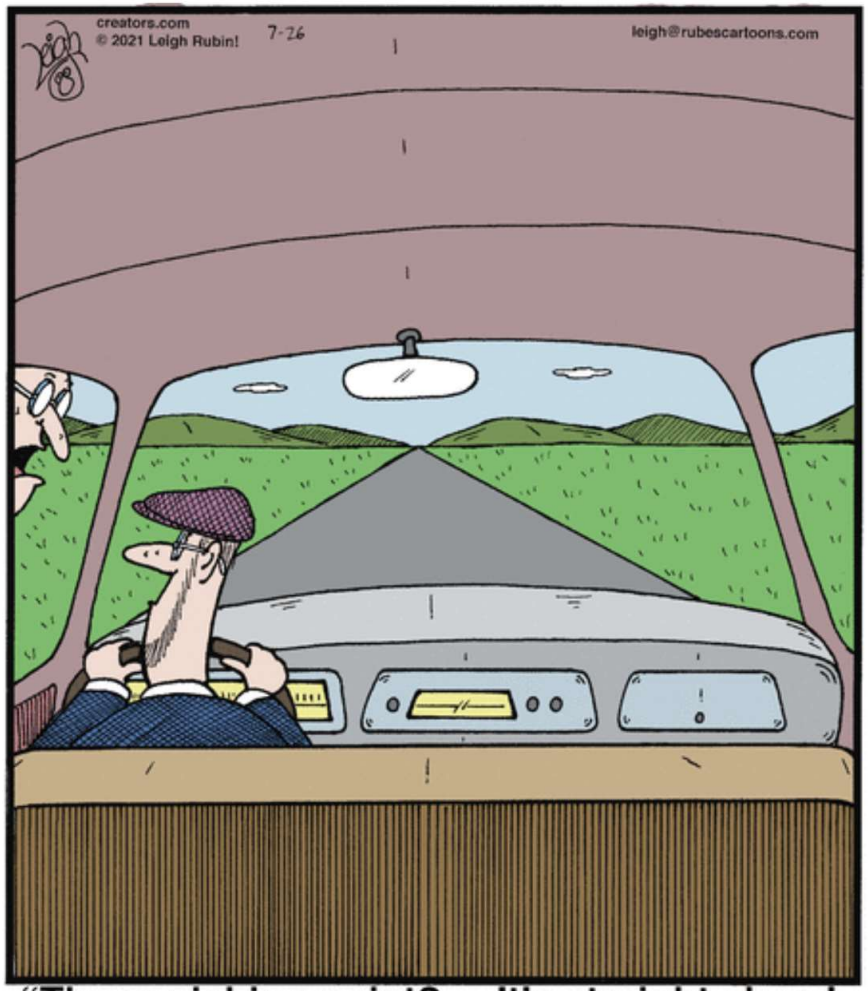
Parallel/Orthographic is good for lining things up and comparing sizes



The ***Vanishing Point***



Perspective is more realistic-looking



"The vanishing point? ... It's straight ahead. You can't miss it."

https://www.gocomics.com/rubes

OpenGL Projection Functions

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

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

Use one of (glOrtho, gluPerspective), but not both!

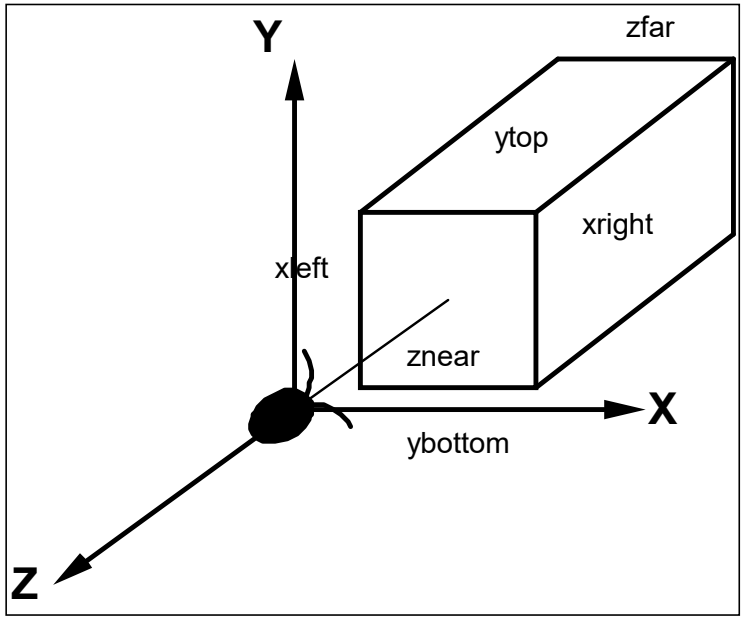
```
glMatrixMode( GL_MODELVIEW );
glLoadIdentity( );

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( );
```

OpenGL Projection Functions

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

if( WhichProjection == ORTHO )
        glOrtho( -2.f, 2.f,     -2.f, 2.f,     0.1f, 1000.f );
 else
        gluPerspective( 70.f, 1.f,      0.1f, 1000.f );

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( );
```

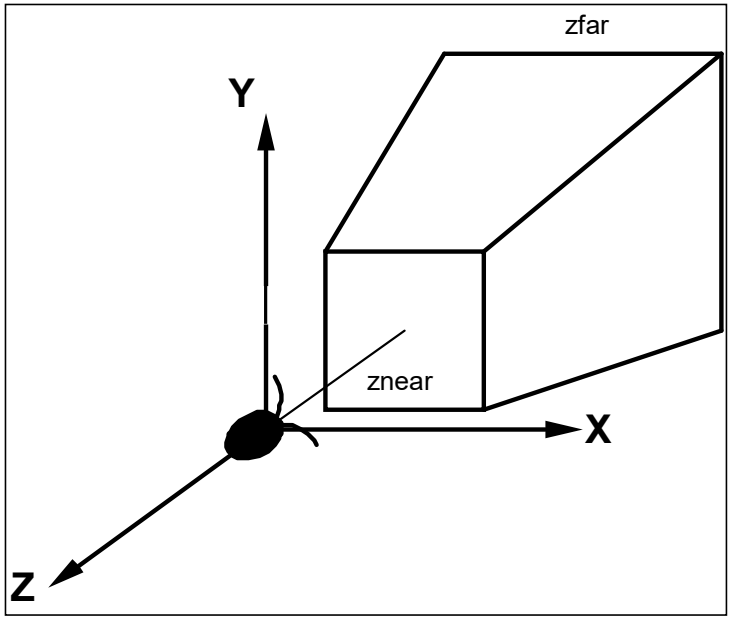
How the Viewing Volumes Look from the Outside

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



Parallel/Orthographic

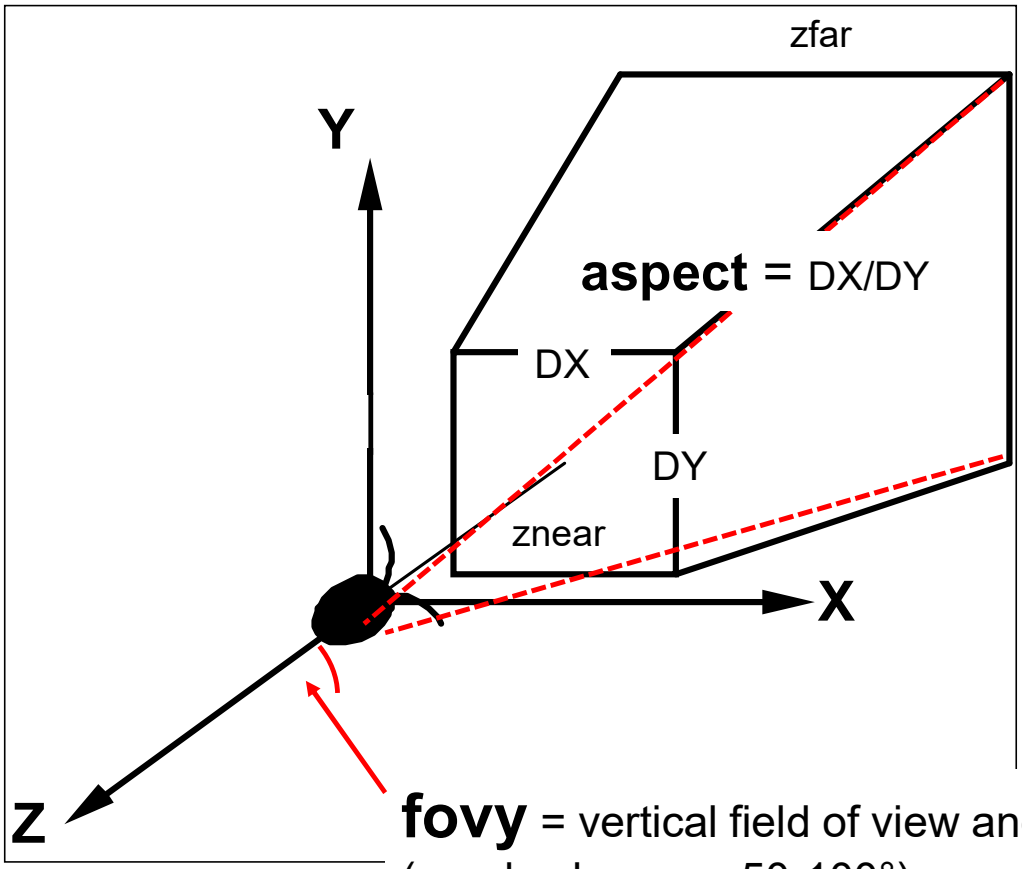
gluPerspective(fovy, aspect, zn, zf);



Perspective

The Perspective Viewing Frustum

gluPerspective(fovy, aspect, zn, zf);



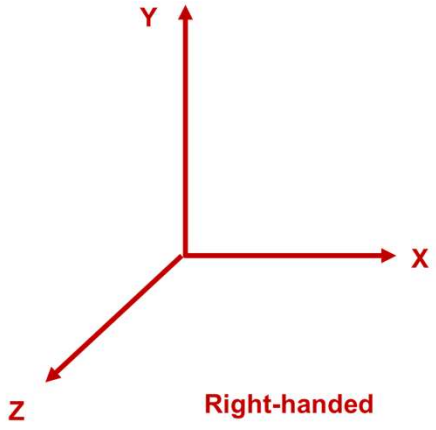
Arbitrary Viewing

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

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

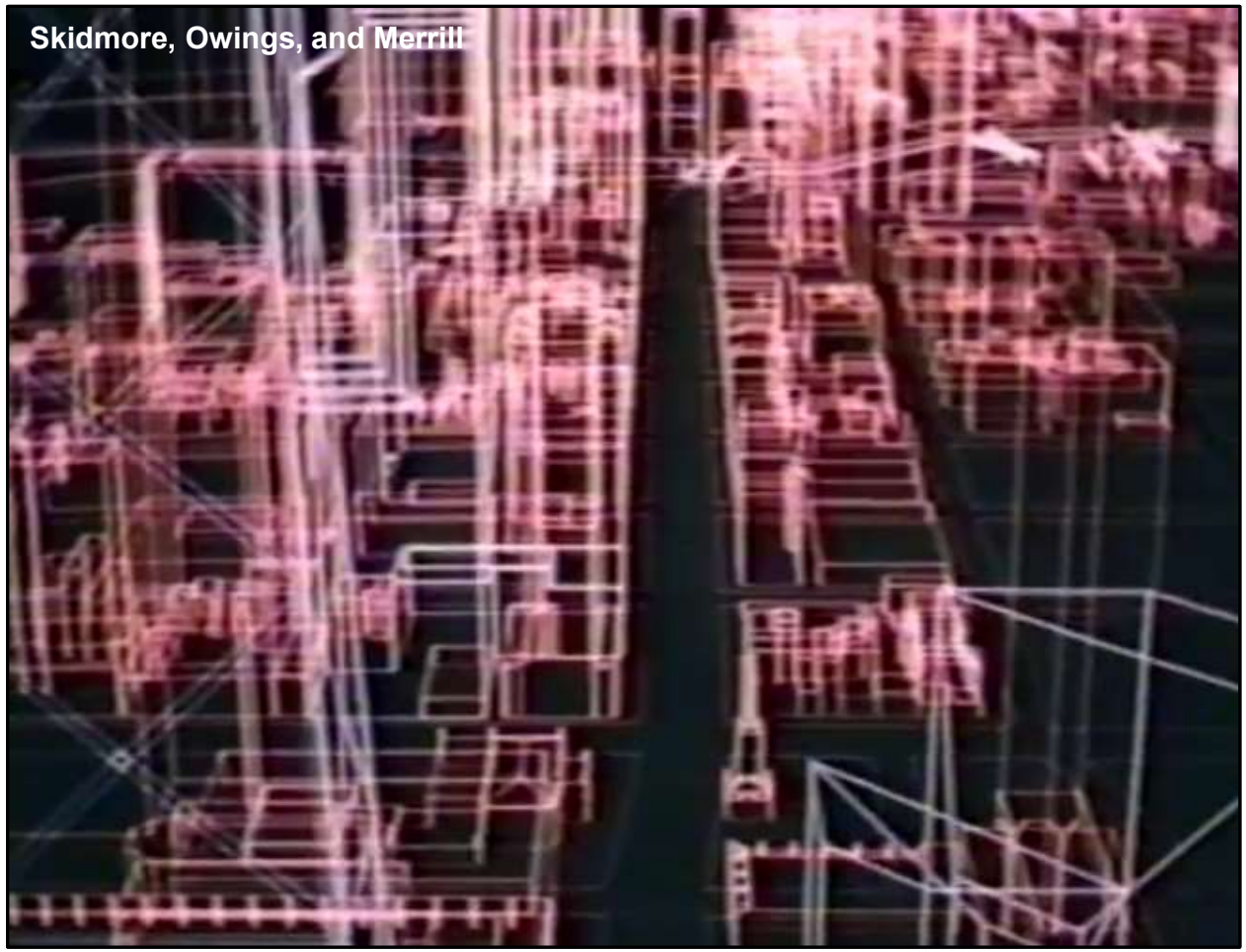
Eye Position *Look-at Position* *Up vector*

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



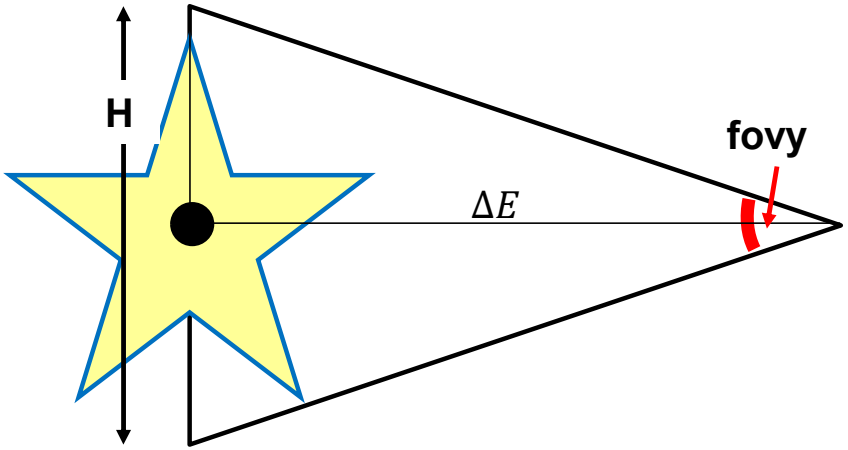
How Can You Be Sure You See Your Scene?

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:

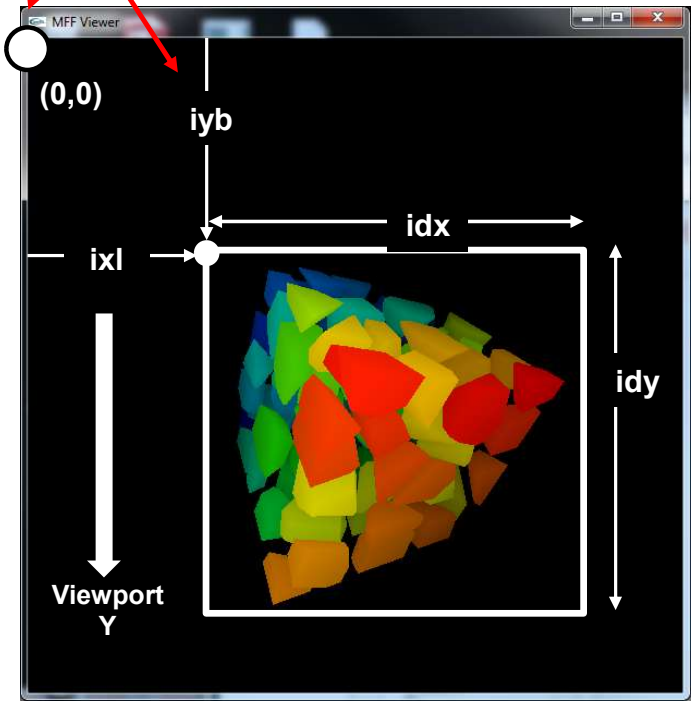
$$\Delta E = \frac{H}{2tan(\frac{fovy}{2})}$$

Specifying a Viewport

Be sure the y:x aspect ratios match!!

```
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( );
```

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



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 };
const int 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
};
```

Change this to be your name!

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

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

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( );
}
```

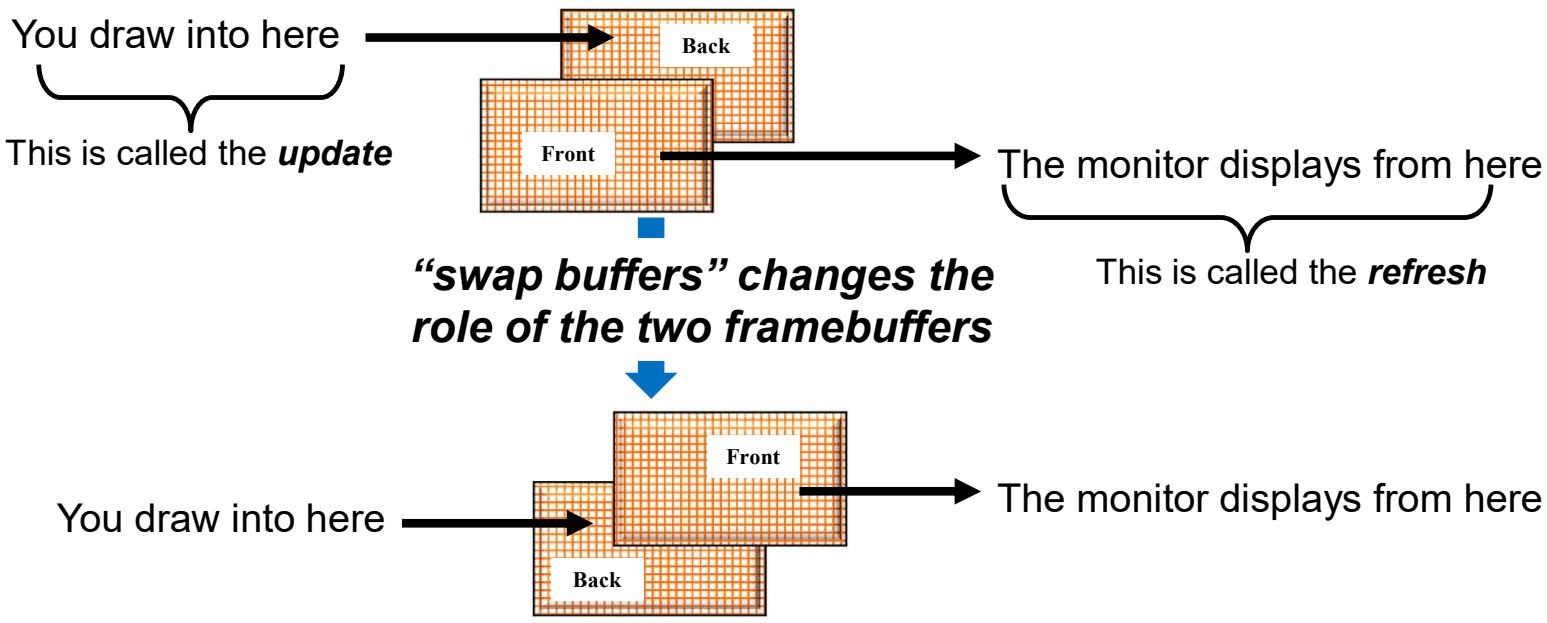
(x,y,z), to be *translated* by the ModelView matrix

glutSwapBuffers( )

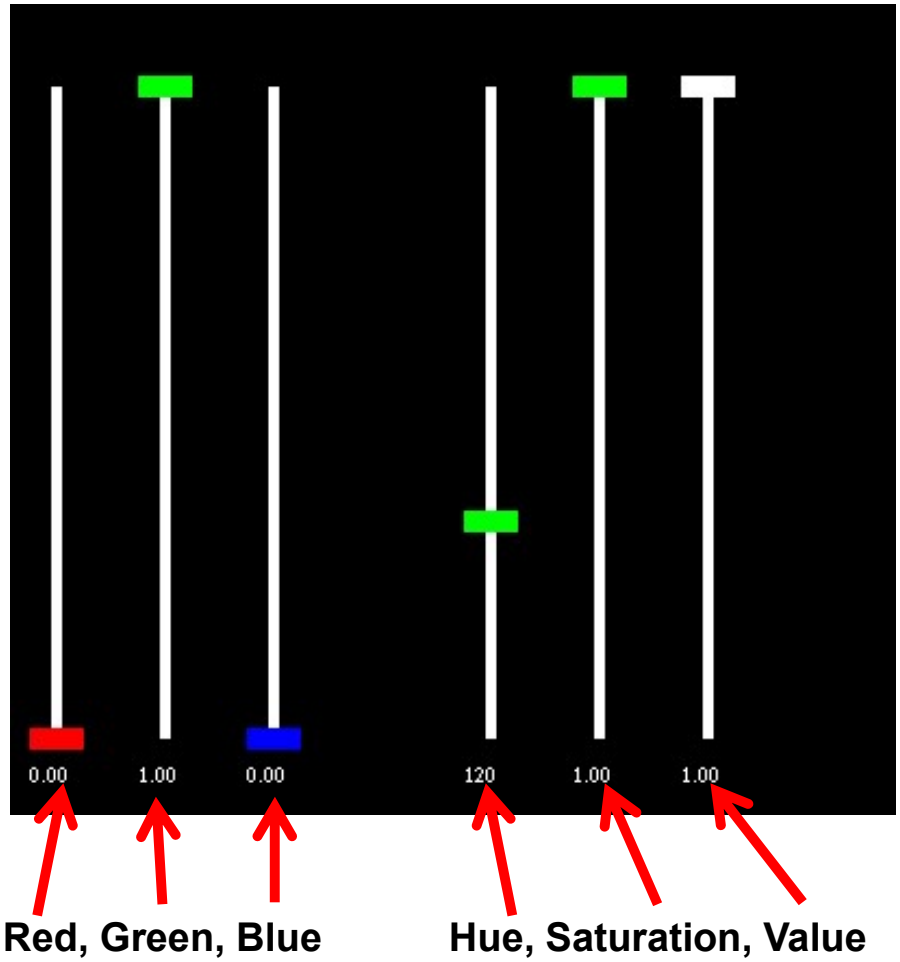
```
// swap the double-buffered framebuffers:

glutSwapBuffers( );
```

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

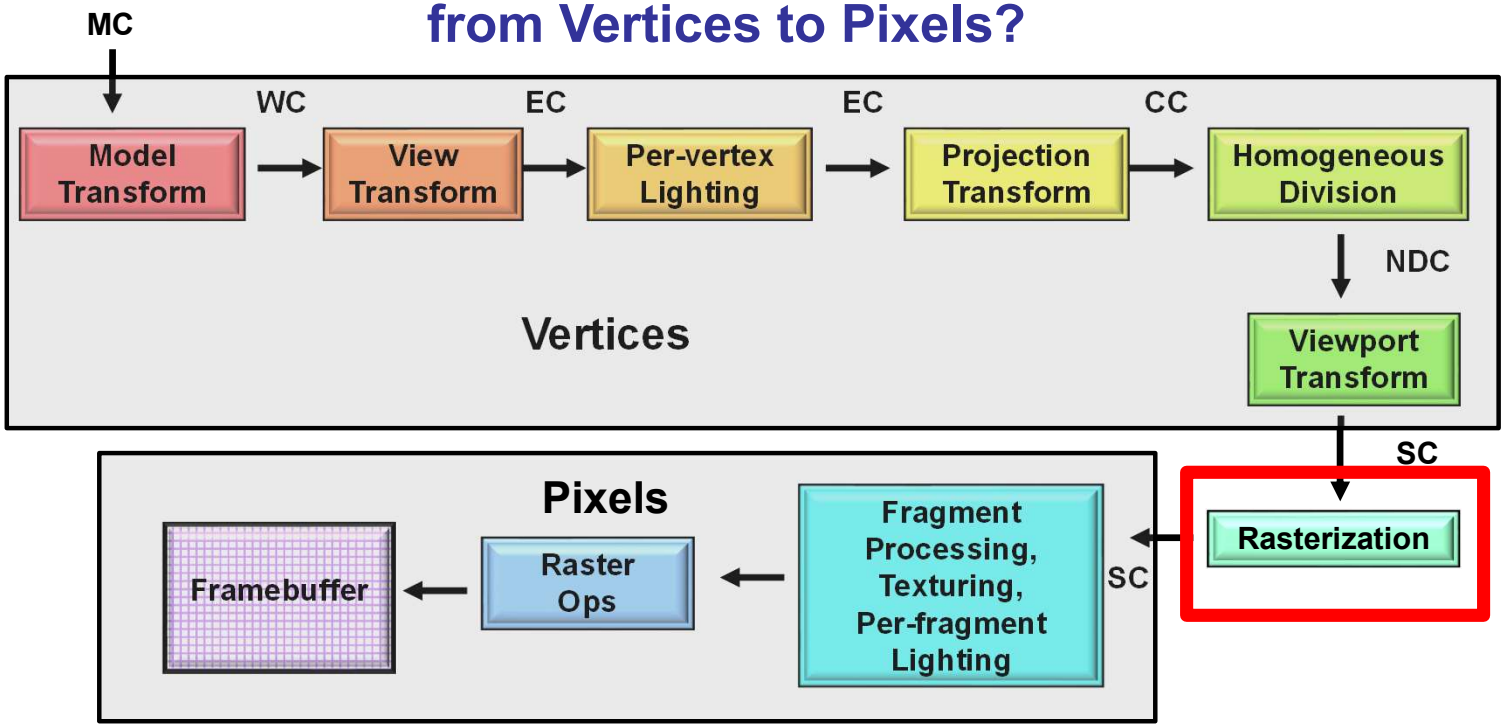
```
glDrawBuffer( GL_BACK );
```



The OSU *ColorPicker* Program



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

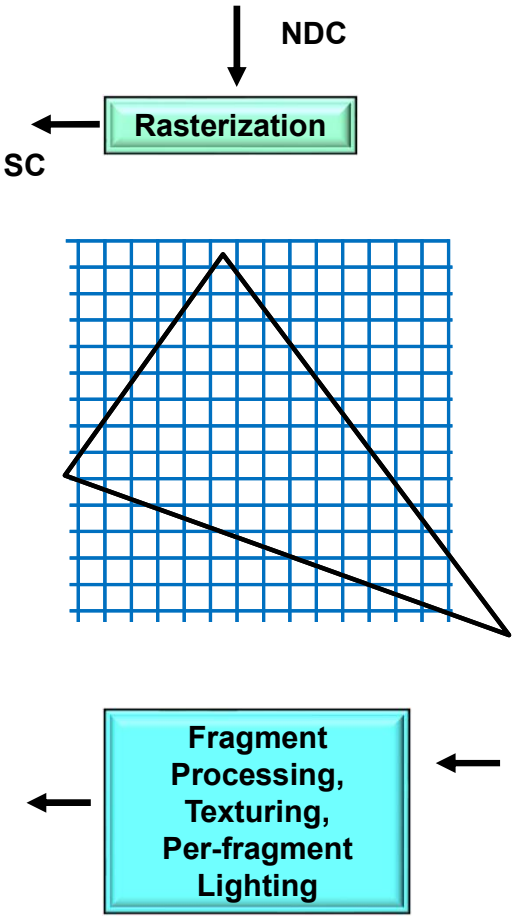
Sidebar: How Did We Make the Transition from Vertices to Pixels?

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, the word "pixel" is defined as having its full RGBA already computed. A fragment does not yet have its final RGBA computed, but all of the information needed to compute the RGBA is available to it.

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!



Display Lists

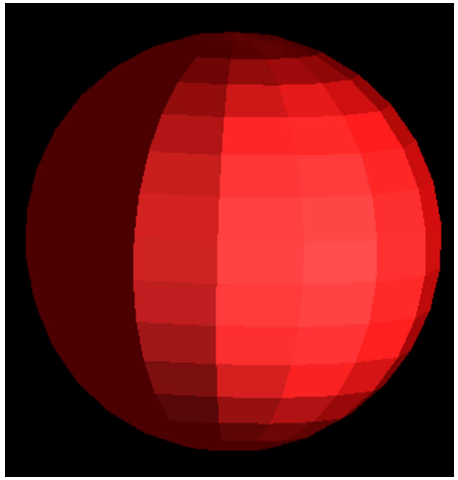
Mike Bailey

mjb@cs.oregonstate.edu

Oregon State
University
Computer Graphics

Drawing a Sphere – Notice a lot of time-consuming Trig Function Calls!

```
void
OsuSphere( float radius, int slices, int stacks )
{
	struct point top, bot;		// top, bottom points
	struct point *p;
	NumLngs = slices;
	NumLats = stacks;

	Pts = new struct point[ NumLngs * NumLats ];
	for( int ilat = 0; ilat < NumLats; ilat++ )
	{
		float lat = -M_PI/2.  +  M_PI * (float)ilat / (float)(NumLats-1);
		float xz = cos( lat );
		float y  = sin( lat );
		for( int ilng = 0; ilng < NumLngs; ilng++ )
		{
			float lng = -M_PI  +  2. * M_PI * (float)ilng / (float)(NumLngs-1);
			float x =  xz * cos( lng );
			float z = -xz * sin( lng );
			p = PtsPointer( ilat, ilng );
			p->x  = radius * x;
			p->y  = radius * y;
			p->z  = radius * z;
			p->nx = x;
			p->ny = y;
			p->nz = z;
			p->s = ( lng + M_PI    ) / ( 2.*M_PI );
			p->t = ( lat + M_PI/2. ) / M_PI;
		}
	}
```



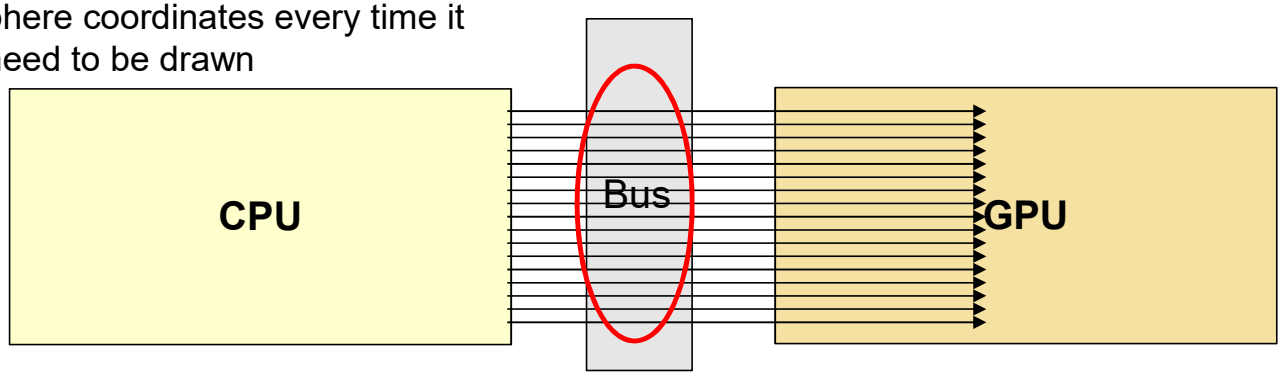
Even worse, the trig calls are inside single or double-nested for-loops!

```
top.x  =  0.;        top.y  = radius;      top.z  = 0.;
top.nx = 0.;         top.ny = 1.;          top.nz = 0.;
top.s  = 0.;         top.t  = 1.;
bot.x  =  0.;        bot.y  = -radius;     bot.z = 0.;
bot.nx = 0.;         bot.ny = -1.;         bot.nz = 0.;
bot.s  = 0.;         bot.t  =  0.;

glBegin( GL_QUADS );
for( int ilng = 0; ilng < NumLngs-1; ilng++ )
{
        p = PtsPointer( NumLats-1, ilng );
        DrawPoint( p );
        p = PtsPointer( NumLats-2, ilng );
        DrawPoint( p );
        p = PtsPointer( NumLats-2, ilng+1 );
        DrawPoint( p );
        p = PtsPointer( NumLats-1, ilng+1 );
        DrawPoint( p );
}
glEnd( );

glBegin( GL_QUADS );
for( int ilng = 0; ilng < NumLngs-1; ilng++ )
{
        p = PtsPointer( 0, ilng );
        DrawPoint( p );
        p = PtsPointer( 0, ilng+1 );
        DrawPoint( p );
        p = PtsPointer( 1, ilng+1 );
        DrawPoint( p );
        p = PtsPointer( 1, ilng );
        DrawPoint( p );
}
glEnd( );
```

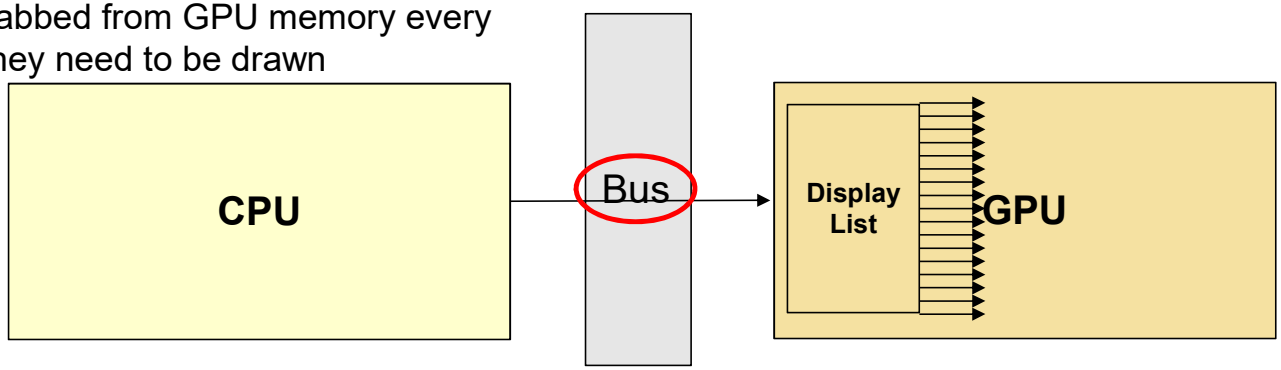
```
glBegin( GL_QUADS );
for( int ilat = 2; ilat < NumLats-1; ilat++ )
{
	for( int ilng = 0; ilng < NumLngs-1; ilng++ )
	{
		p = PtsPointer( ilat-1, ilng );
		DrawPoint( p );
		p = PtsPointer( ilat-1, ilng+1 );
		DrawPoint( p );
		p = PtsPointer( ilat, ilng+1 );
		DrawPoint( p );
		p = PtsPointer( ilat, ilng );
		DrawPoint( p );
	}
}
glEnd( );
```

You don't want to execute all that code every time you want to redraw the scene, so draw it once, store the numbers in GPU memory, and call them back up later

Without a Display List:
The CPU re-computes and transmits the sphere coordinates every time it they need to be drawn



With a Display List:
The CPU computes and transmits the sphere coordinates *once* and then they are grabbed from GPU memory every time they need to be drawn



You don't want to execute all that code every time you want to redraw the scene, so draw it once, store the numbers in GPU memory, and call them back up later

The solution is to incur the sphere-creation overhead *once*, and whenever the sphere needs to be re-drawn, just draw the saved numbers, not the equations. This is a **Display List**.

1. How many unique, unused, consecutive DL identifiers to give back to you

2 . The ID of the first DL in the unique, unused list

*Creating the Display List in **InitLists( ):***

```
// a global GLuint variable:
SphereList = glGenLists( 1 );
glNewList( SphereList, GL_COMPILE );

OsuSphere( 5., 30, 30 );

glEndList( );
```

3. Open up a display list in (GPU) memory

4. The coordinates, etc. end up in memory instead of being sent to the display

5. All done with storing the numbers in the DL

*Calling up the Display List in **Display( ):***

```
glCallList( SphereList );
```

6. Pull all the coordinates, etc. from memory, just as if the code to generate them had been executed here

Oregon State University
Computer Graphics

A Common Display List Misconception

Let's say that we are creating a rectangle in a Display List, like this:

```
float  L, W;          // length and width global variables
int    RectList;      // rectangle display list
```

```
L = 10.;  W = 5.;
glNewList( RectList, GL_COMPILE );
        << draw a rectangle using L and W >>
glEndList( );
```

Then, when we go to use the DL, we do this:

```
L = 4.;    W = 2.;
glCallList( RectList );
```

What size rectangle will it draw? 10x5? 4x2?

It will draw a 10x5 rectangle. Display Lists bake in the ***numbers***. They retain no knowledge of what ***variables*** were used to create those numbers!

The GL Utility Toolkit (GLUT)

Mike Bailey

mjb@cs.oregonstate.edu

Oregon State
University
Computer Graphics

What is GLUT?

The **GL Utility Toolkit** (GLUT) serves two major purposes:

1. It interfaces with your operating system and window system
2. It provides various application utilities, such as drawing 3D shapes for you

You can find GLUT (actually freeGLUT) at:

http://freeglut.sourceforge.net/

You don't actually have to go out here. We will give you some libraries that are ready-to-use.

Using GLUT to Setup the Window

All the GLUT_XXX constants are #defined in **glut.h**

GLUT_RGBA	I want to display colors
GLUT_DOUBLE	I want to do double-buffering
GLUT_DEPTH	I want to use a depth-buffer while rendering

```
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 );
```

Constants not beginning with **GL_** or **GLUT_** are user-defined

Using GLUT to Specify Event-driven Callback Functions

```
glutSetWindow( MainWindow );
glutDisplayFunc( Display );
glutReshapeFunc( Resize );
glutKeyboardFunc( Keyboard );
glutMouseFunc( MouseButton );
glutMotionFunc( MouseMotion );

glutPassiveMotionFunc( NULL );
glutVisibilityFunc( Visibility );
glutEntryFunc( NULL );
glutSpecialFunc( NULL );
glutSpaceballMotionFunc( NULL );
glutSpaceballRotateFunc( NULL );
glutSpaceballButtonFunc( NULL );
glutButtonBoxFunc( NULL );
glutDialsFunc( NULL );
glutTabletMotionFunc( NULL );
glutTabletButtonFunc( NULL );
glutMenuStateFunc( NULL );
glutTimerFunc( -1, NULL, 0 );
glutIdleFunc( Animate );
```

For example, the **Keyboard( )** function gets called whenever a keyboard key is hit

A NULL callback function means that this event will be ignored

The *Keyboard* Callback Function

```
void
Keyboard( unsigned char c, int x, int y )
{
	if( DebugOn != 0 )
		fprintf( stderr, "Keyboard: '%c' (0x%0x)\n", c, c );

	switch( c )
	{
		case 'o':  case 'O':
			NowProjection = ORTHO;
			break;

		case 'p':  case 'P':
			NowProjection = PERSP;
			break;

		case 'q':  case 'Q':
		case ESCAPE:
			DoMainMenu( QUIT );	// will not ever return
			break;			// keep the compiler happy

		default:
			fprintf( stderr, "Don't know what to do with keyboard hit: '%c' (0x%0x)\n", c, c );
	}

	// force a call to Display( ):

	glutSetWindow( MainWindow );
	glutPostRedisplay( );
}
```

Where the mouse was when the key was hit

The key that was hit

Assign new display parameter values depending on what key was hit

Good programming practice

glutPostRedisplay( ) forces your Display( ) function to be called to redraw the scene with the new display parameter values

The *MouseButton* Callback Function

```
void
MouseButton( int button, int state, int x, int y )
{
	int b = 0;			// LEFT, MIDDLE, or RIGHT
	if( DebugOn != 0 )
		fprintf( stderr, "MouseButton: %d, %d, %d, %d\n", button, state, x, y );

	// get the proper button bit mask:
	switch( button )
	{
		case GLUT_LEFT_BUTTON:
			b = LEFT;		break;

		case GLUT_MIDDLE_BUTTON:
			b = MIDDLE;		break;

		case GLUT_RIGHT_BUTTON:
			b = RIGHT;		break;

		default:
			b = 0;
			fprintf( stderr, "Unknown mouse button: %d\n", button );
	}

	// button down sets the bit, up clears the bit:
	if( state == GLUT_DOWN )
	{
		Xmouse = x;
		Ymouse = y;
		ActiveButton |= b;		// set the proper bit
	}
	else
	{
		ActiveButton &= ~b;		// clear the proper bit
	}
}
```

Where the mouse was when the button was hit

GLUT_DOWN or GLUT_UP

Which button was hit

Oregon State University
Computer Graphics

The *MouseMotion* Callback Function

```
void
MouseMotion( int x, int y )
{
	if( DebugOn != 0 )
		fprintf( stderr, "MouseMotion: %d, %d\n", x, y );

	int dx = x - Xmouse;		// change in mouse coords
	int dy = y - Ymouse;		// change in mouse coords

	if( ( ActiveButton & LEFT ) != 0 )
	{
		Xrot += ( ANGFACT*dy );
		Yrot += ( ANGFACT*dx );
	}

	if( ( ActiveButton & MIDDLE ) != 0 )
	{
		Scale += SCLFACT * (float) ( dx - dy );

		// keep object from turning inside-out or disappearing:
		if( Scale < MINSCALE )
			Scale = MINSCALE;
	}

	Xmouse = x;			// new current position
	Ymouse = y;

	glutSetWindow( MainWindow );
	glutPostRedisplay( );
}
```

Where the mouse moved to

If the mouse moved with the left button down, do a rotate

If the mouse moved with the middle button down, do a scale

glutPostRedisplay( ) forces your Display( ) function to be called to redraw the scene with the new display parameter values

Oregon State University
Computer Graphics

The *Animate* Idle Callback Function

The Idle Function gets called when the GLUT event handler has nothing else to do

```
glutSetWindow( MainWindow );

glutIdleFunc( Animate );
```

Setting it up in InitGraphics( )

We'll talk about this later. This is a good way to control your animations!

```
void
Animate( )
{
	// put animation stuff in here -- change some global variables
	// for Display( ) to find:

	int ms = glutGet( GLUT_ELAPSED_TIME );		// milliseconds
	ms  %=  MS_PER_CYCLE;
	Time = (float)ms  /  (float)MS_PER_CYCLE;	// [ 0., 1. )

	// force GLUT to do a call to Display( ) next time it is convenient:

	glutSetWindow( MainWindow );
	glutPostRedisplay( );
}
```

glutPostRedisplay( ) forces your Display( ) function to be called to redraw the scene with the new display parameter values

Pop-up Menus are Straightforward to Create with GLUT

```
void
InitMenus( )
{
	glutSetWindow( MainWindow );

	int numColors = sizeof( Colors ) / ( 3*sizeof(int) );
	int colormenu = glutCreateMenu( DoColorMenu );
	for( int i = 0; i < numColors; i++ )
	{
		glutAddMenuEntry( ColorNames[i], i );
	}

	int axesmenu = glutCreateMenu( DoAxesMenu );
	glutAddMenuEntry( "Off", 0 );
	glutAddMenuEntry( "On",  1 );

	int depthcuemenu = glutCreateMenu( DoDepthMenu );
	glutAddMenuEntry( "Off",  0 );
	glutAddMenuEntry( "On",  1 );

	int debugmenu = glutCreateMenu( DoDebugMenu );
	glutAddMenuEntry( "Off",  0 );
	glutAddMenuEntry( "On",  1 );

	int projmenu = glutCreateMenu( DoProjectMenu );
	glutAddMenuEntry( "Orthographic",  ORTHO );
	glutAddMenuEntry( "Perspective",  PERSP );

	int mainmenu = glutCreateMenu( DoMainMenu );
	glutAddSubMenu(   "Axes",  axesmenu);
	glutAddSubMenu(   "Colors",  colormenu);
	glutAddSubMenu(   "Depth Cue",  depthcuemenu);
	glutAddSubMenu(   "Projection",  projmenu );
	glutAddMenuEntry( "Reset",  RESET );
	glutAddSubMenu(   "Debug",  debugmenu);
	glutAddMenuEntry( "Quit",  QUIT );

	// attach the pop-up menu to the right mouse button:
	glutAttachMenu( GLUT_RIGHT_BUTTON );
}
```

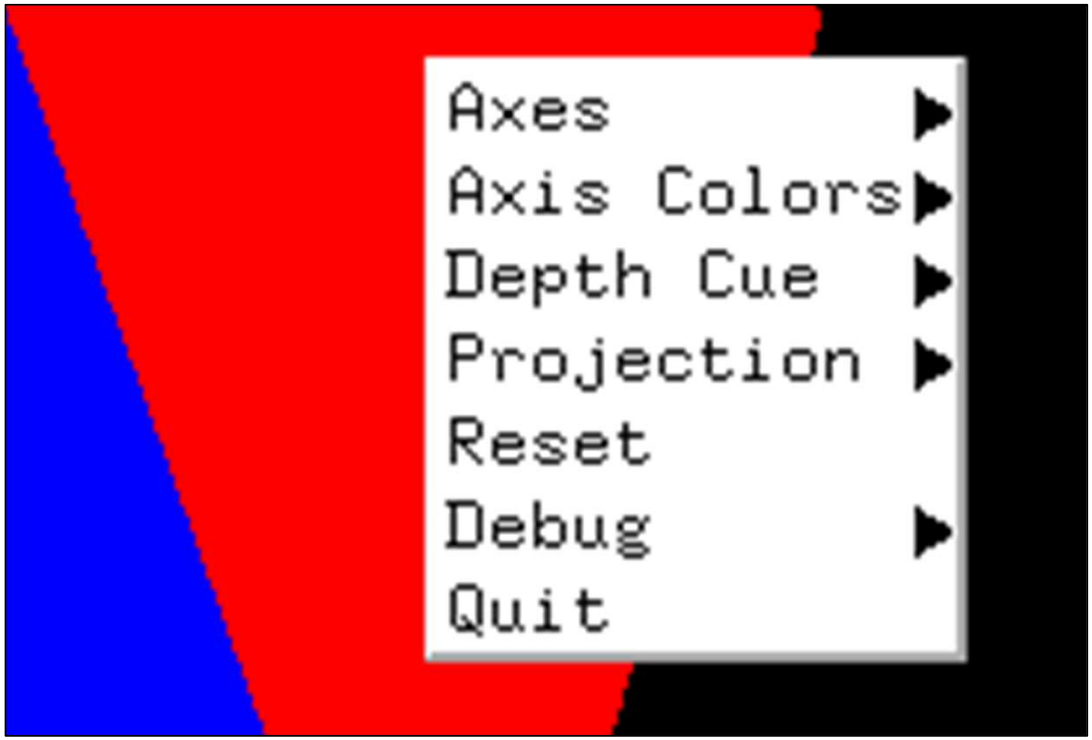
This is the color menu's callback function. When the user selects from this pop-up menu, its callback function gets executed. Its argument is the integer ID of the menu item that was selected. You specify that integer ID in **glutAddMenuEntry( ).**

This is how you create hierarchical sub-menus

Finally, tell GLUT which mouse button activates the entire menu hierarchy

Oregon State University
Computer Graphics

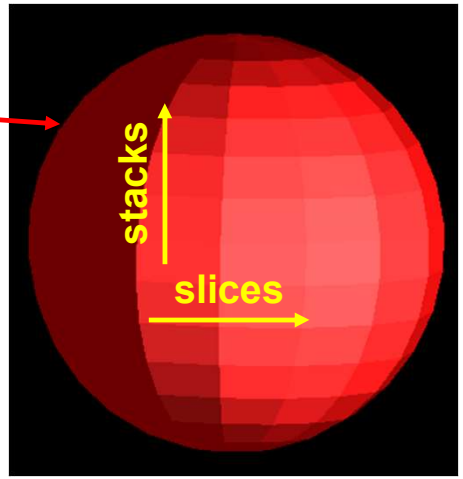
When You Hit the Right Mouse Button, This Menu Pops Up

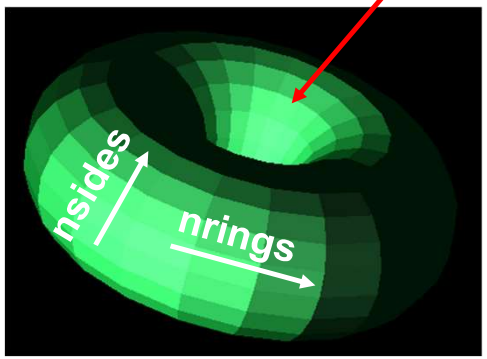


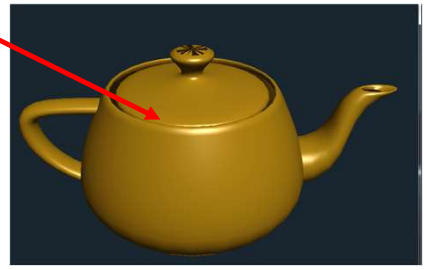
Feel free to add as many of your own items into this menu as you want!

The GLUT 3D Objects

- glutSolidSphere(radius, slices, stacks);
- glutWireSphere(radius, slices, stacks) ;
- glutSolidCube(size);
- glutWireCube(size);
- glutSolidCone(base, height, slices, stacks);
- glutWireCone (base, height, slices, stacks);
- glutSolidTorus(innerRadius, outerRadius, nsides, nrings);
- glutWireTorus(innerRadius, outerRadius, nsides, nrings);
- glutSolidDodecahedron( );
- glutWireDodecahedron( );
- glutSolidOctahedron( );
- glutWireOctahedron( );
- glutSolidTetrahedron( );
- glutWireTetrahedron( );
- glutSolidIcosahedron( );
- glutWireIcosahedron( );
- glutSolidTeapot(size);
- glutWireTeapot(size);



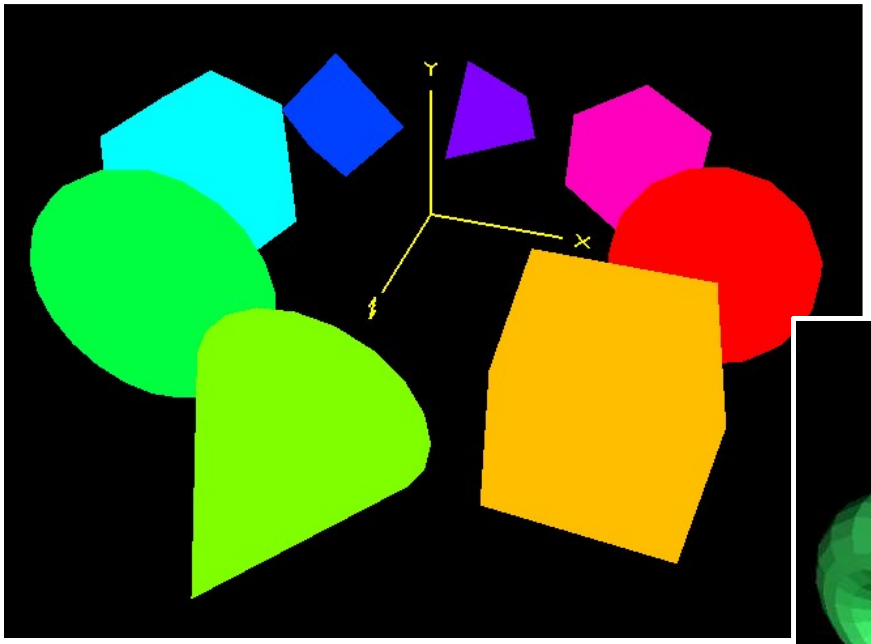




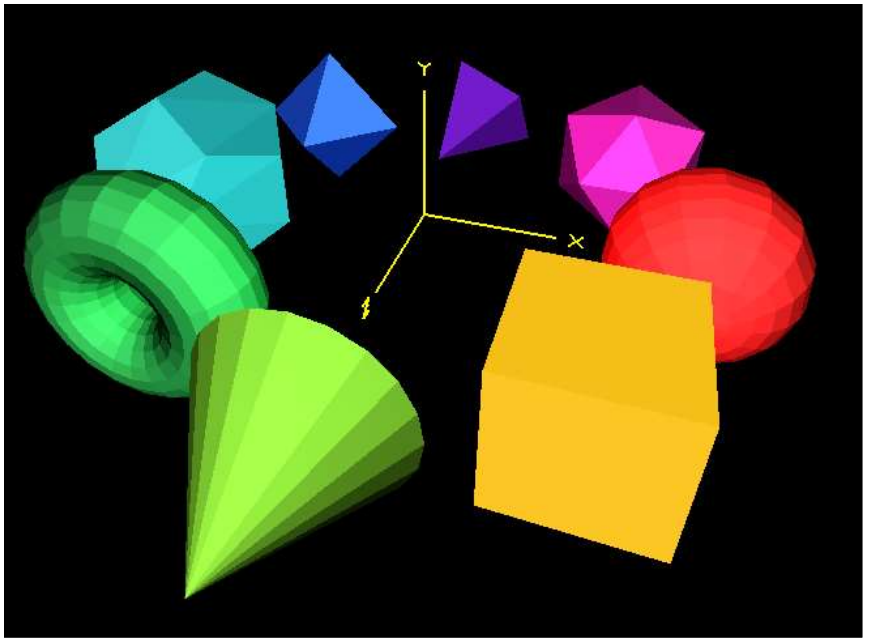
In case you have a hard time remembering which direction "slices" are, think of this:

The GLUT 3D Objects

Without *lighting*, the GLUT solids don't look very cool. I'd recommend you stick with the wireframe versions of the GLUT 3D objects for now! We will get to lighting soon.



Without lighting



With lighting

The OSU 3D Objects

Warning! I recommend that you do not use the following GLUT functions:

- glutSolidSphere(radius, slices, stacks);
- glutSolidCone(base, height, slices, stacks);
- glutSolidTorus(innerRadius, outerRadius, nsides, nrings);

Use our own OSU versions of these instead:

- **OsuSphere(radius, slices, stacks);**
- **OsuCone(radBot, radTop, height, slices, stacks);**
- **OsuTorus(innerRadius, outerRadius, nsides, nrings);**

Our versions are better and more complete. Plus, you have the source code in case you want to make custom modifications.

Using the OSU 3D Objects

Global Variables:

```
int  SphereDL;
int  ConeDL;
int  TorusDL;
```

In InitLists( ):

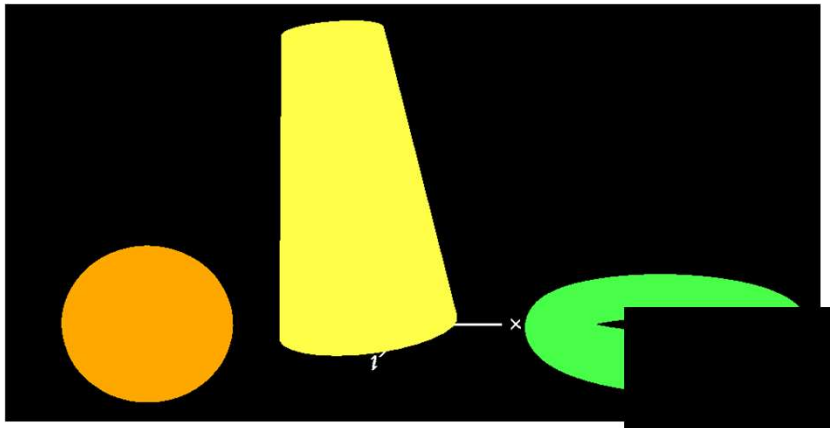
```
SphereDL = glGenLists( 1 );
glNewList( SphereDL, GL_COMPILE );
        OsuSphere(1., 32, 32);
glEndList( );

ConeDL = glGenLists(1);
glNewList(ConeDL, GL_COMPILE);
        OsuCone(1.0f, 0.5f, 3.f, 32, 32);
glEndList();

TorusDL = glGenLists(1);
glNewList(TorusDL, GL_COMPILE);
        OsuTorus(0.25f, 1., 32, 64);
glEndList();
```

In Display( ):

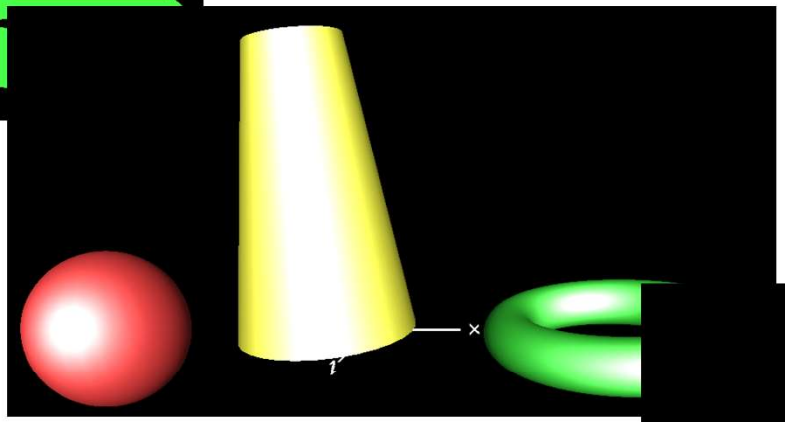
```
glColor3f(    0.8f, 0.2f, 0.2f);
SetMaterial( 0.8f, 0.2f, 0.2f, 10.f );
glCallList( SphereDL );

glColor3f(    0.8f, 0.8f, 0.2f);
SetMaterial( 0.8f, 0.8f, 0.2f, 8.f );
glCallList( ConeDL );

glColor3f(  0.2f, 0.8f, 0.2f);
SetMaterial( 0.2f, 0.8f, 0.2f, 6.f );
glCallList( TorusDL );
```

The OSU 3D Objects Can All Be…

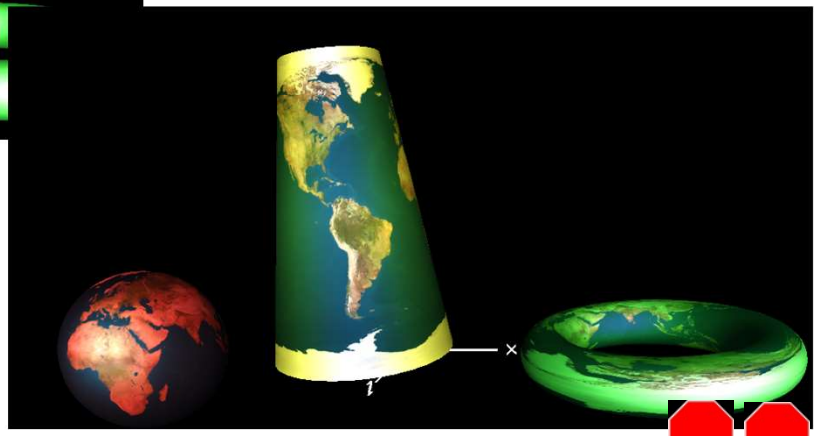
Colorized:



Lit:



Textured:



Color in Computer Graphics

Oregon State University
Computer Graphics

Your Intensity/Color Sensors

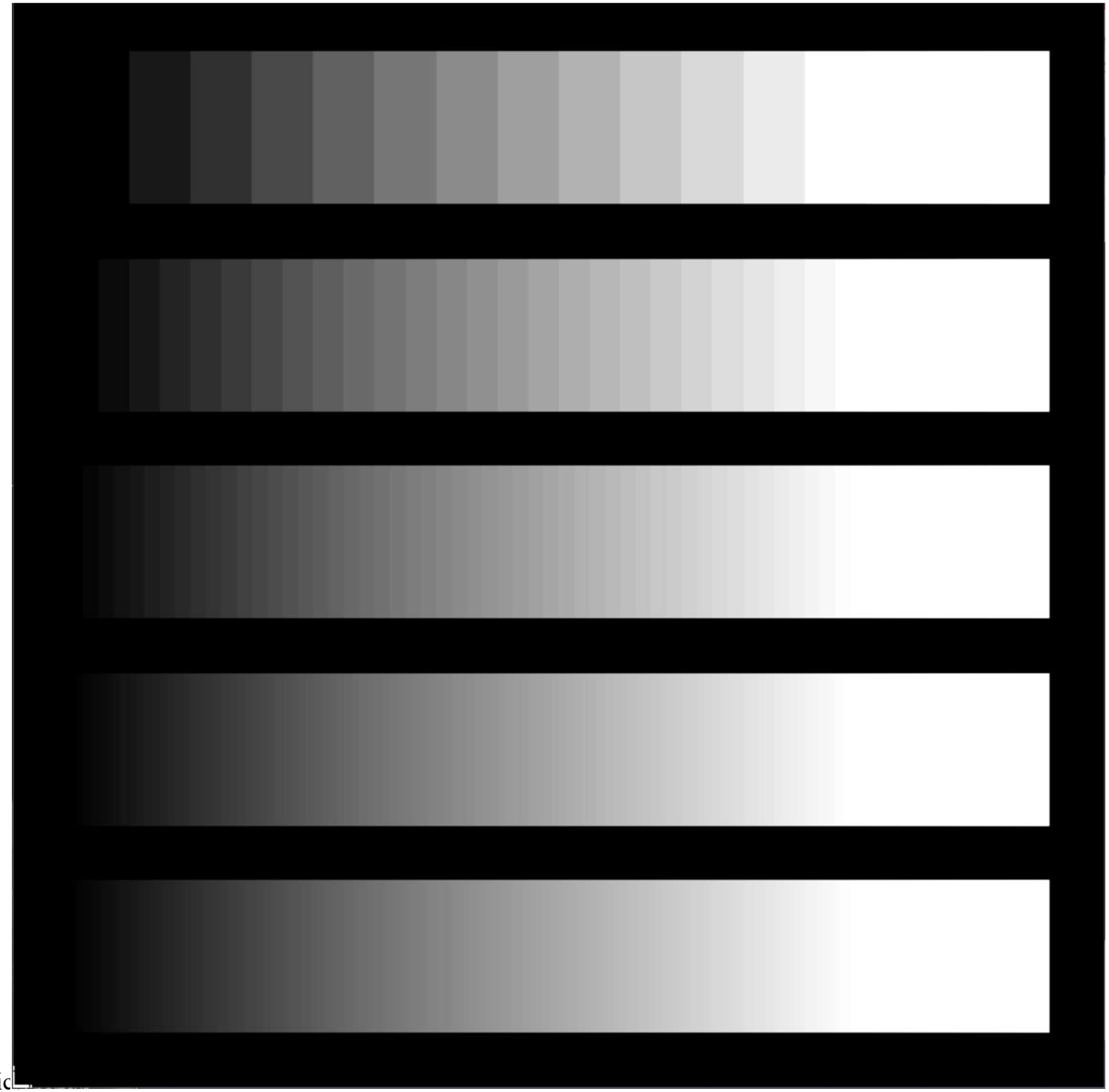
Rods

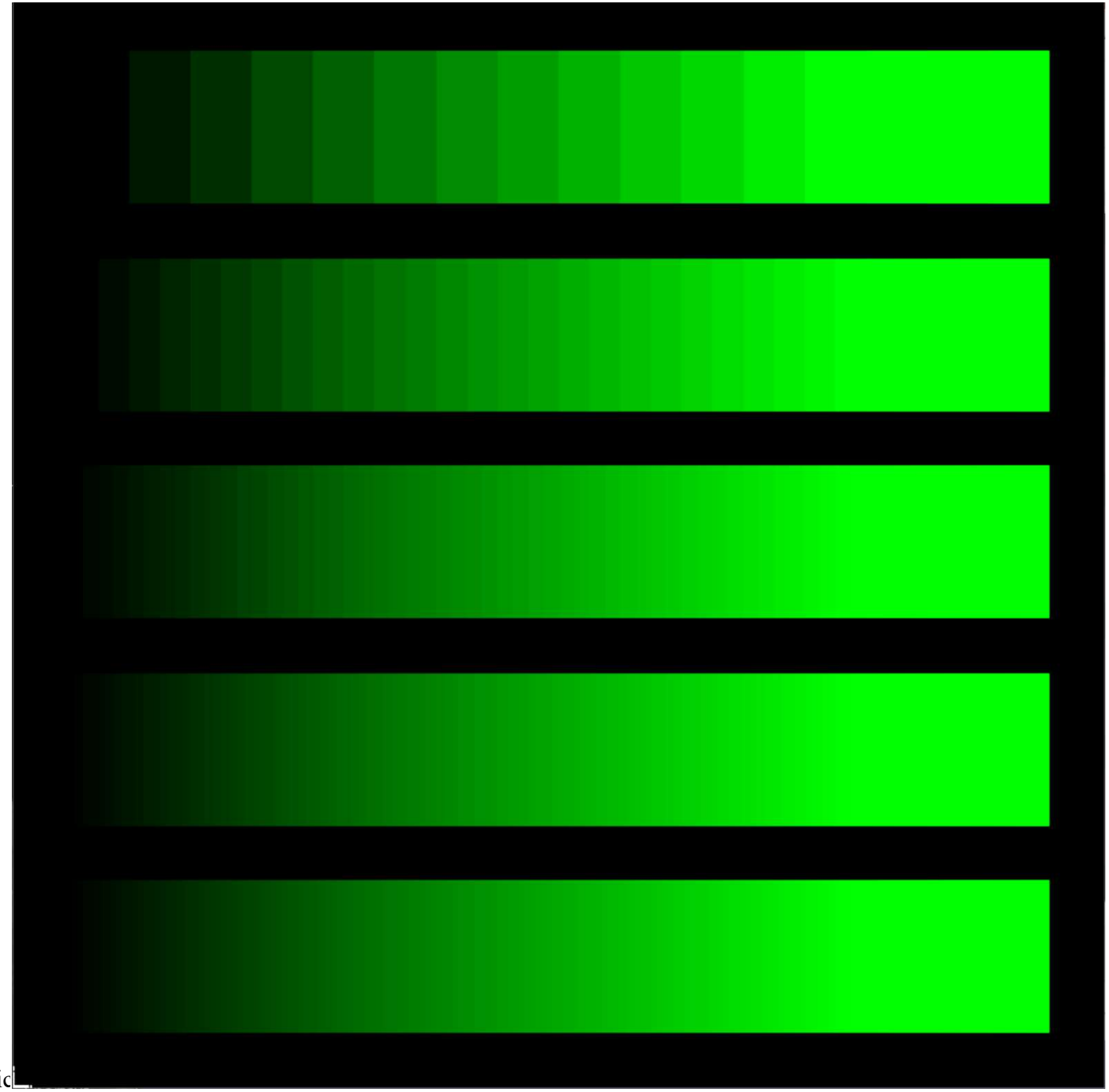
- ~115,000,000
- Concentrated on the *periphery* of the retina
- Sensitive to *intensity*
- Most sensitive at 500 nm (~green)

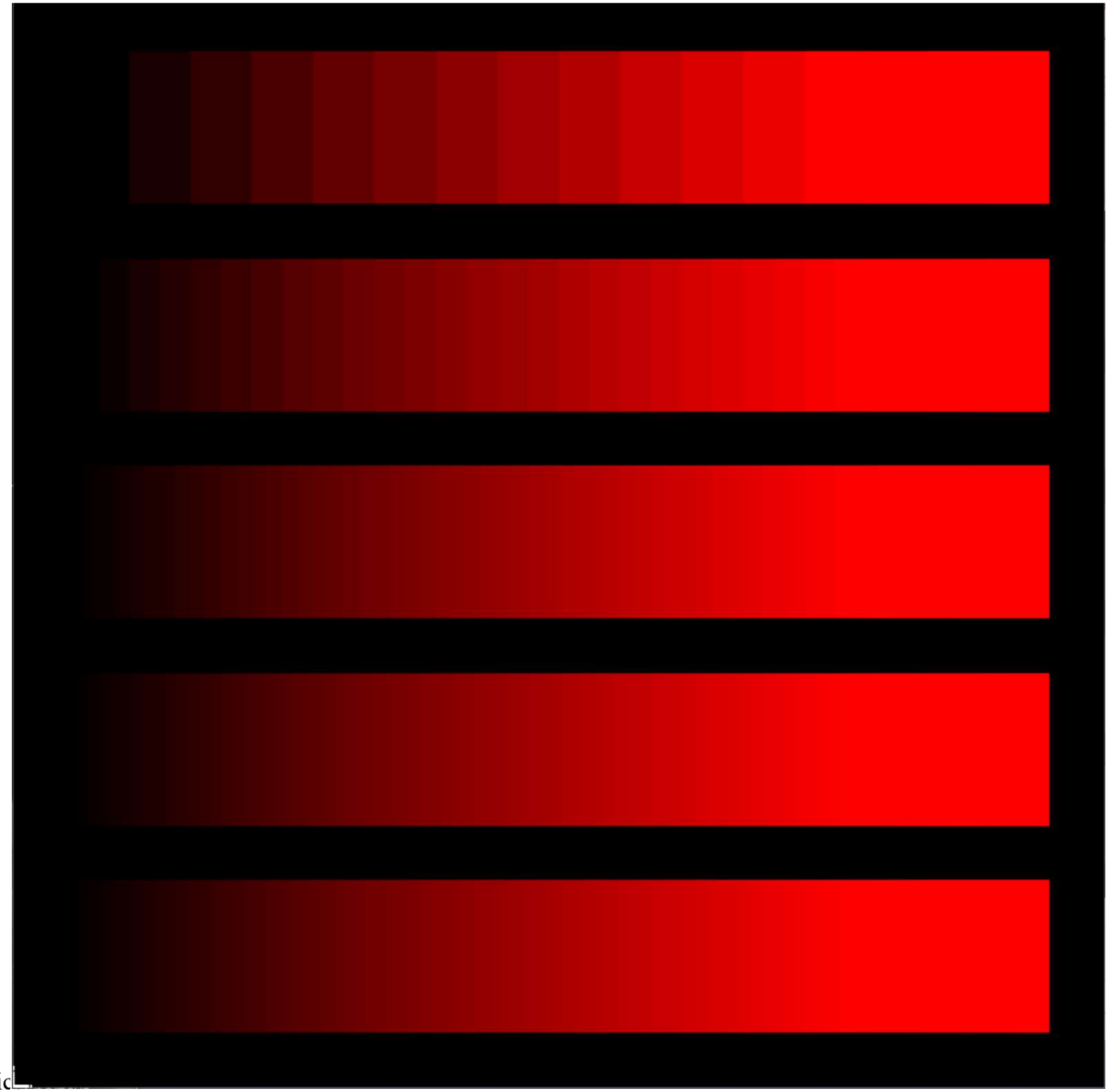
Cones

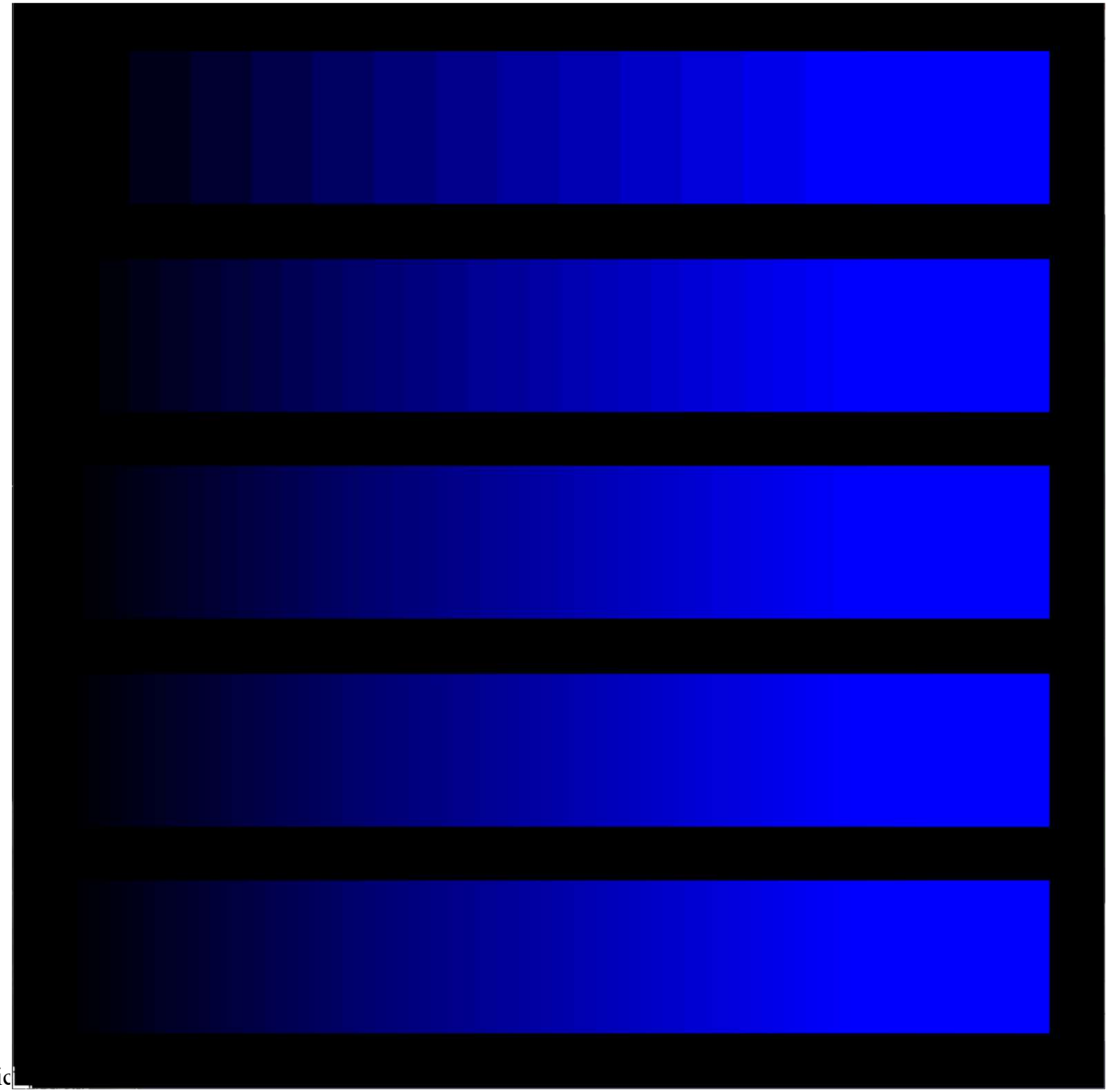
- ~7,000,000
- Concentrated near the *center* of the retina
- Sensitive to *color*
- Three types of cones: long(~red), medium (~green), and short (~blue) wavelengths

But are you equally-sensitive to all wavelengths?









Sidebar: How Many Pixels Do You Need?

A person with 20/20 vision has a visual acuity of:
1 arc-minute = 1/60°

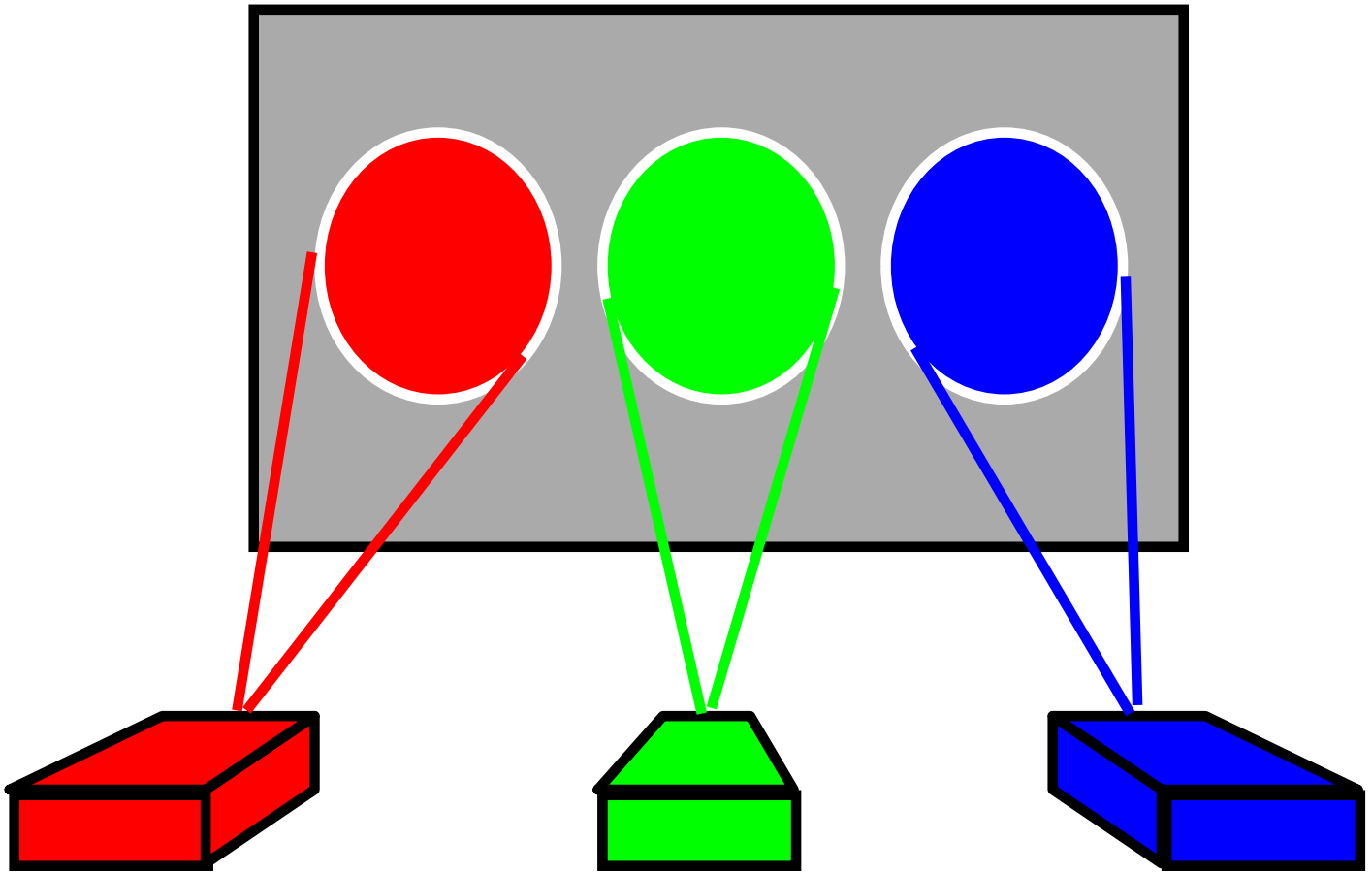
$$\Theta = {}^{1}/_{60}{}^{\circ} = .00029^{R}$$

$$\text{Density} = \frac{1}{D\Theta}$$

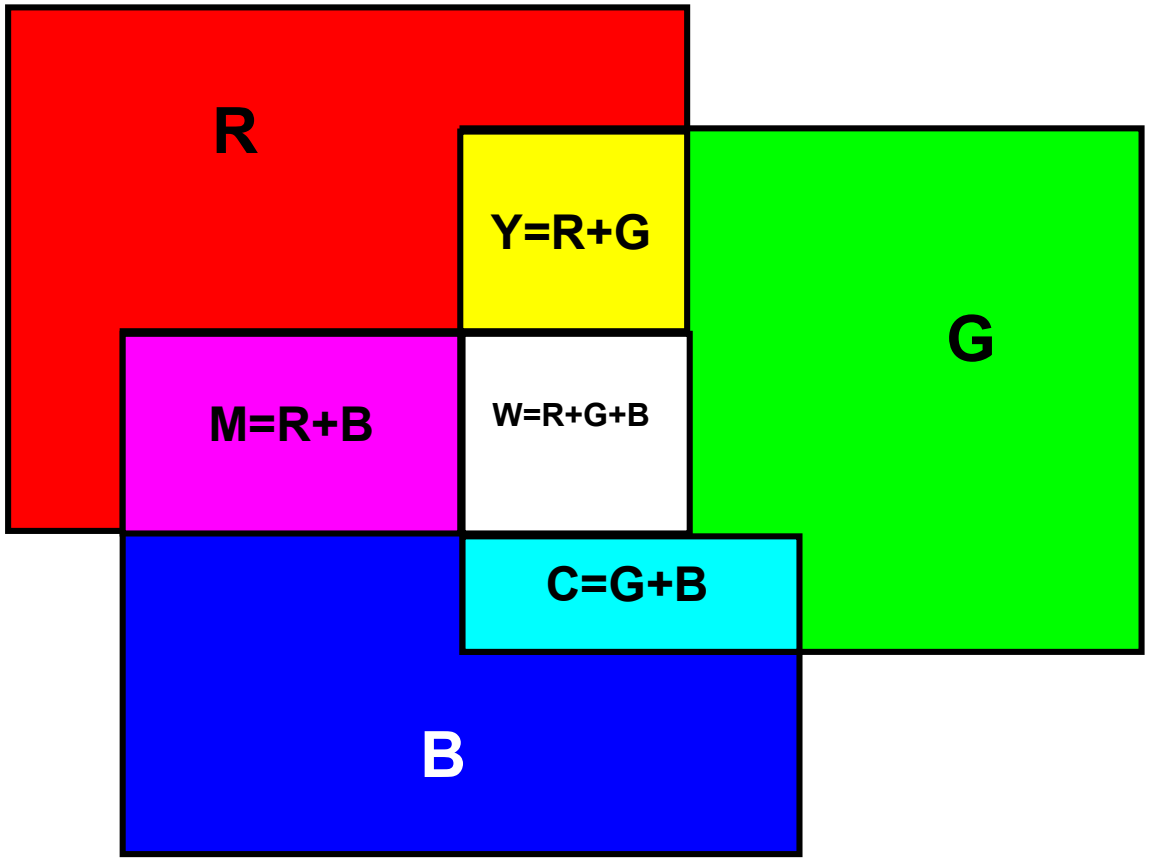
Viewing Distance (inches)	Required Pixel Density (ppi)	If the monitor's resolution is 1600 x 1200, then its diagonal size would need to be:
36	95	21”
31	111	18”
24	143	14”
12	286	7”
9	400	5”
6	600	3”

Oregon State University
Computer Graphics

Monitors: Additive Colors



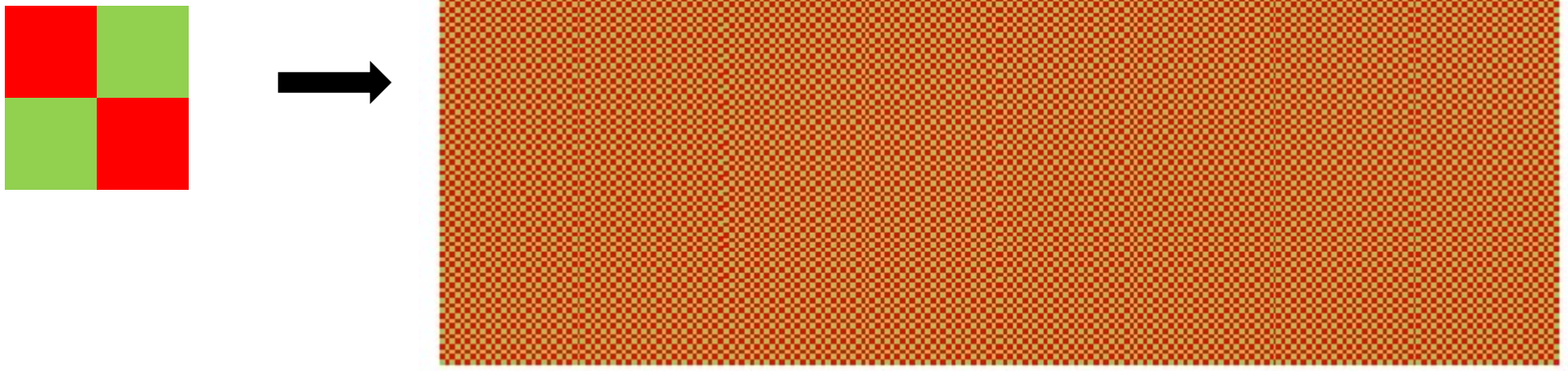
Additive Color (RGB)



OpenGL: → **glColor3f(r, g, b);**

0. ≤ r, g, b ≤ 1.

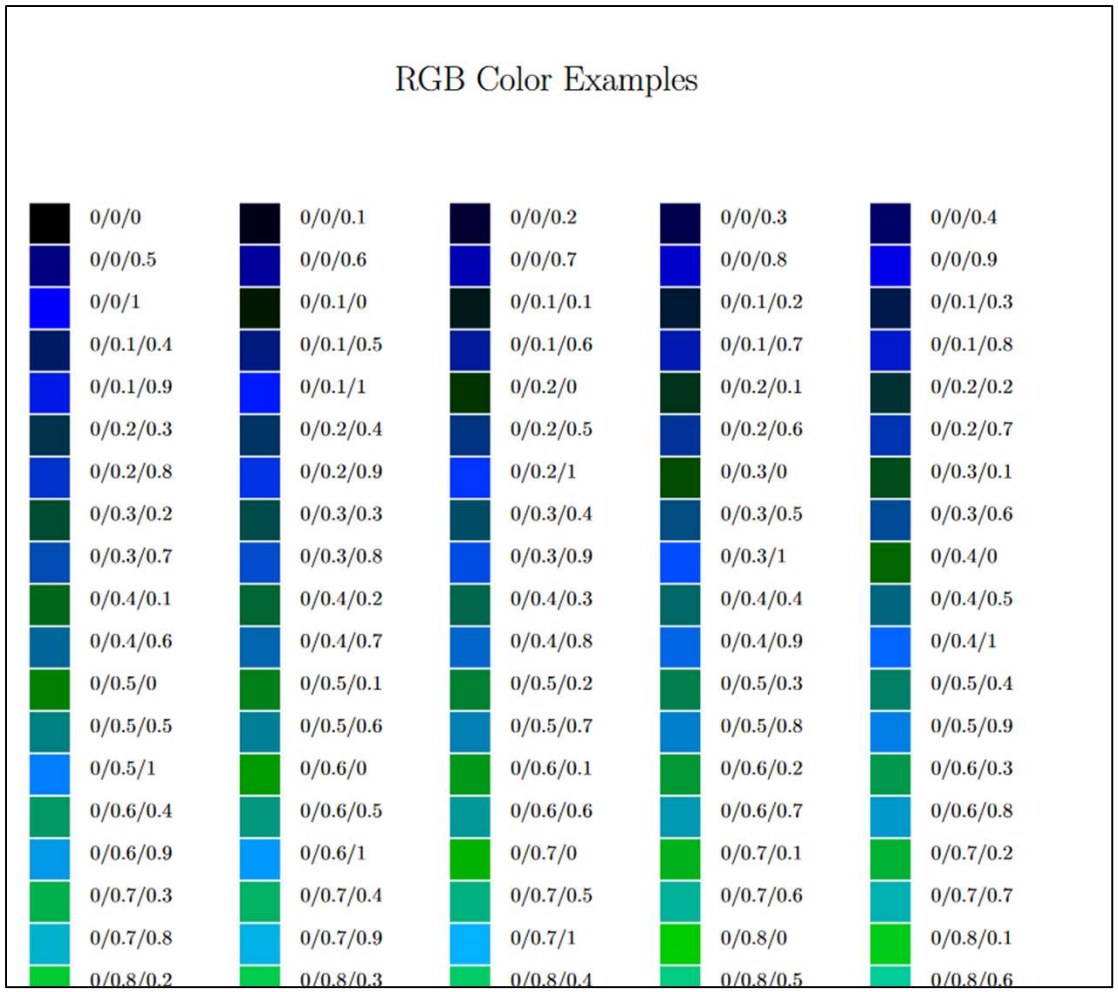
Yes, Our Vision System Really Does Mush Red and Green Together to Make Yellow!



Color Combinations

Here's a cool website that shows a lot of different color combinations:

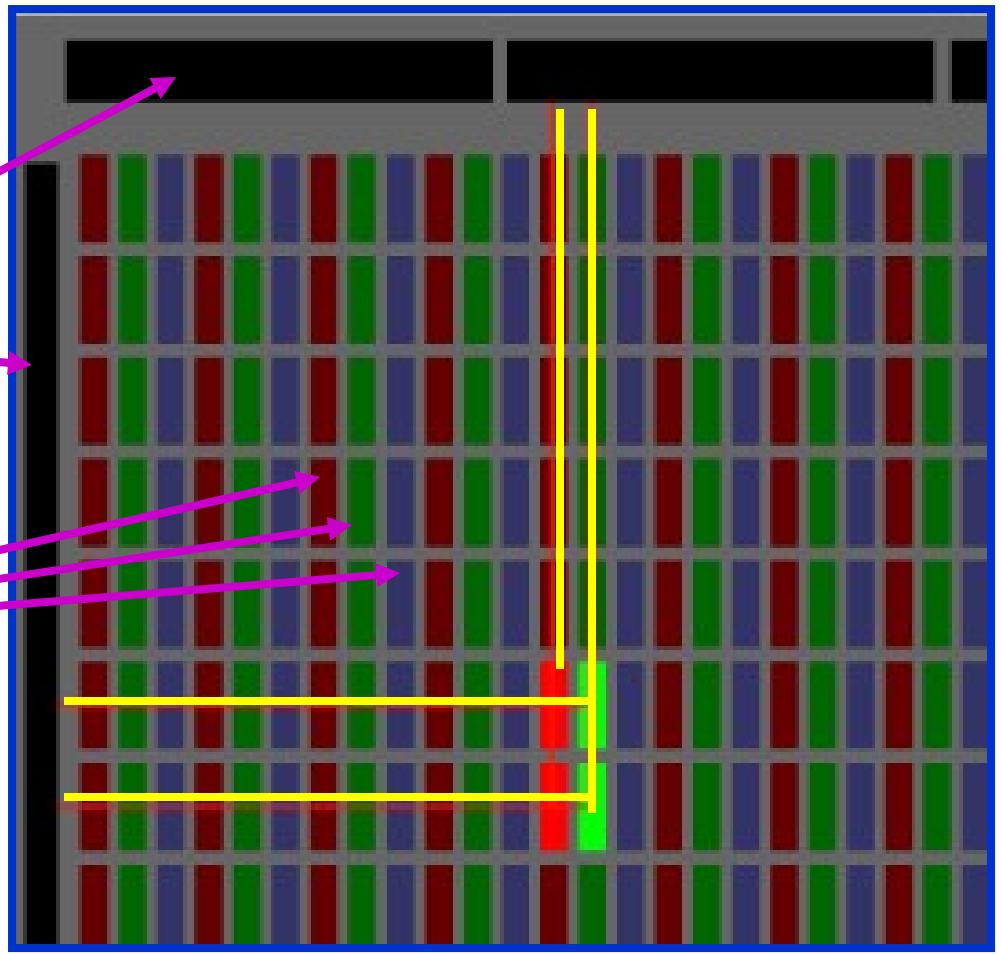
https://www.tug.org/pracjourn/2007-4/walden/color.pdf



LCD Displays "Gate" Color

Most desktop monitors are LCD displays that use white LEDs for backlighting

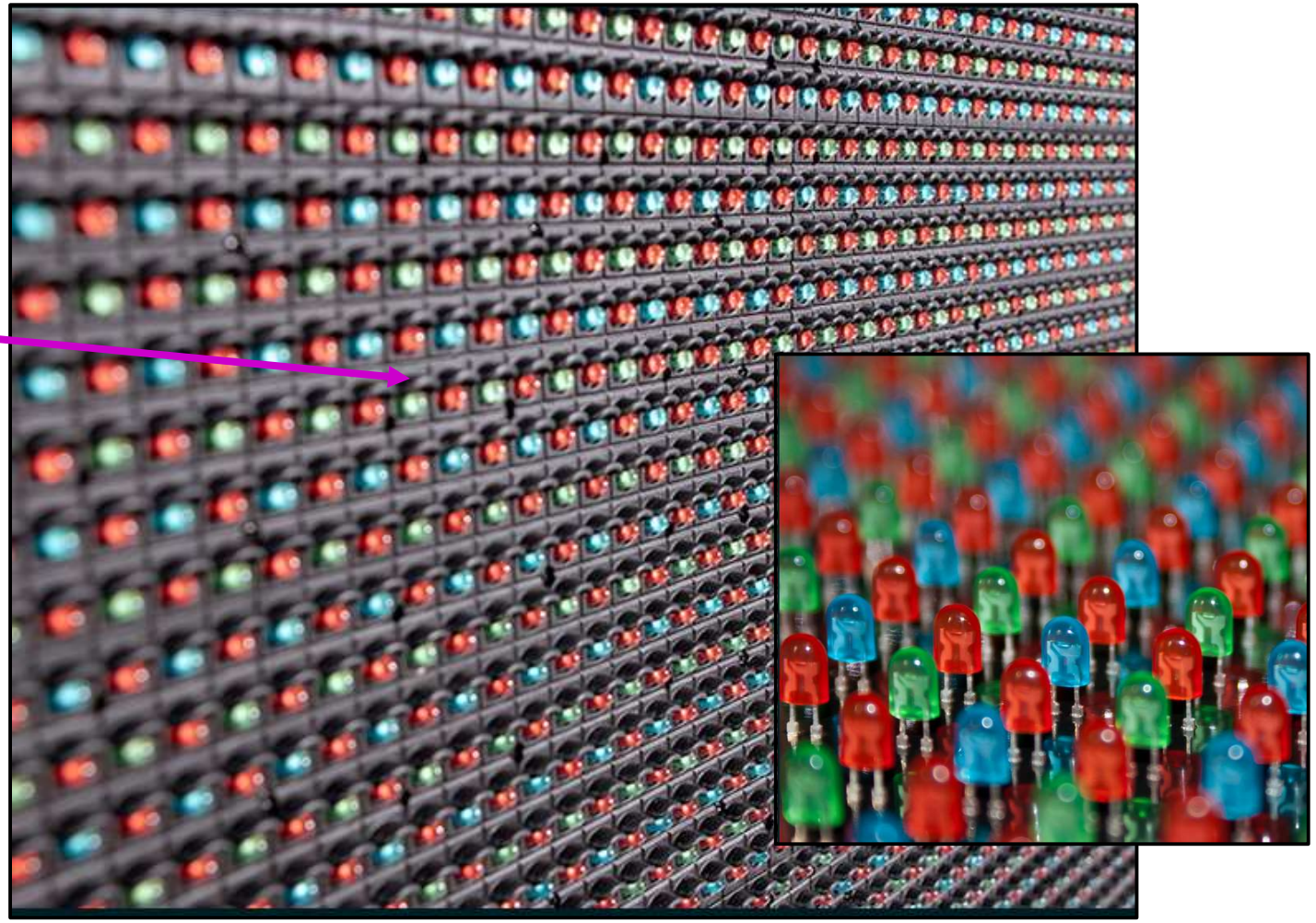
- **Grid of electrodes**
- **Color filters**



http://electronics.howstuffworks.com

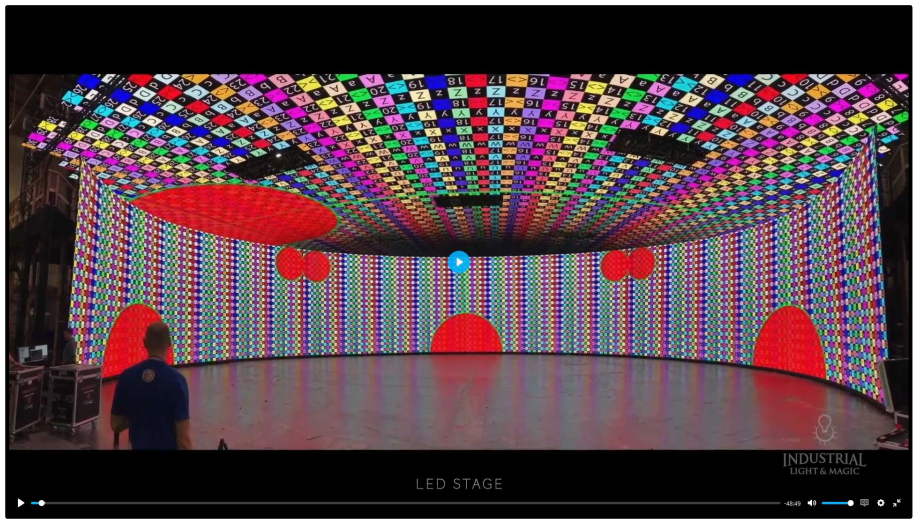
LED Displays *Emit* Color

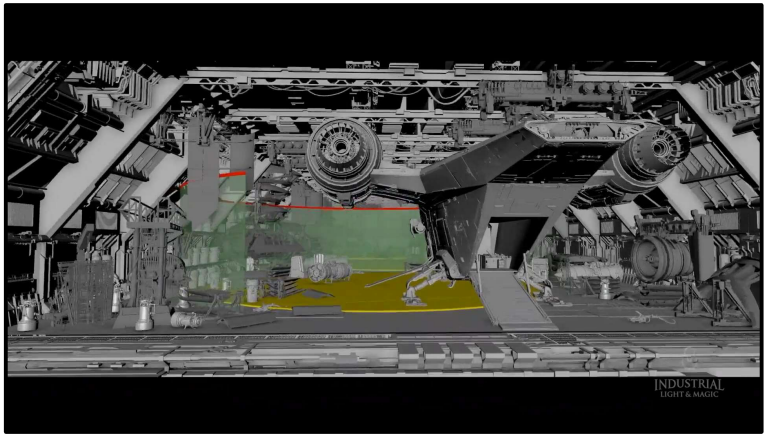
- **Grid of LEDs**



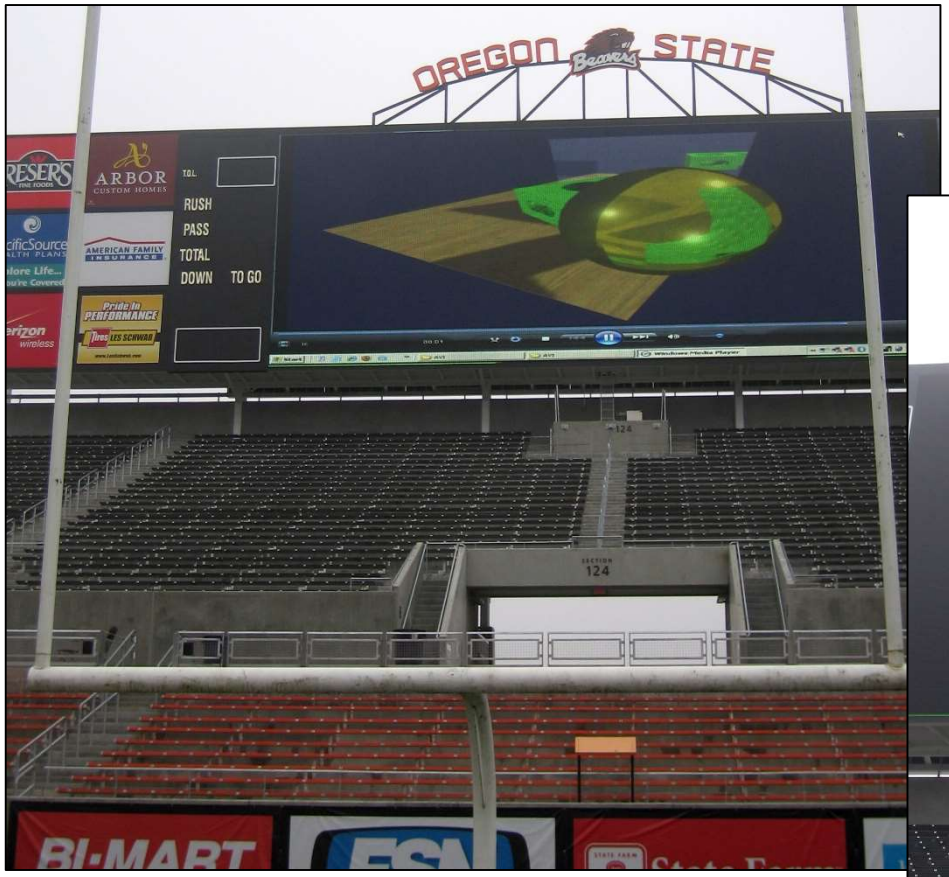
https://www.expromo.eu/en/led-display-facts/

The New Sound Stages use LED Displays



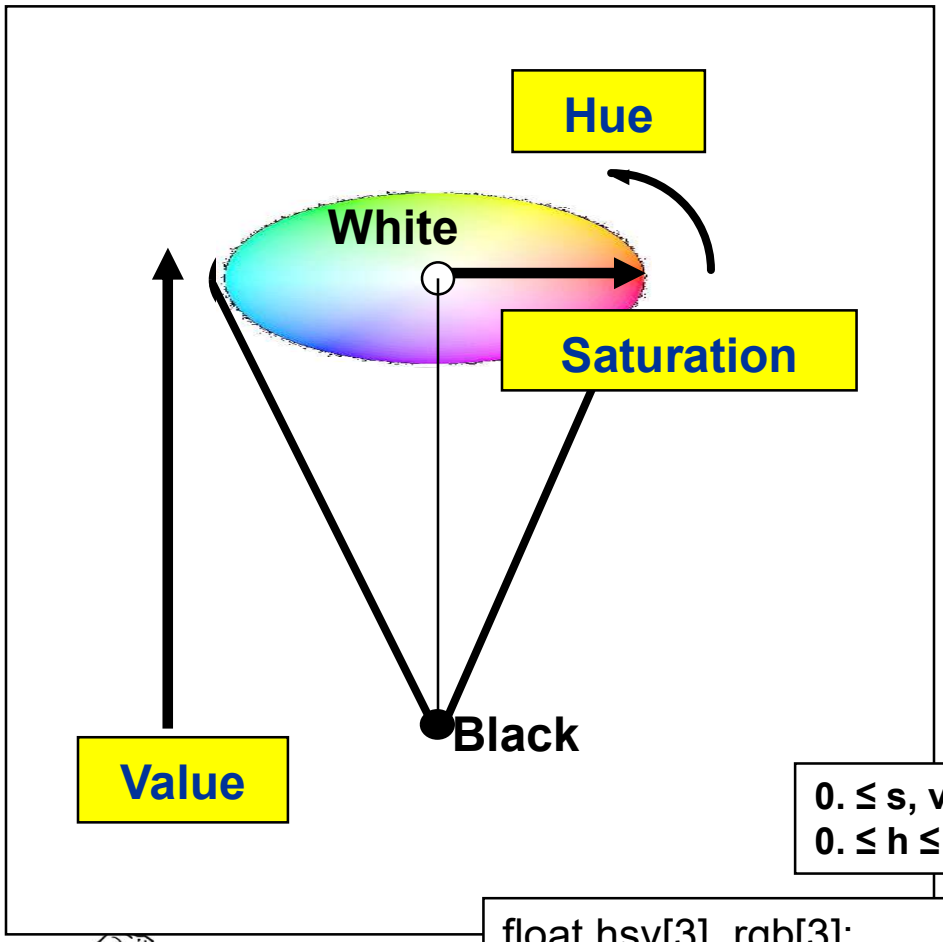


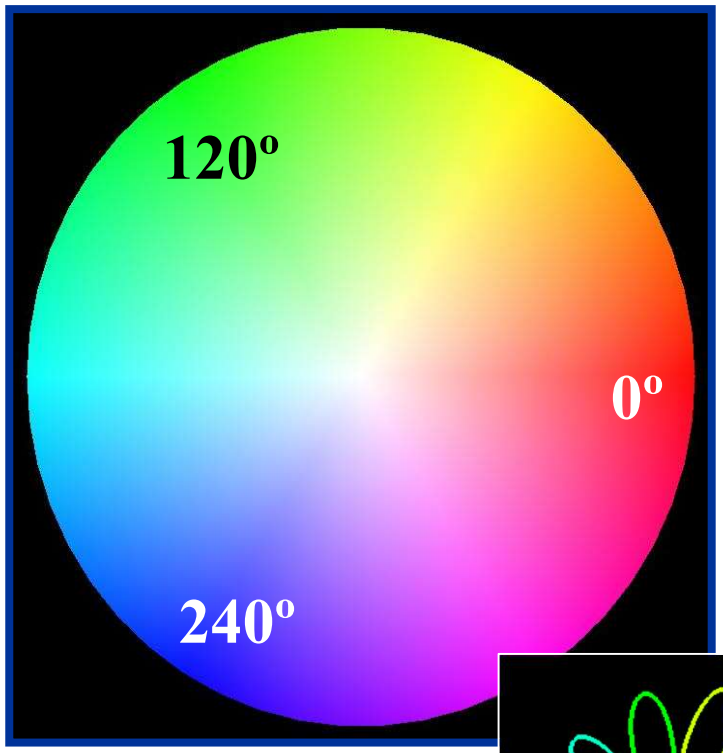
Stadium Jumbotrons use LED Displays

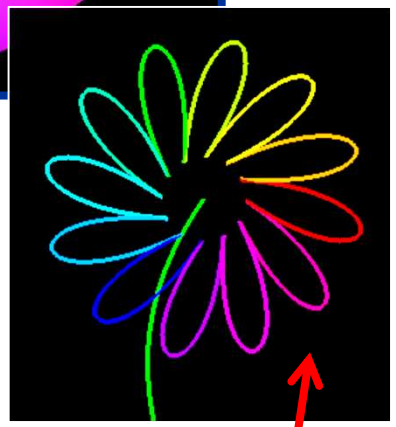


Oregon State University
Computer Graphics

Hue-Saturation-Value (HSV):
For many applications, a more intuitive way to specify additive color







0. ≤ s, v, r, g, b ≤ 1.
0. ≤ h ≤ 360.

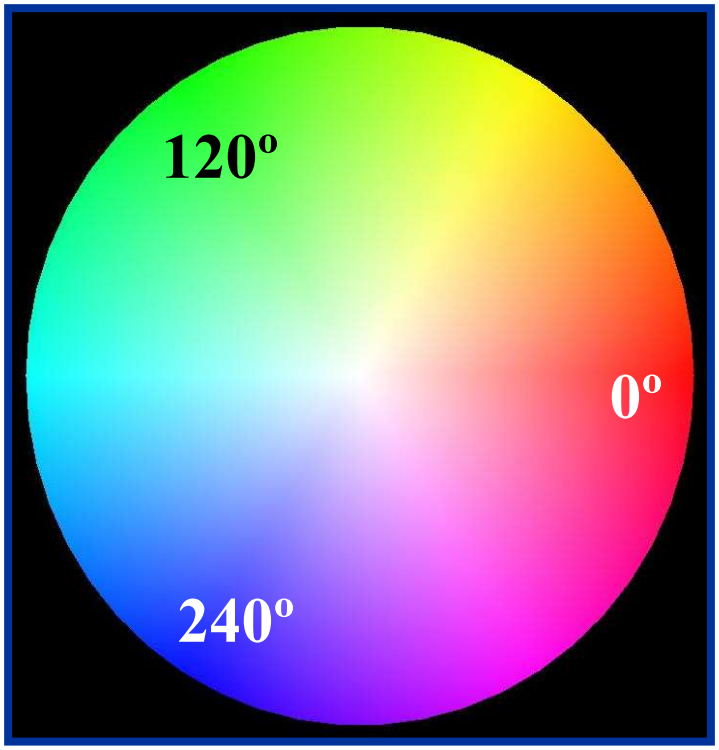
```
float hsv[3], rgb[3];
hsv[0] = something between 0. and 360.
hsv[1] = hsv[2] = 1.;
HsvRgb( hsv, rgb );
glColor3fv( rgb );
```

The HsvRgb( ) function is in your sample code

Marching around the Hue color wheel is a nice way to get a range of colors

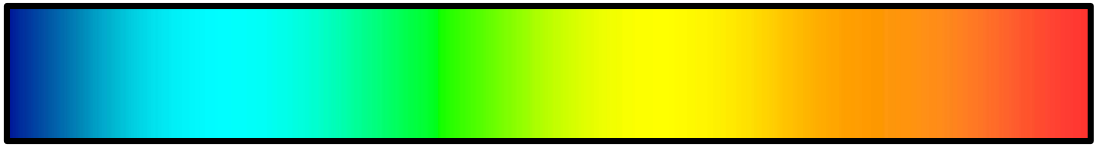
Home Depot uses a form of HSV :-)

Hue-Saturation-Value (HSV):
For many vis applications, a simpler way to specify additive color



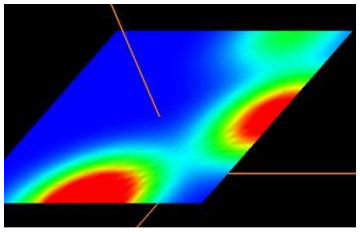
Notice that blue-green-red in HSV space corresponds to the visible portion of the electromagnetic spectrum

Blue: 380 nm **Green: 520 nm** **Red: 780 nm**

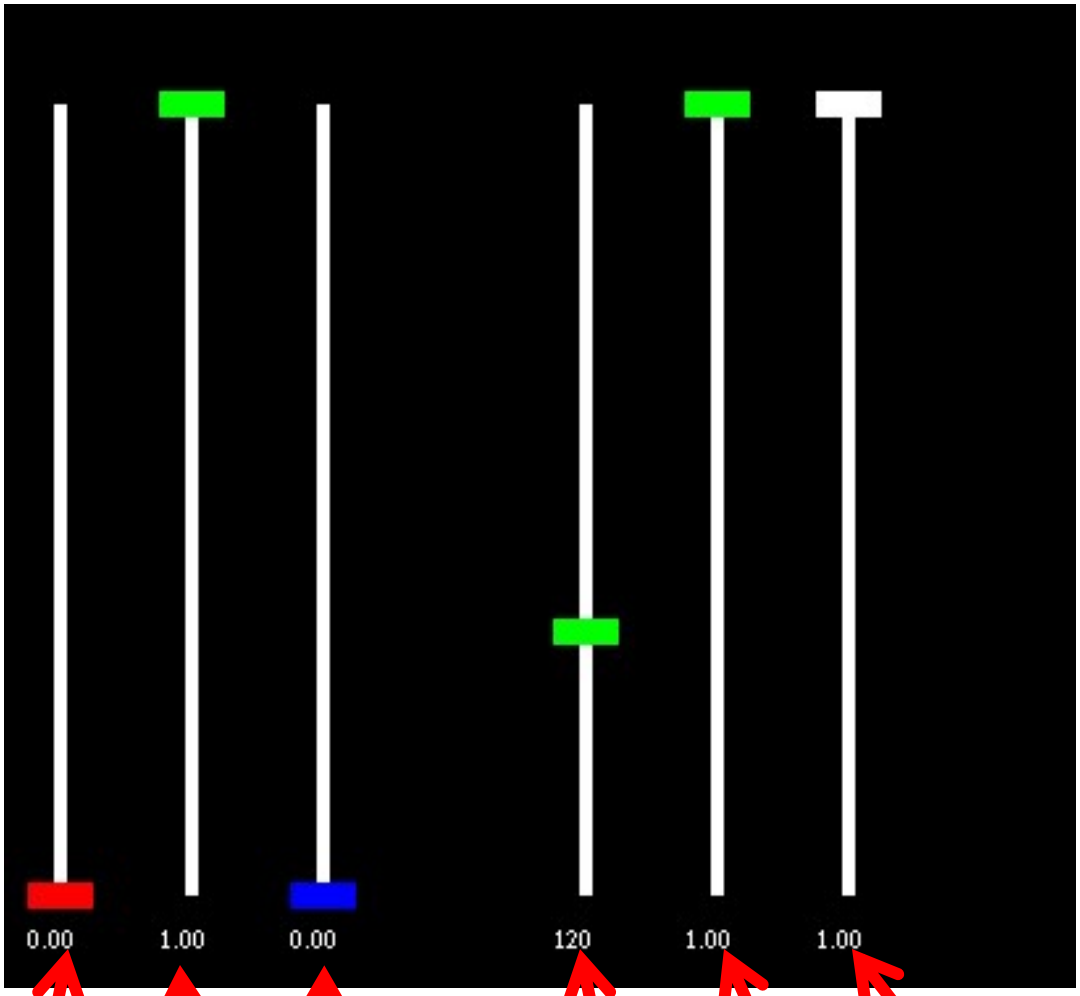


$$Hue = 240. - 240.\frac{S - S_{\min}}{S_{\max} - S_{\min}}$$

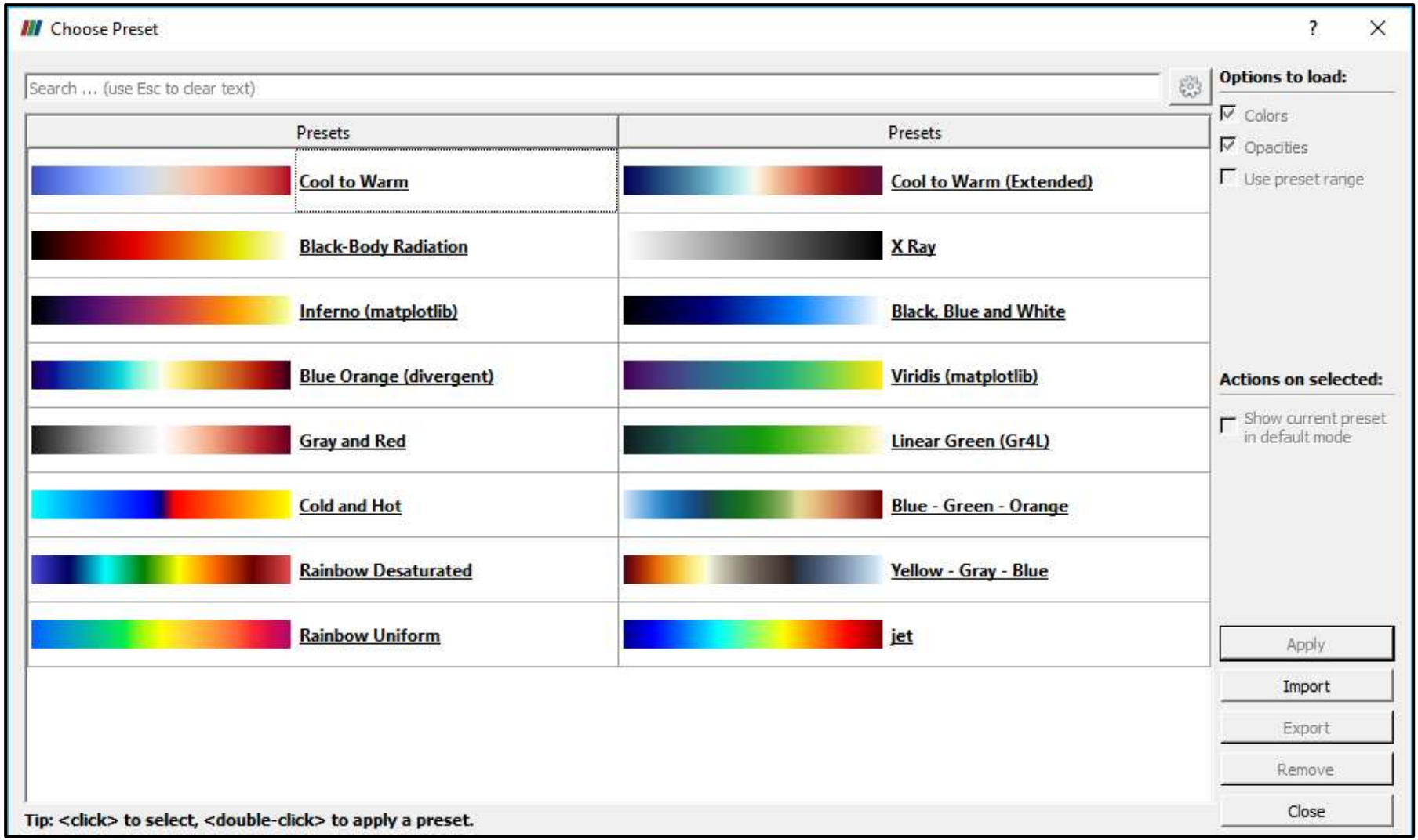
Turning a scalar value into a hue when using the Rainbow Color Scale



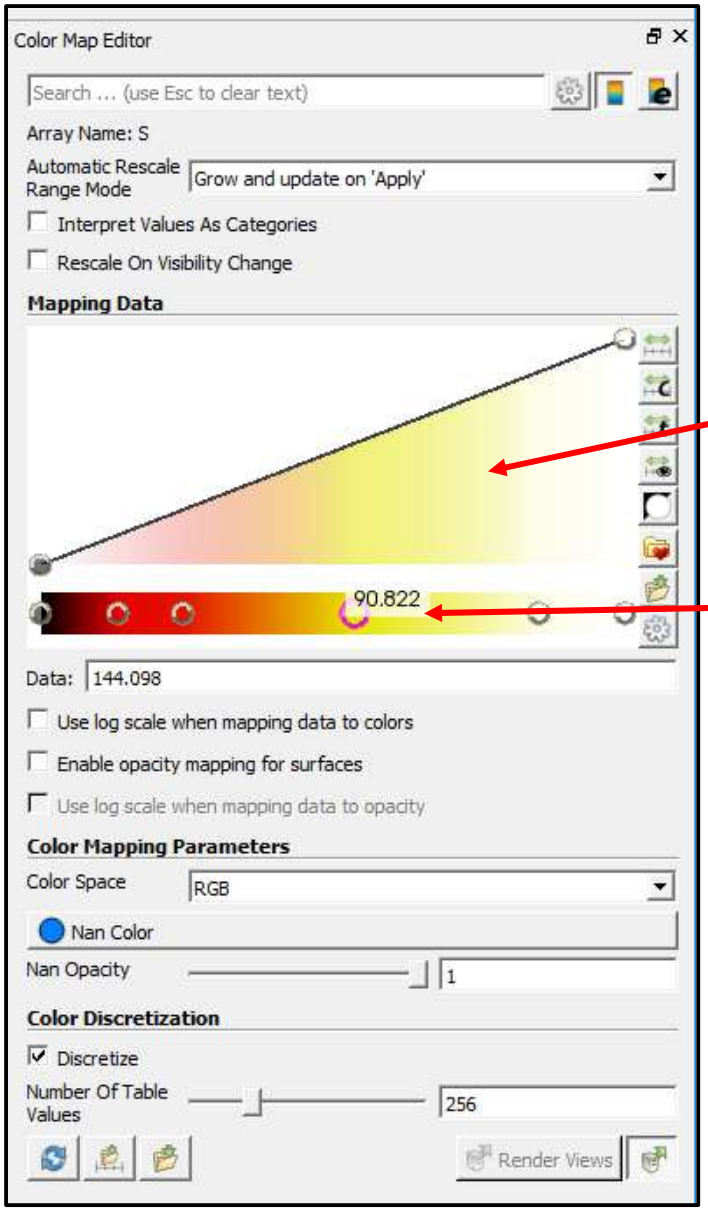
Hue-Saturation-Value:
The *OSU ColorPicker* Program



ParaView Allows You to Pick Among Several Preset Color Ranges



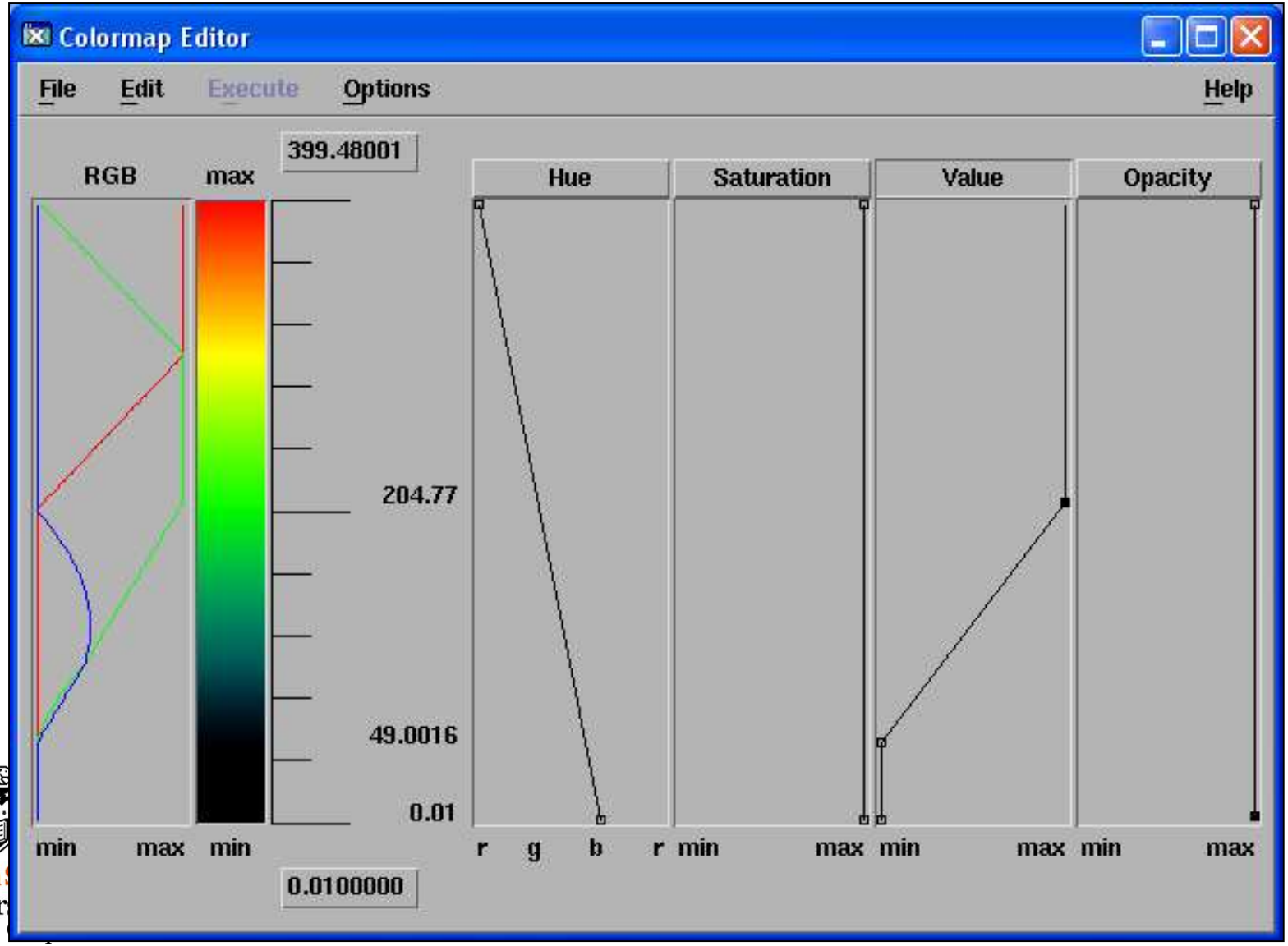
ParaView Allows You to Sculpt Your Own Color Range



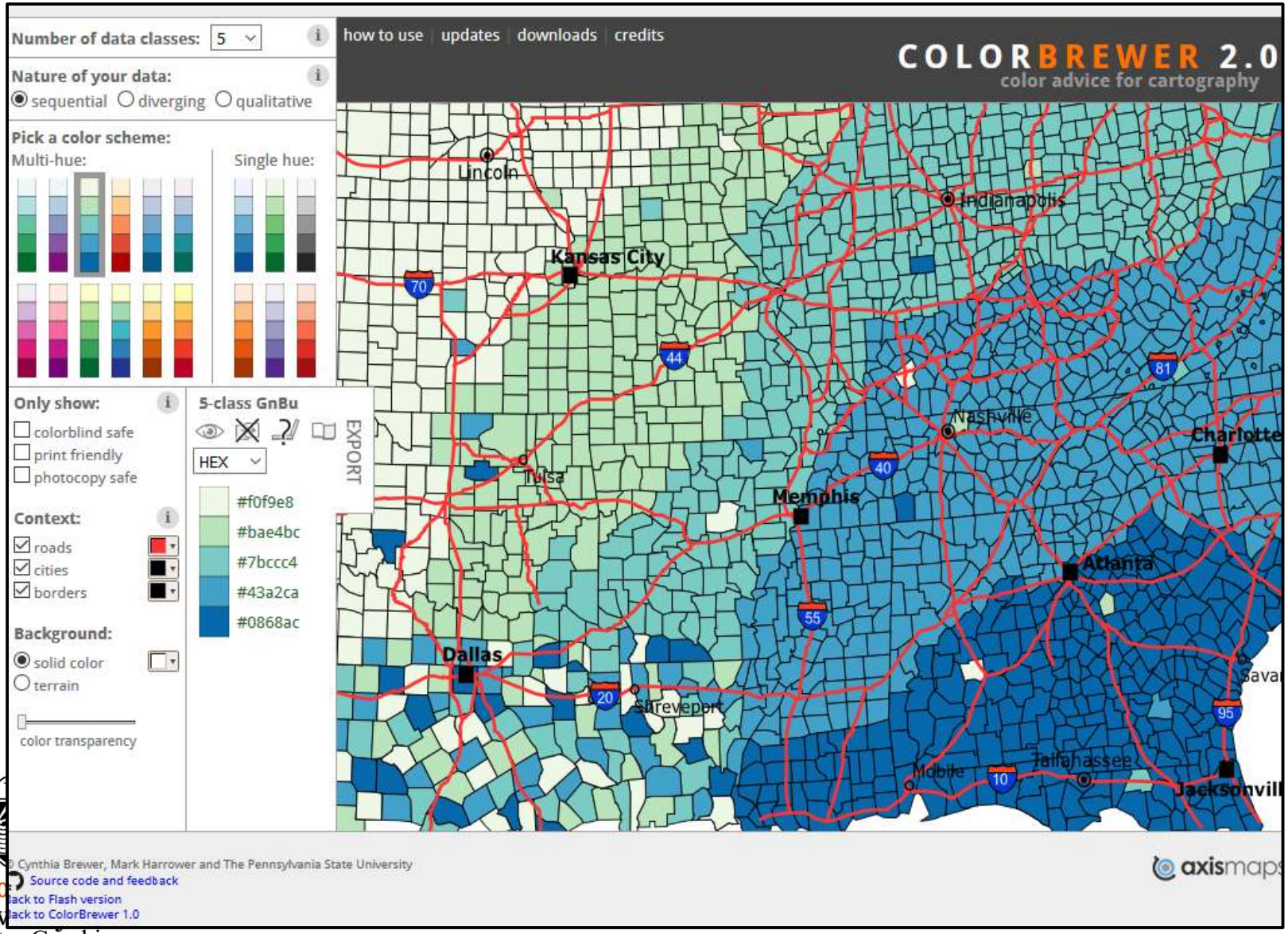
Opacity

Color

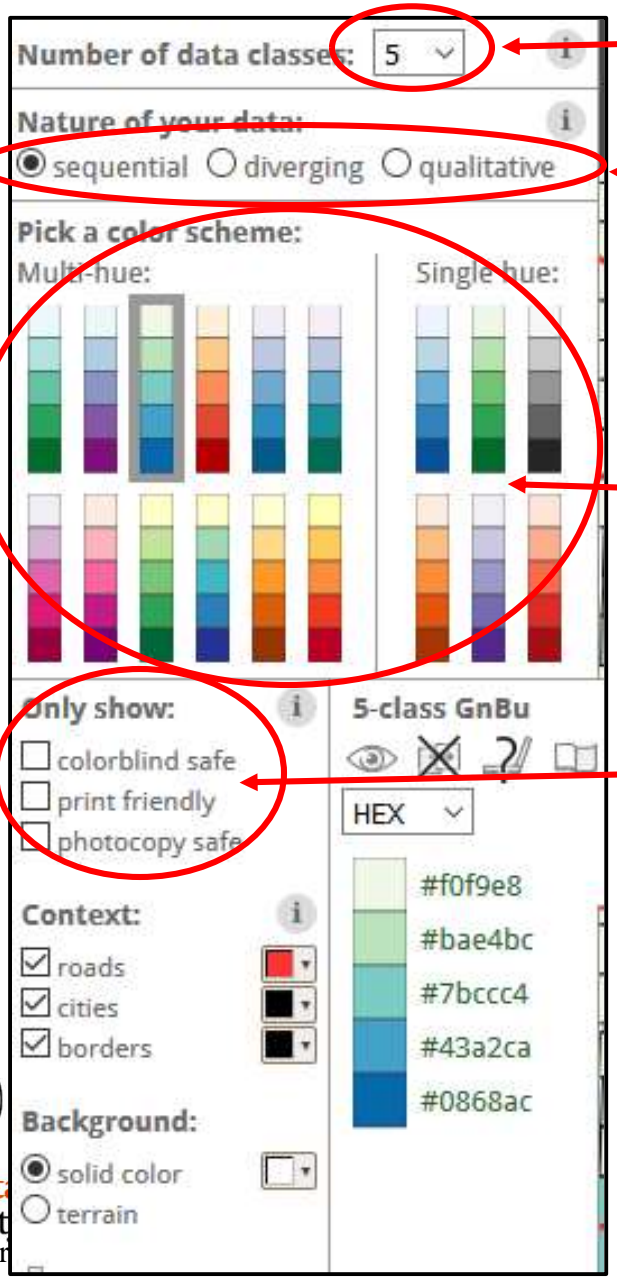
OpenDX Allows you to Sculpt the Transfer Function in HSV



http://colorbrewer2.org



http://colorbrewer2.org



Number of discrete colors needed

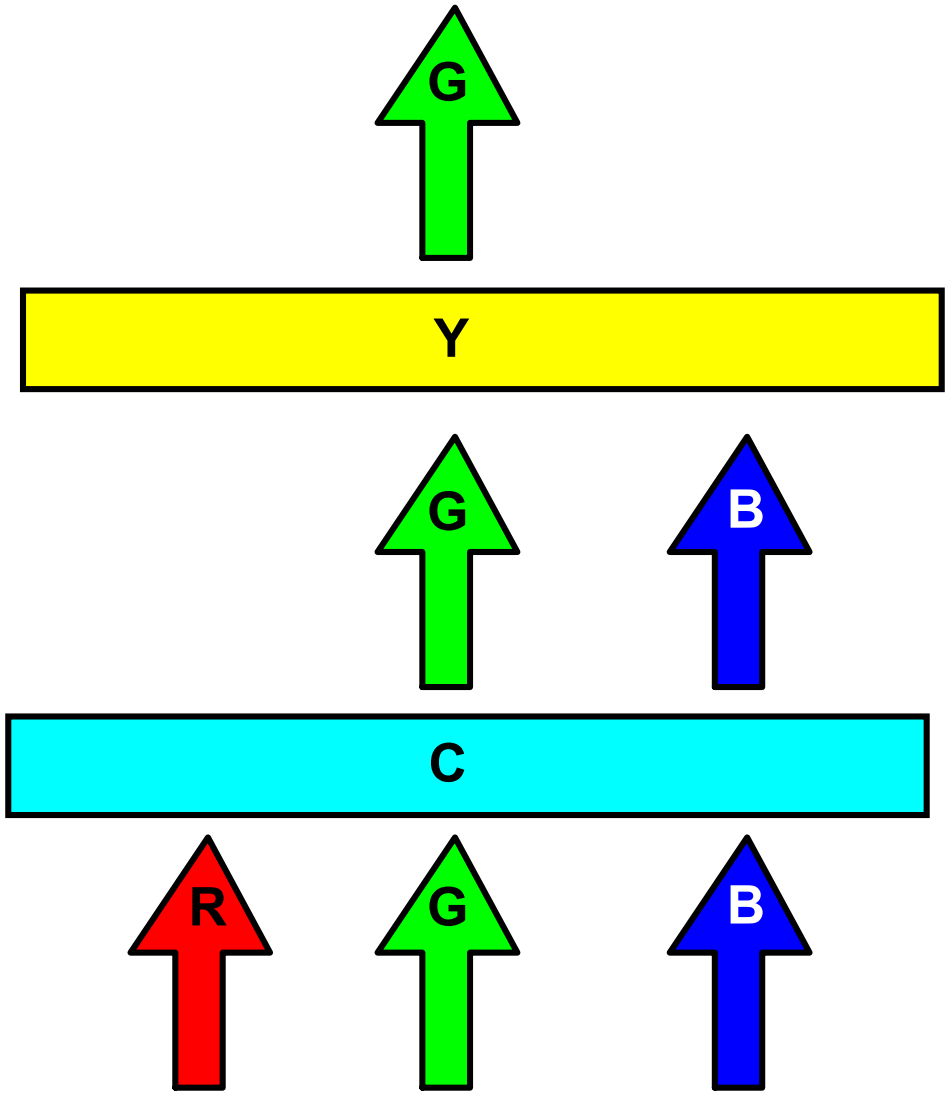
Type of data

Color schemes

Ways of restricting the color schemes (the ***colorblind safe*** option is especially important!)

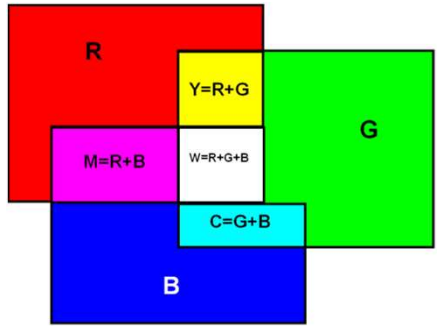
A good way to explore discrete color spaces

Subtractive Colors (CMYK)

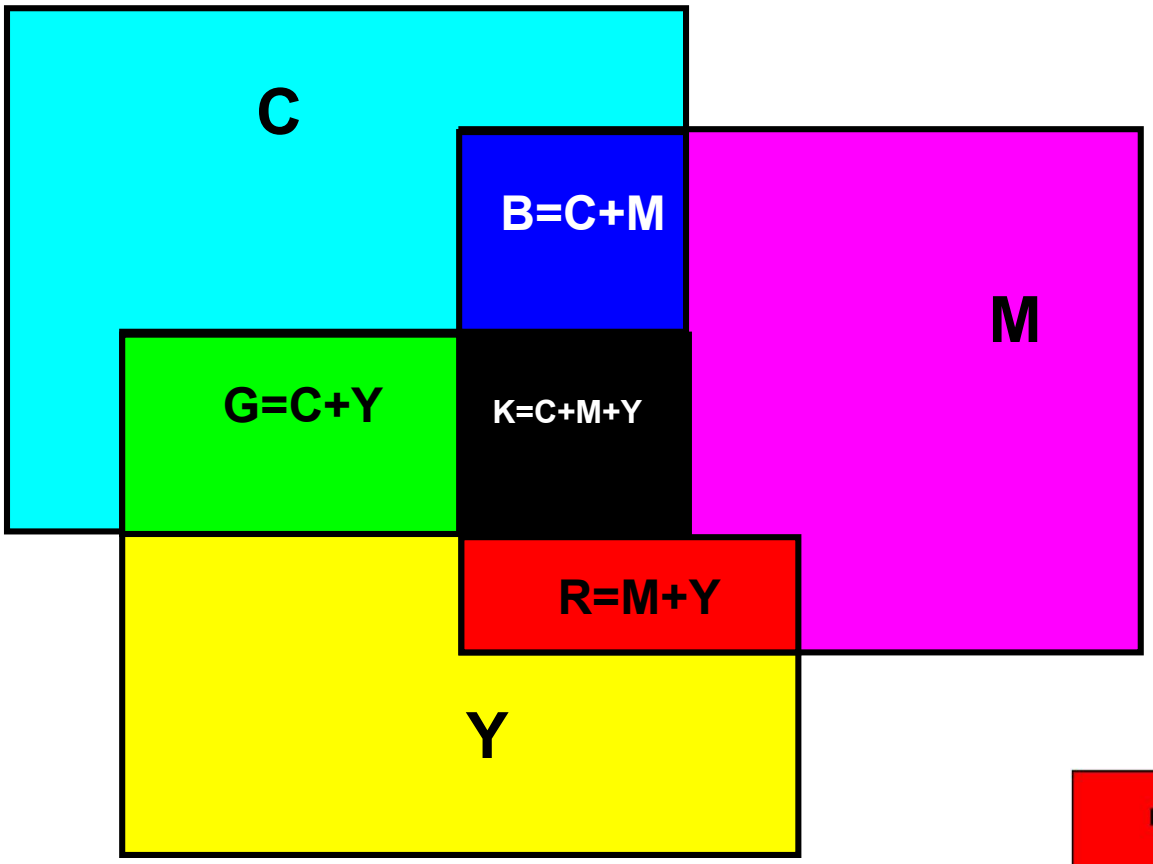


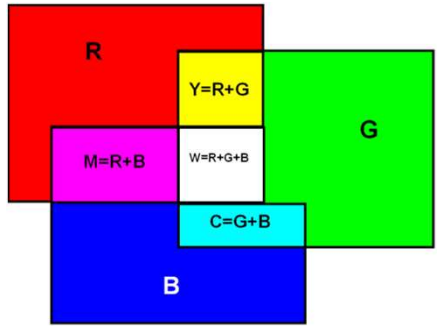
R = Red
G = Green
B = Blue
W = White

C = Cyan
M = Magenta
Y = Yellow
K = Black



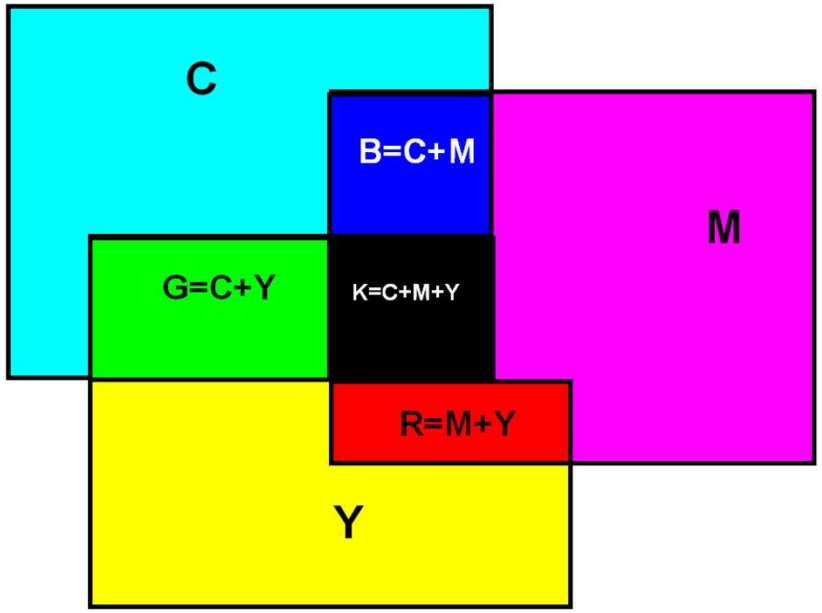
Subtractive Colors (CMYK)



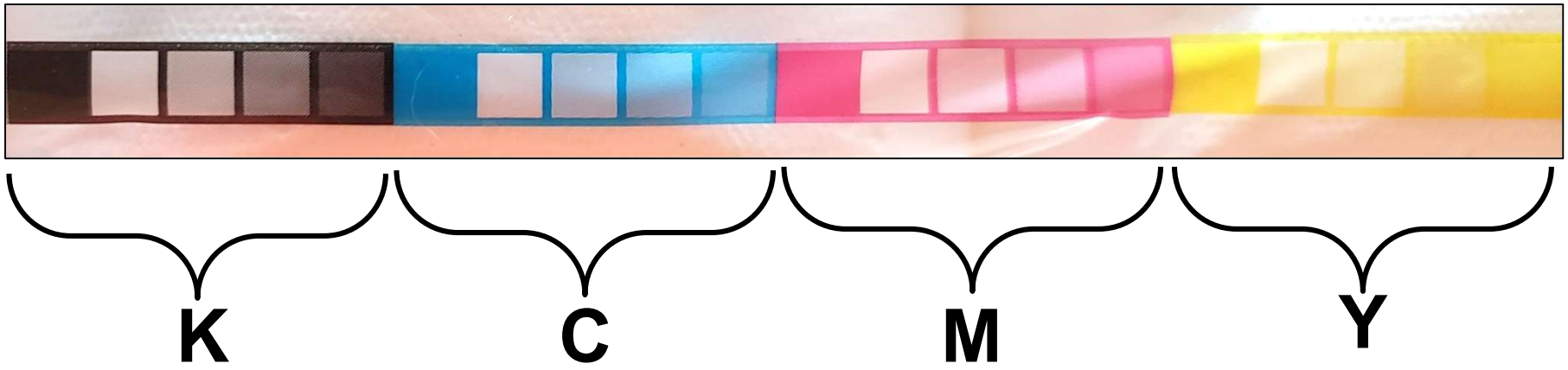


Color Printing

- Uses subtractive colors
- Uses 3 (CMY) or 4 (CMYK) passes
- CMYK printers have a better-looking black
- There is a considerable variation in color *gamut* between products

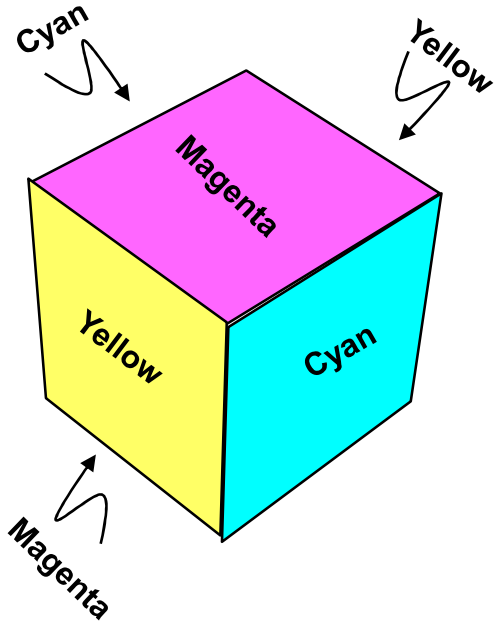


You Often See Color Printing QA Tests Like This

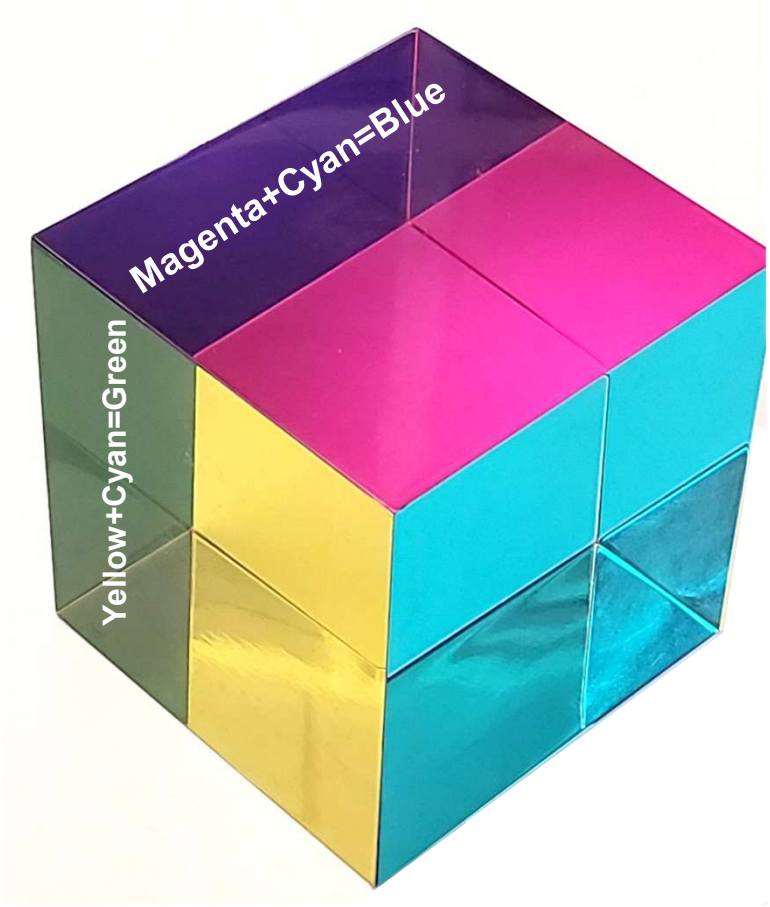


The CMY Cube

How the Cube is setup:

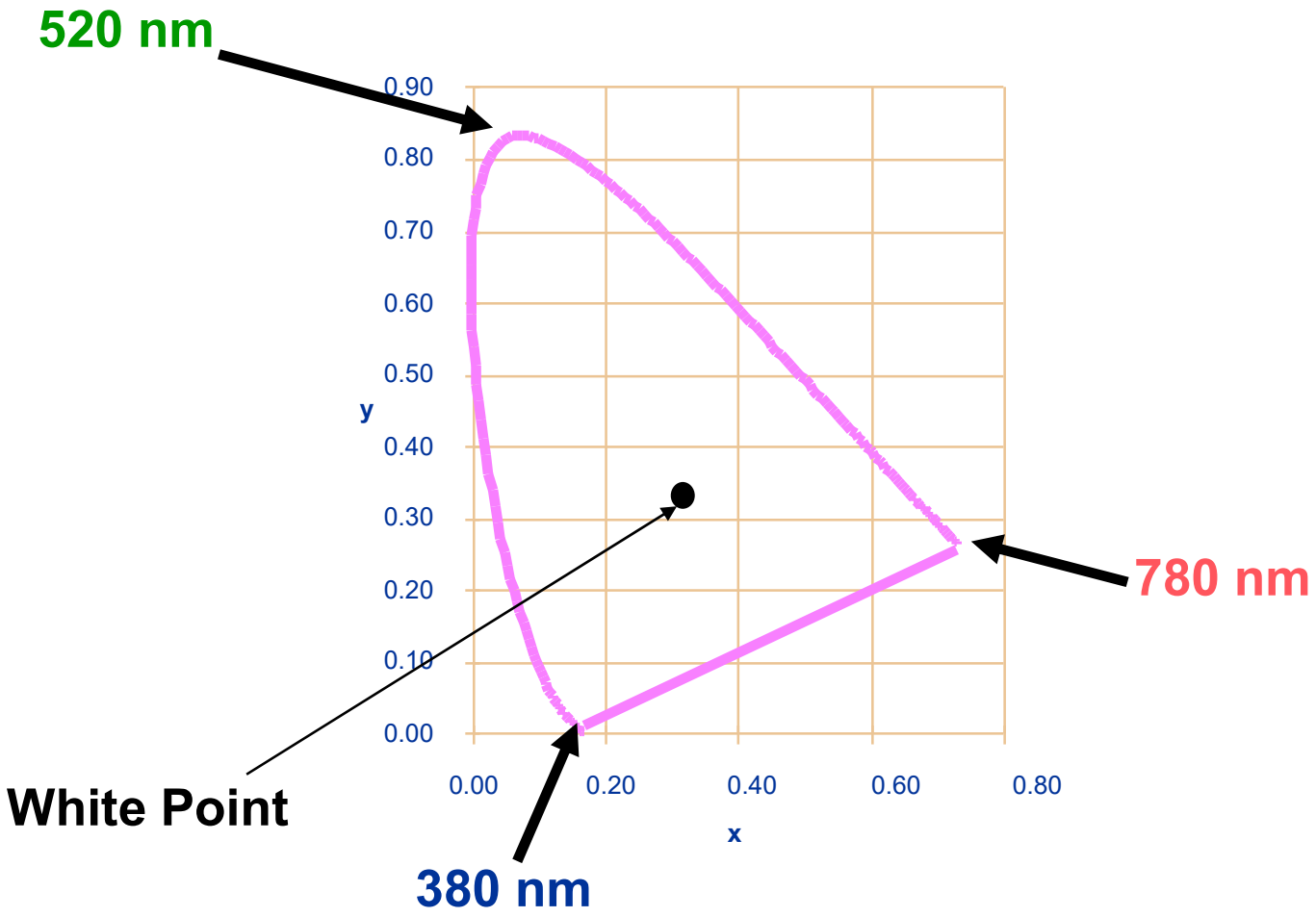


How it looks when you sight through two faces:

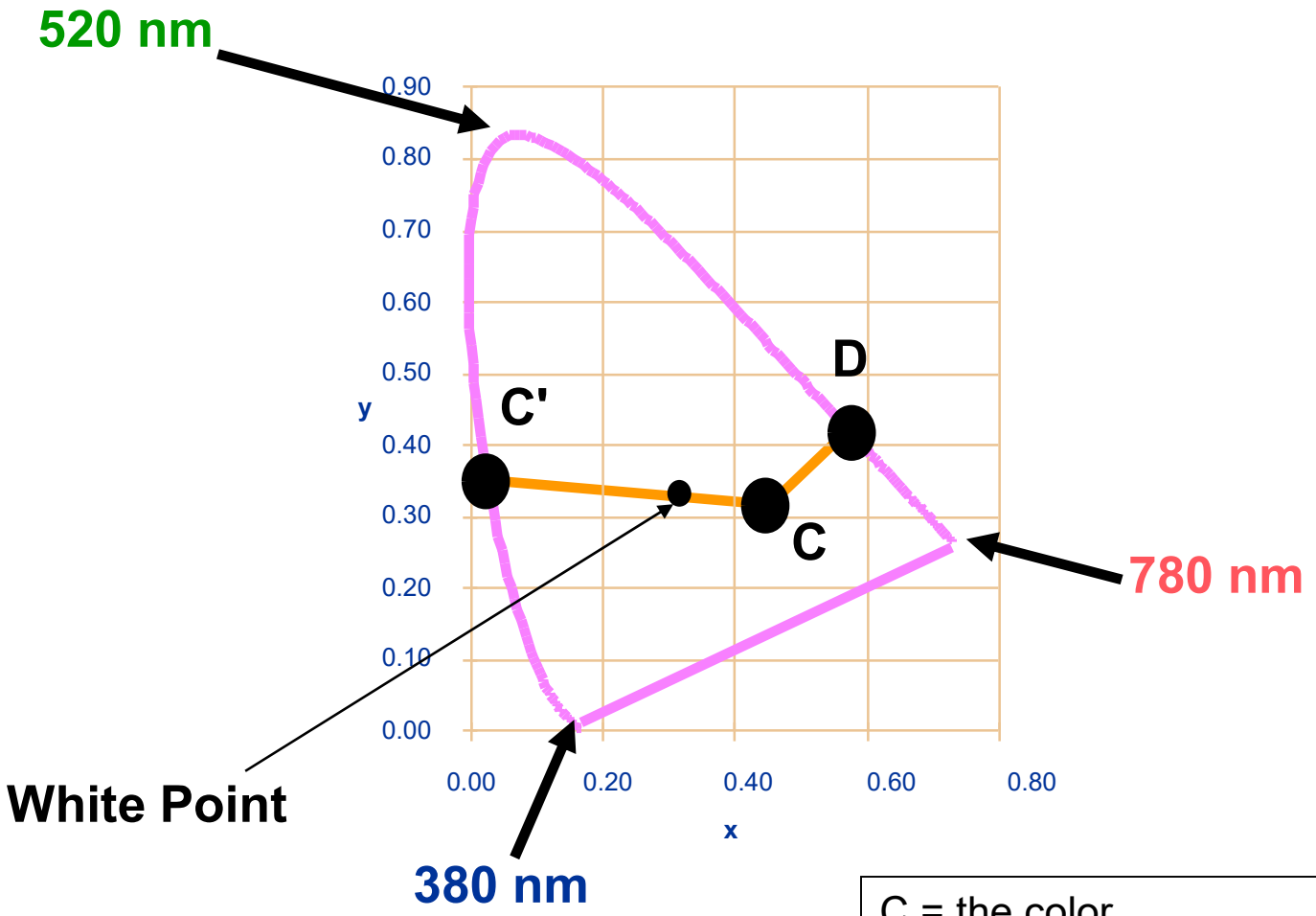


I have one of these in my office! ☺

CIE Chromaticity Diagram

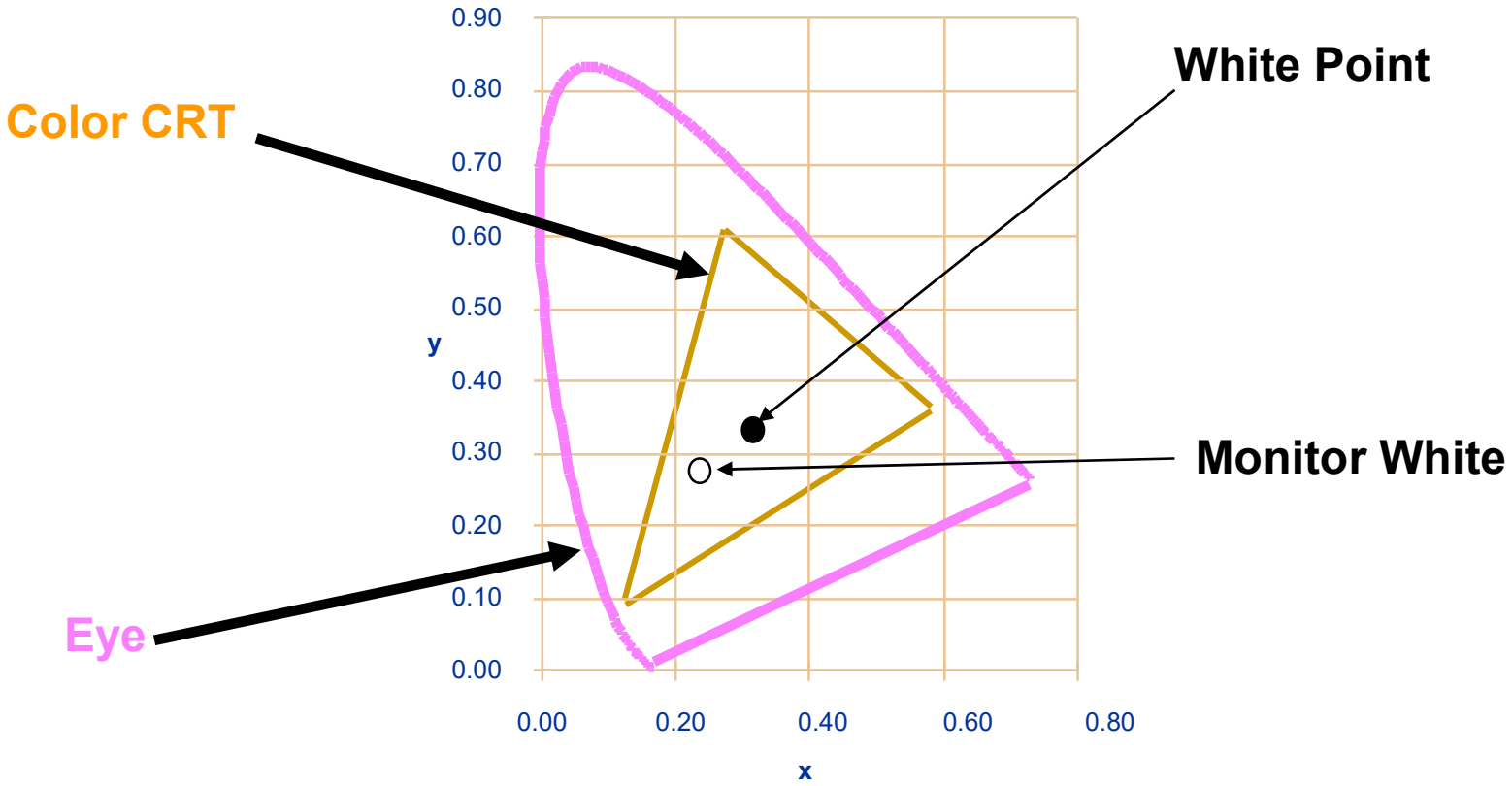


CIE Chromaticity Diagram

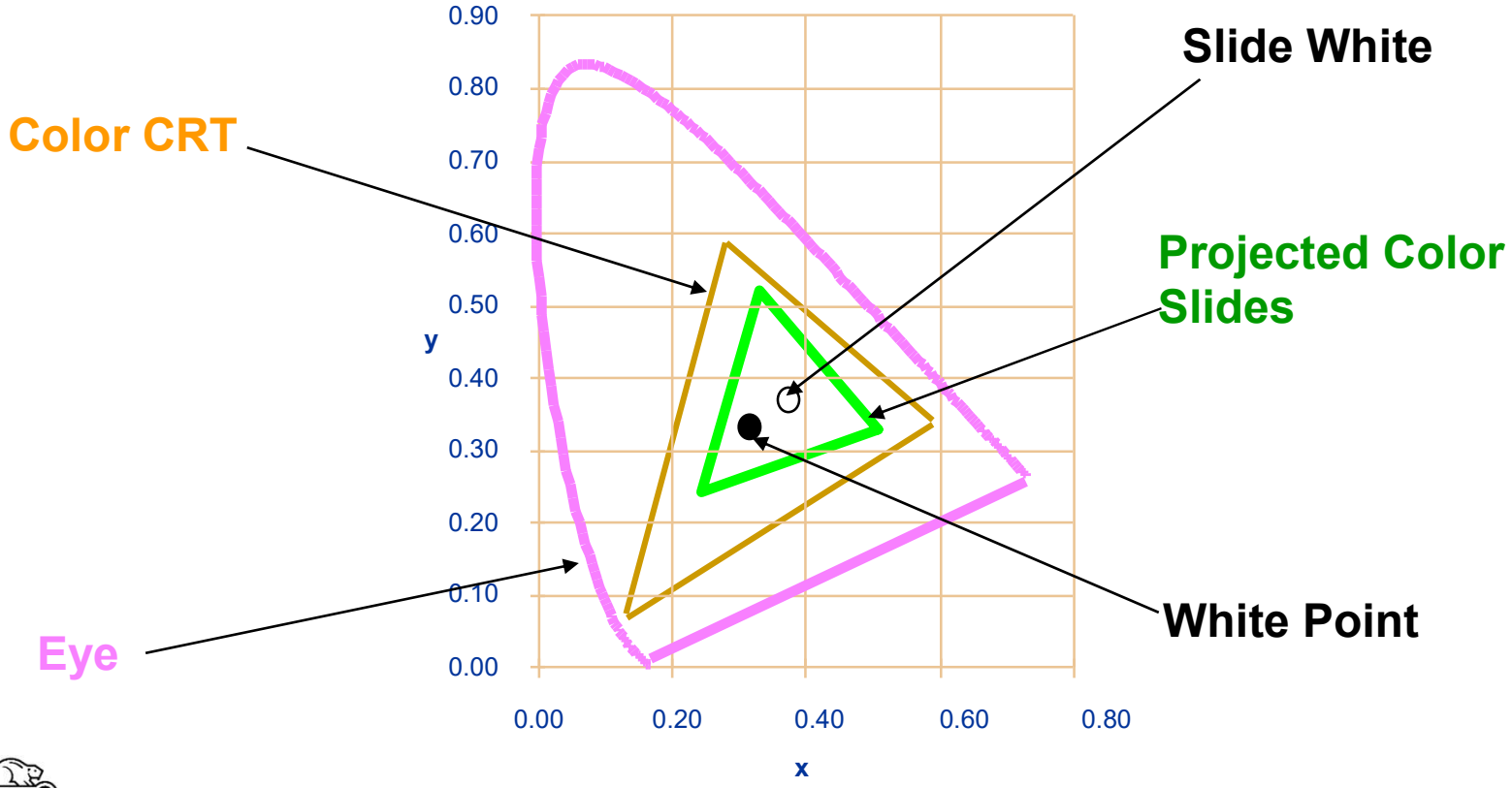


C = the color
D = the dominant wavelength
C ' = the complementary color

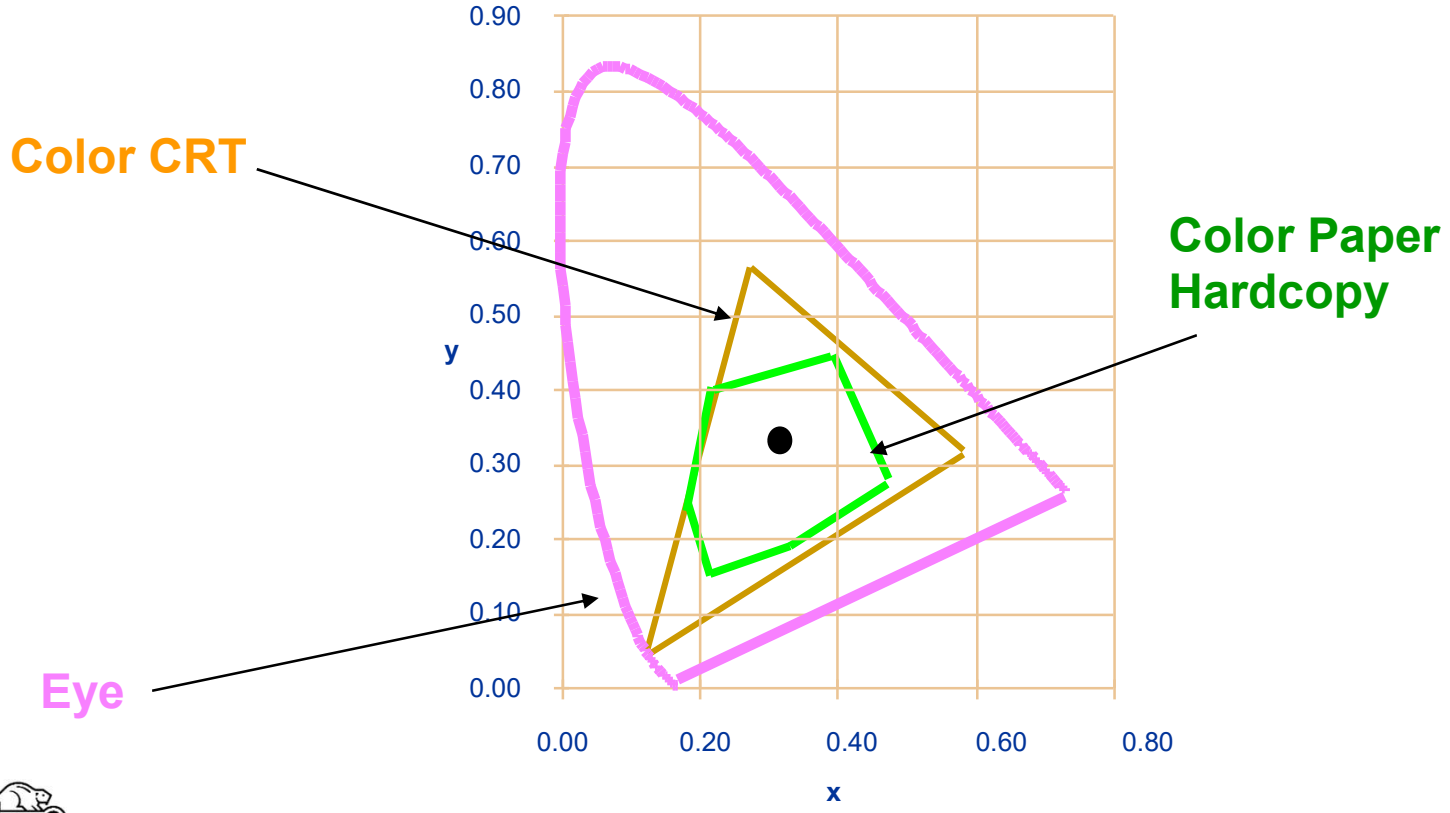
Color Gamut for a Workstation Monitor



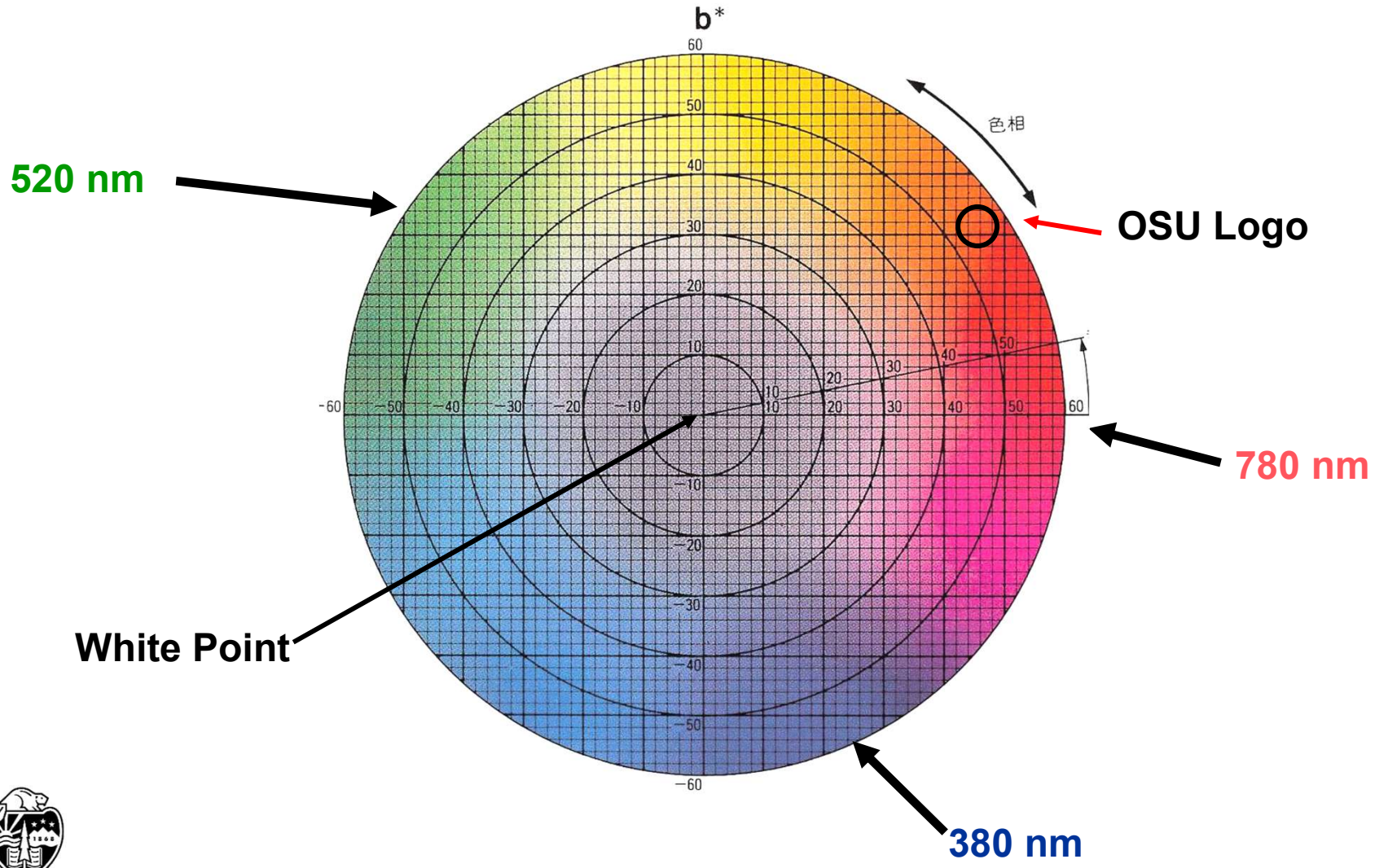
Color Gamut for a Monitor and Color Slides



Color Gamut for a Monitor and Color Printer



The *Perceptually Uniform* L-a-b Color Space

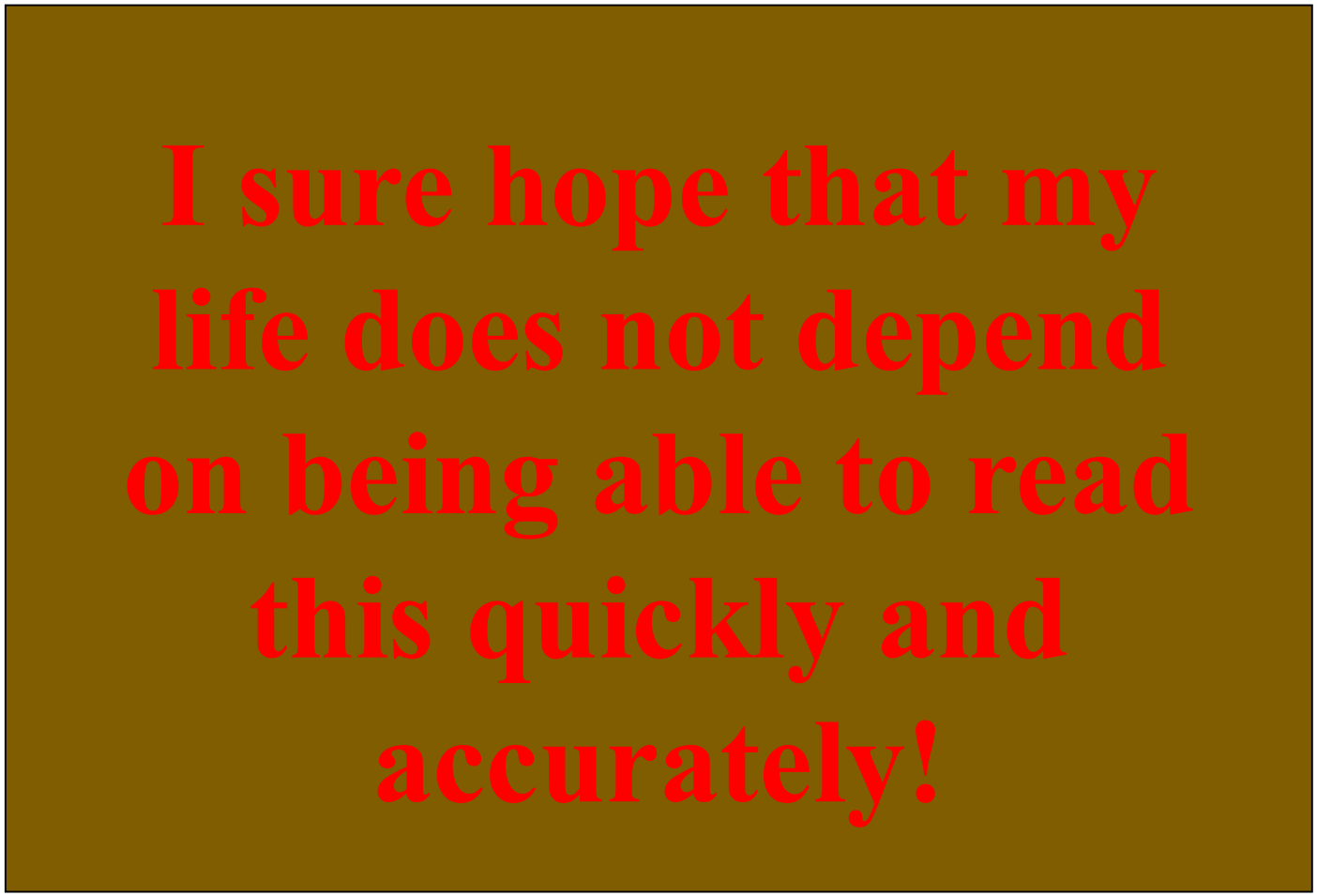


Color Meters Are Able to Measure L-a-b Coordinates

What Makes a Good Contrast?

- Many people think simply adding color onto another color makes a good contrast
- In fact, a better measure is the **Δ Luminance**
- Using this also helps if someone makes a grayscale photocopy of your color hardcopy

Color Alone Doesn't Cut It !



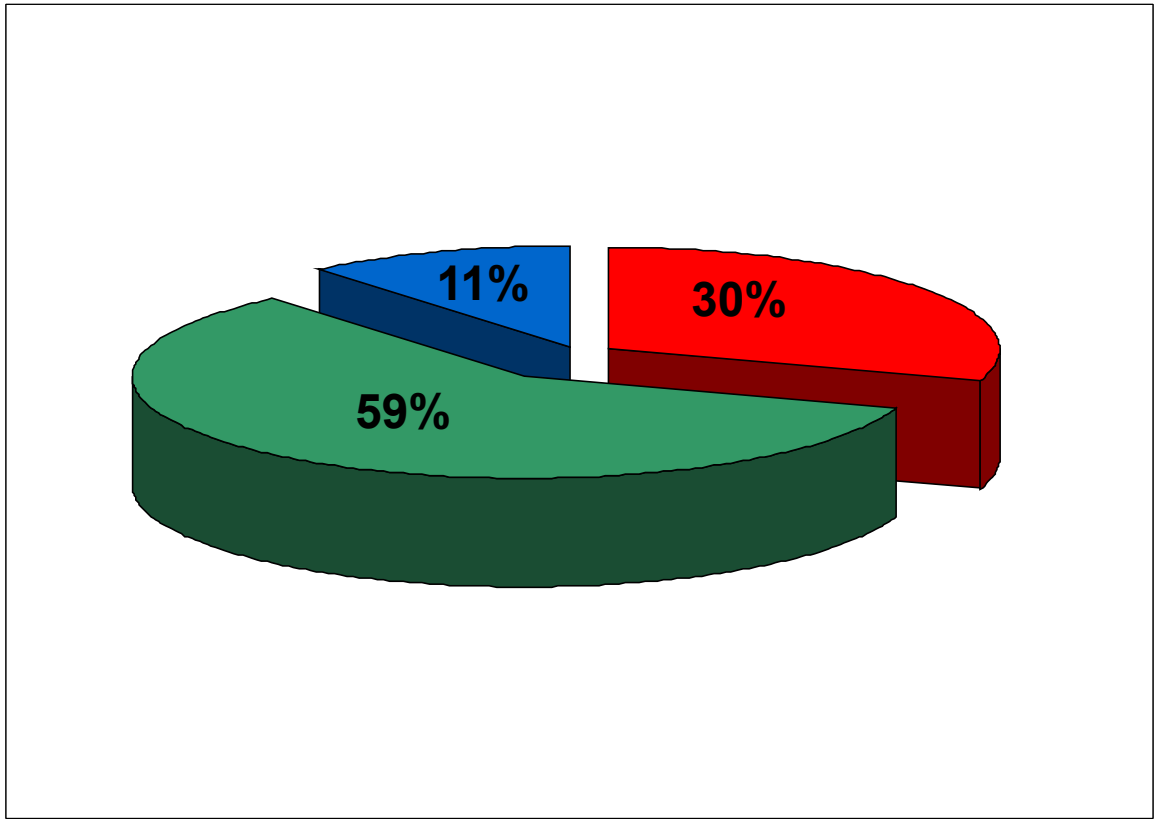
Luminance Contrast is Crucial !

I would prefer that my life depend on being able to read *this* quickly and accurately!


TUESDAY
MARCH 29
3–4 PM
RSVP to:
Or call: 541.737.0664

The Luminance Equation

$$Y = .30*Red + .59*Green + .11*Blue$$



Luminance Table

	R	G	B	Y
Black	0.0	0.0	0.0	0.00
White	1.0	1.0	1.0	1.00
Red	1.0	0.0	0.0	0.30
Green	0.0	1.0	0.0	0.59
Blue	0.0	0.0	1.0	0.11
Cyan	0.0	1.0	1.0	0.70
Magenta	1.0	0.0	1.0	0.41
Orange	1.0	0.5	0.0	0.60
Yellow	1.0	1.0	0.0	0.89

≈ Contrast Table

(I use a ΔL* of about 0.40)

	Black	White	Red	Green	Blue	Cyan	Magenta	Orange	Yellow
Black	0.00	1.00	0.30	0.59	0.11	0.70	0.41	0.60	0.89
White	1.00	0.00	0.70	0.41	0.89	0.30	0.59	0.41	0.11
Red	0.30	0.70	0.00	0.29	0.19	0.40	0.11	0.30	0.59
Green	0.59	0.41	0.29	0.00	0.48	0.11	0.18	0.01	0.30
Blue	0.11	0.89	0.19	0.48	0.00	0.59	0.30	0.49	0.78
Cyan	0.70	0.30	0.40	0.11	0.59	0.00	0.29	0.11	0.19
Magenta	0.41	0.59	0.11	0.18	0.30	0.29	0.00	0.19	0.48
Orange	0.60	0.41	0.30	0.01	0.49	0.11	0.19	0.00	0.30
Yellow	0.89	0.11	0.59	0.30	0.78	0.19	0.48	0.30	0.00


Black
White
Red
Yellow
Green
Blue

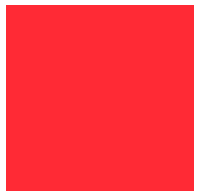
Limit the Total Number of Colors if Viewers are to Discern Information Quickly

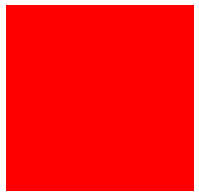
Instructions:

1. Press red to logoff normally

2. Press light red to delete all your files, change your password to something random, and logoff

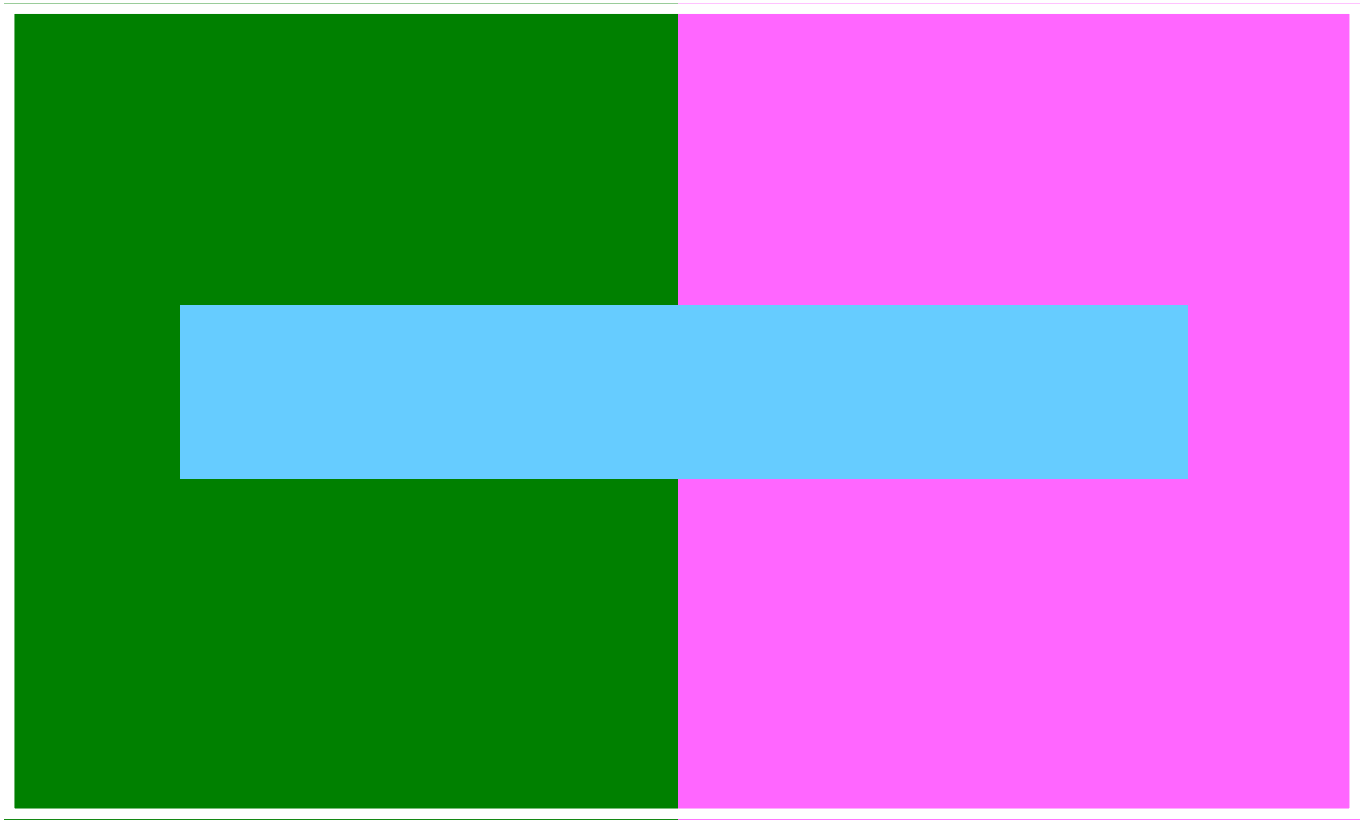
You have 2 seconds • • •



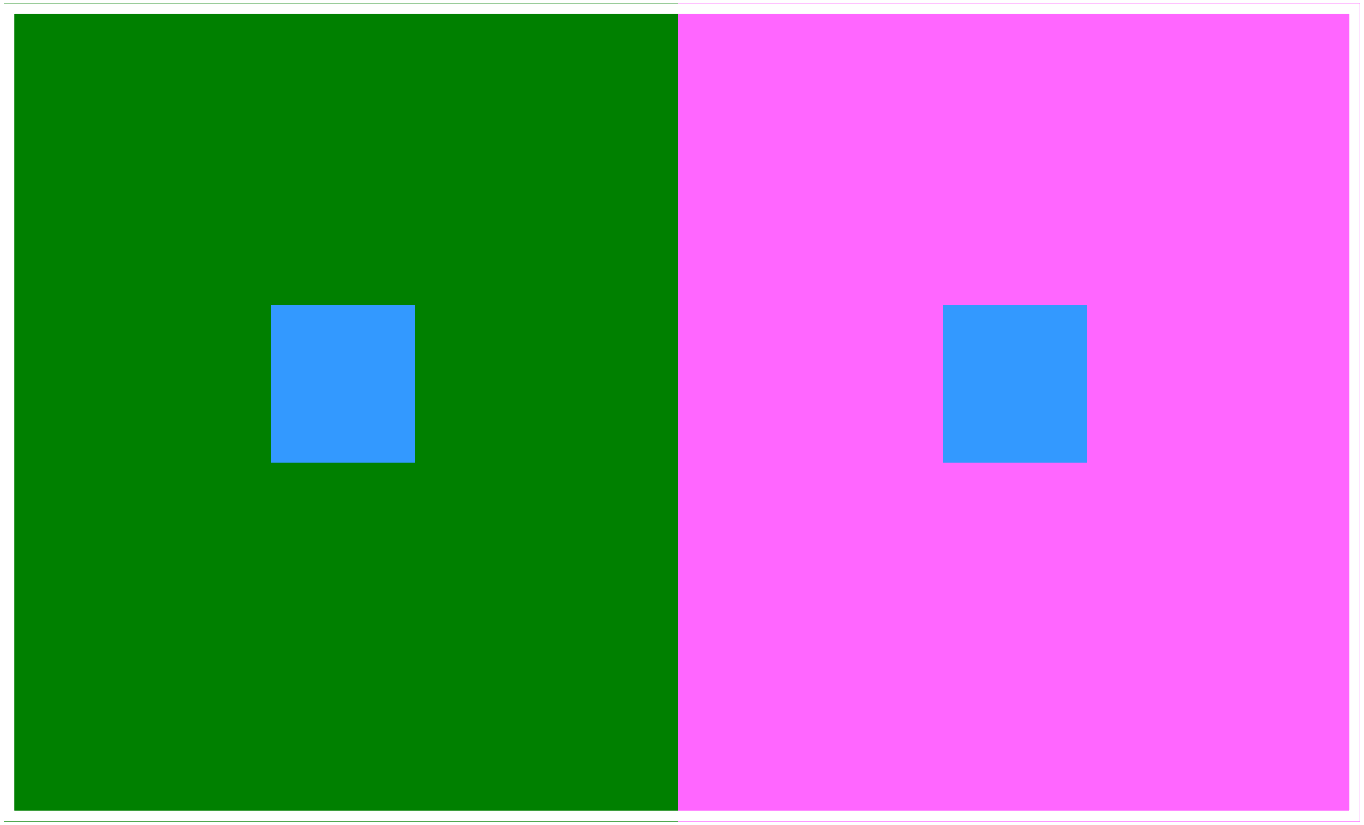


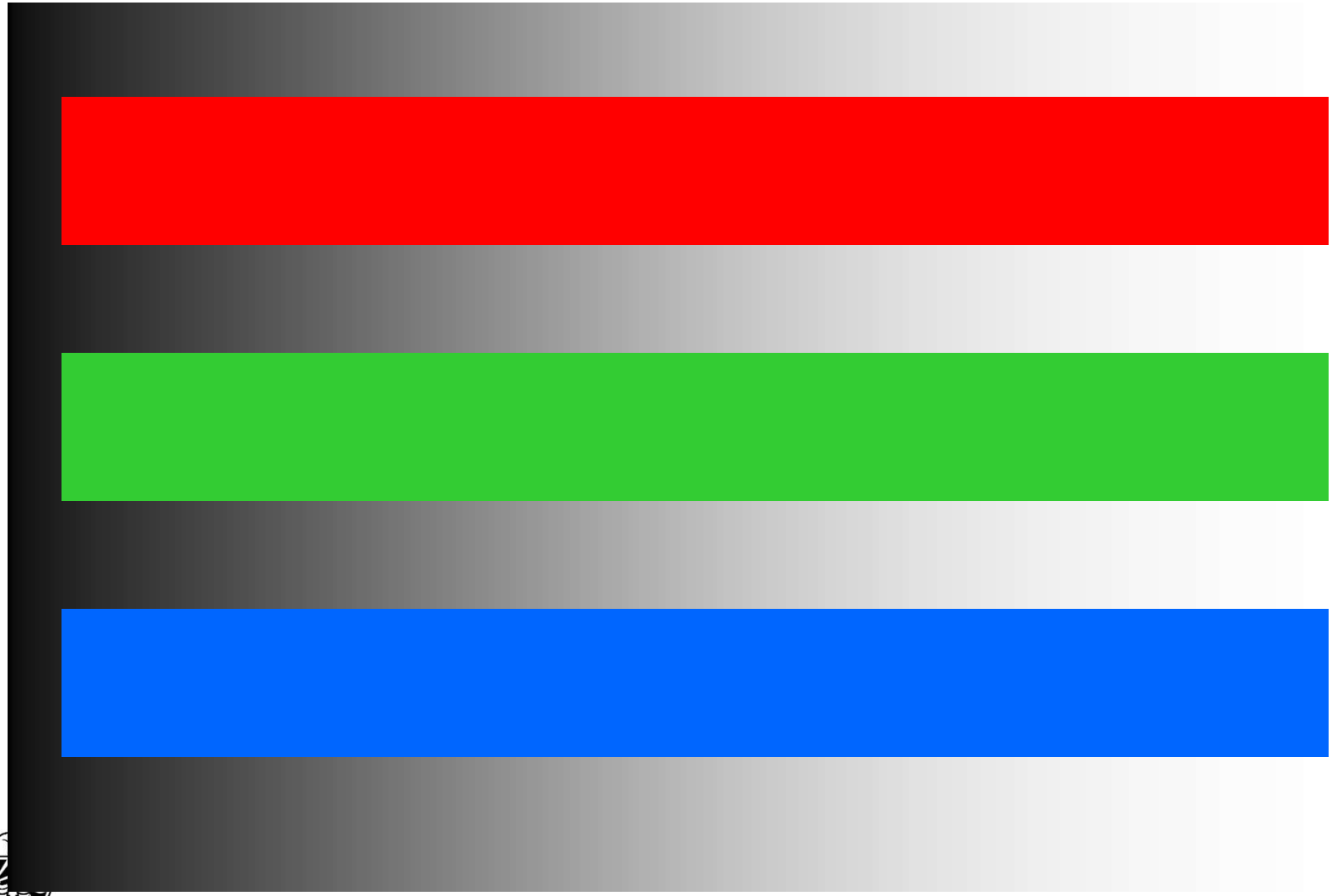
?

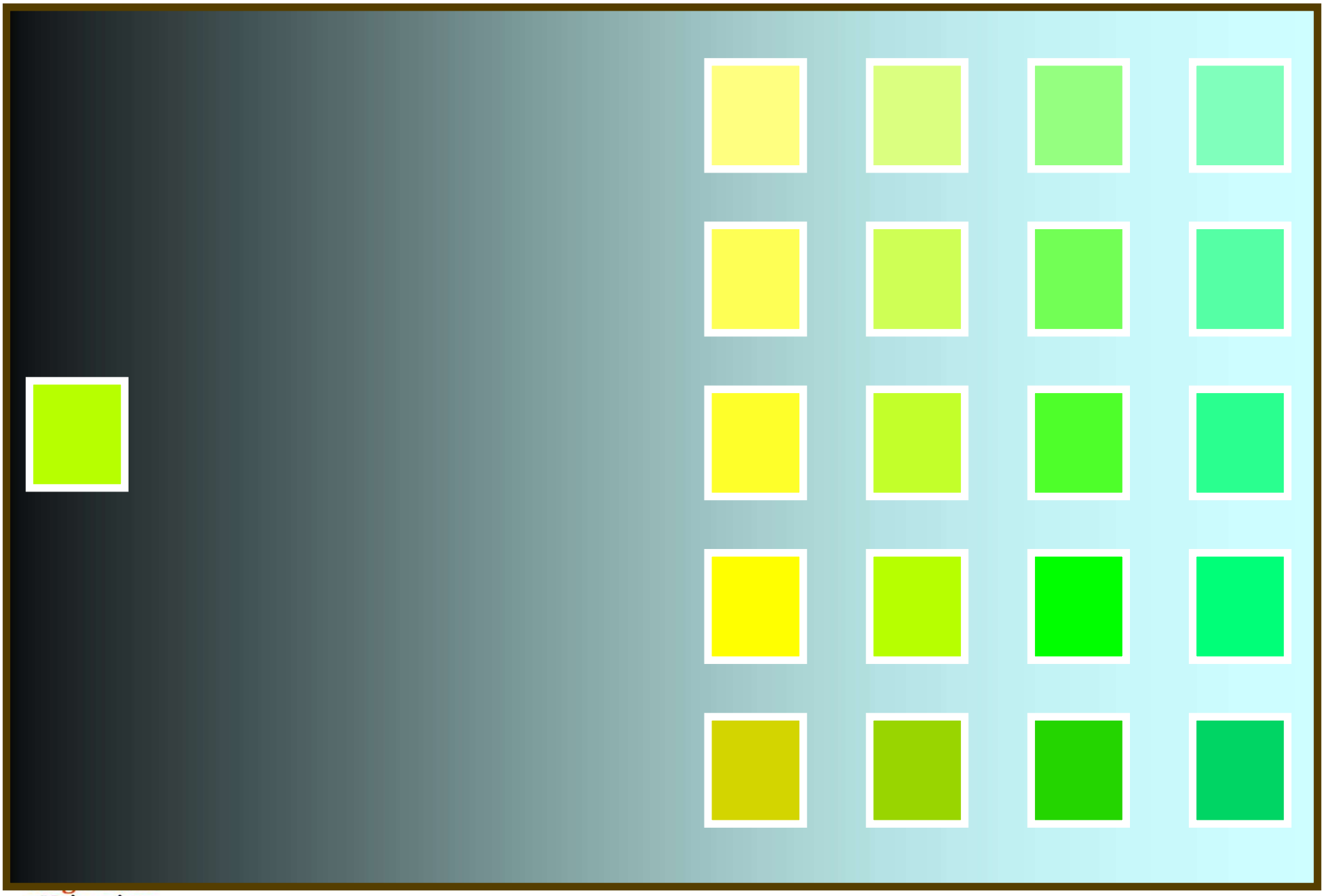
The Ability to Discriminate Colors Changes with Surrounding Color: "Simultaneous Contrast"



The Ability to Discriminate Colors Changes with Surrounding Color: "Simultaneous Contrast"





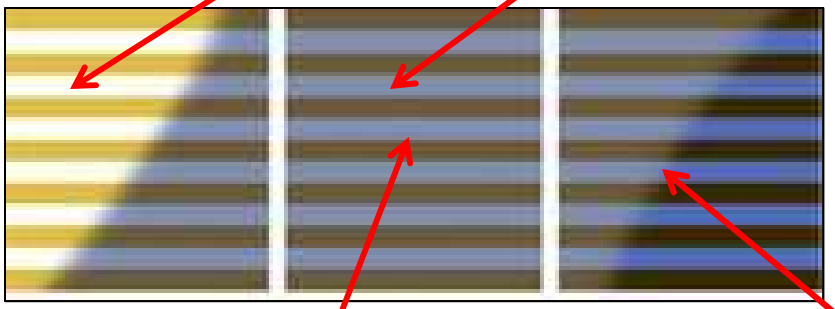


So, What's Up with the "Blue Dress" Debate?

New York Times

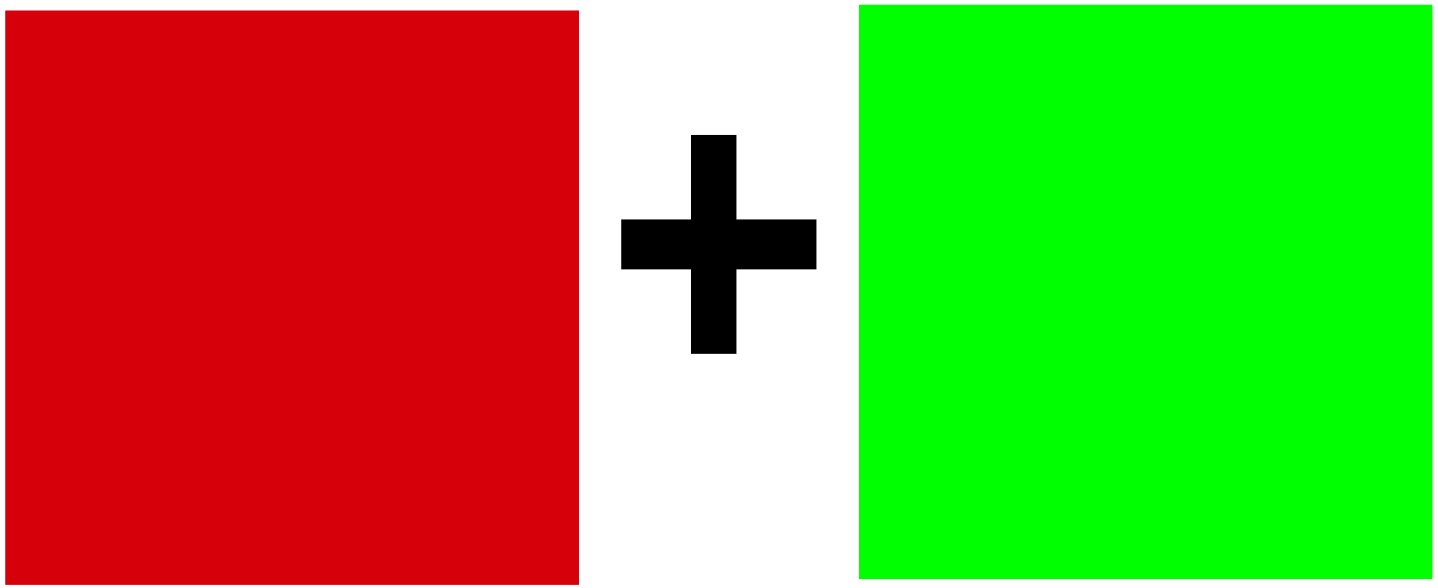
It's all part of the ***Color Constancy*** effect

If you see this color, but you think that the dress is currently in a shadow, you "know" that it must *really* be this color.

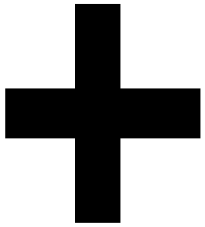


If you see this color, but you think that the dress is currently in bright light, you "know" that it must *really* be this color.

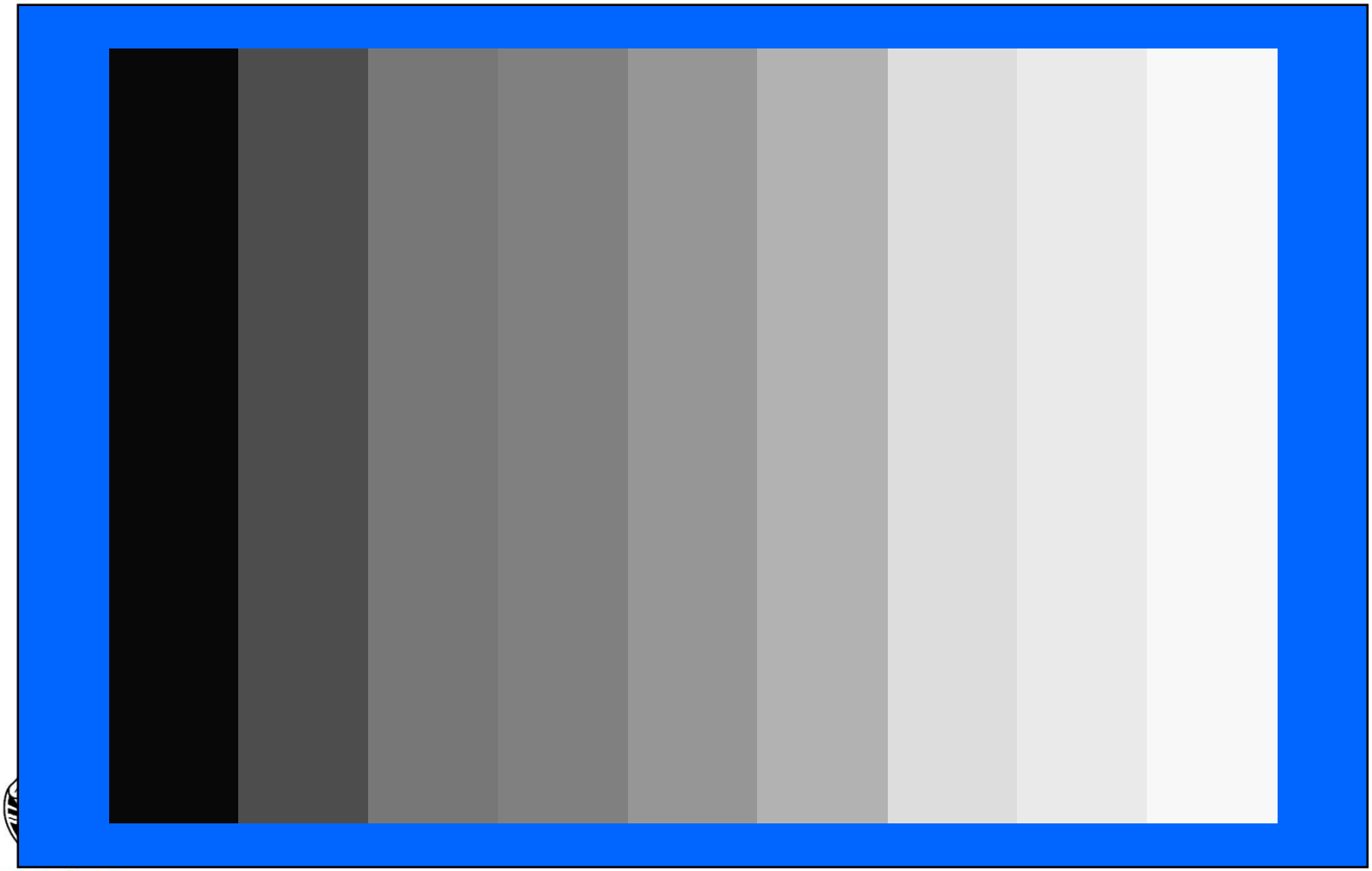
Afterimages



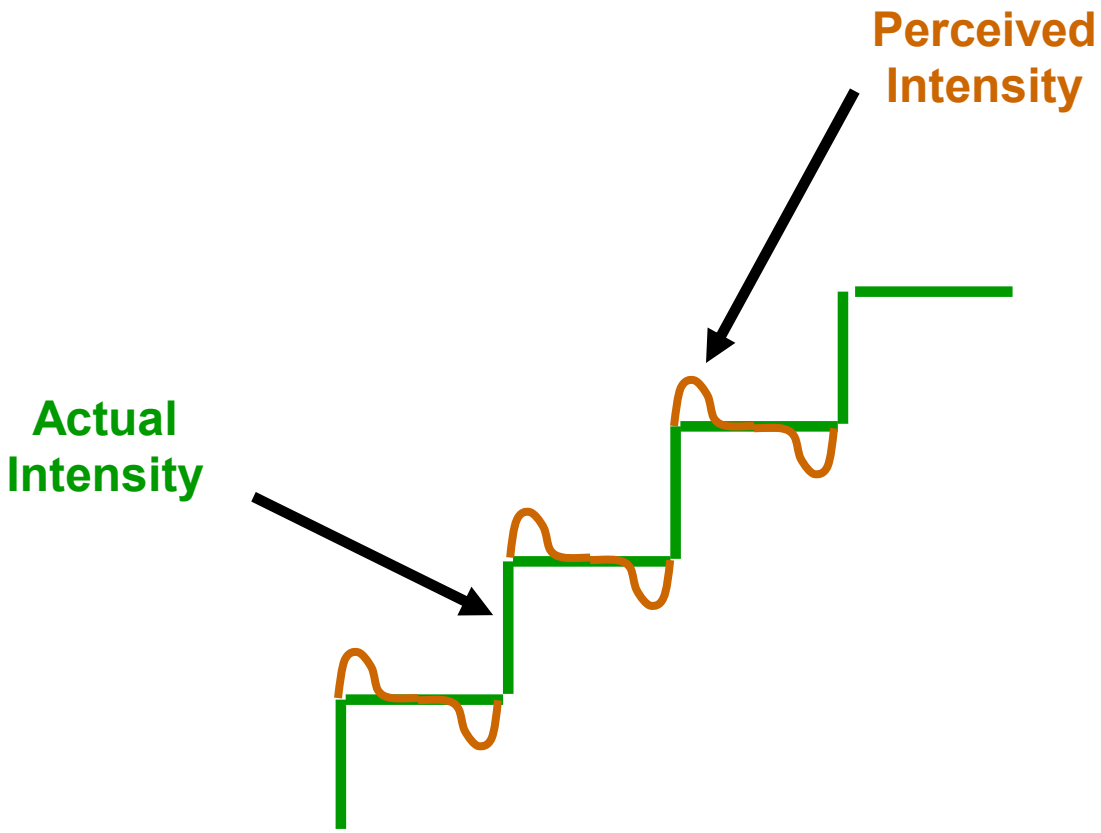
Afterimages



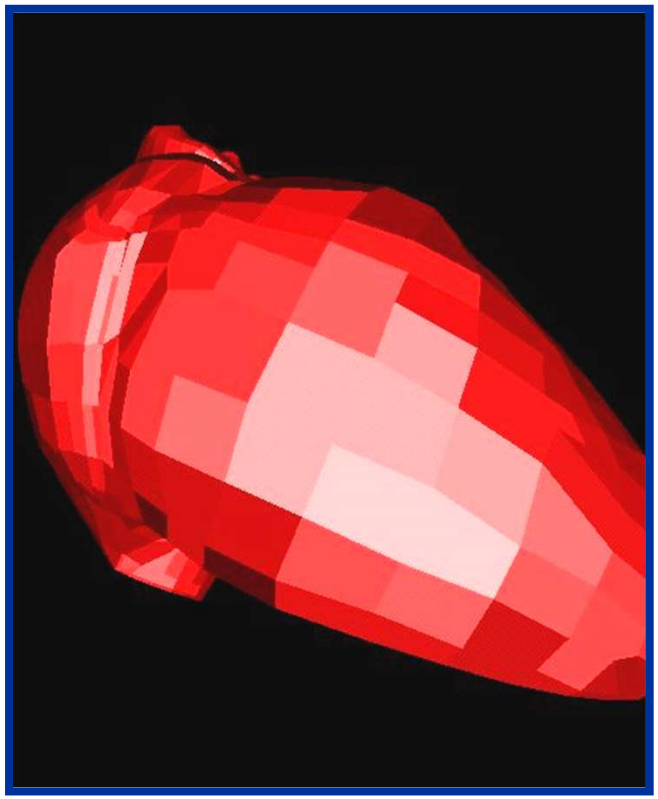
Beware of Mach Banding

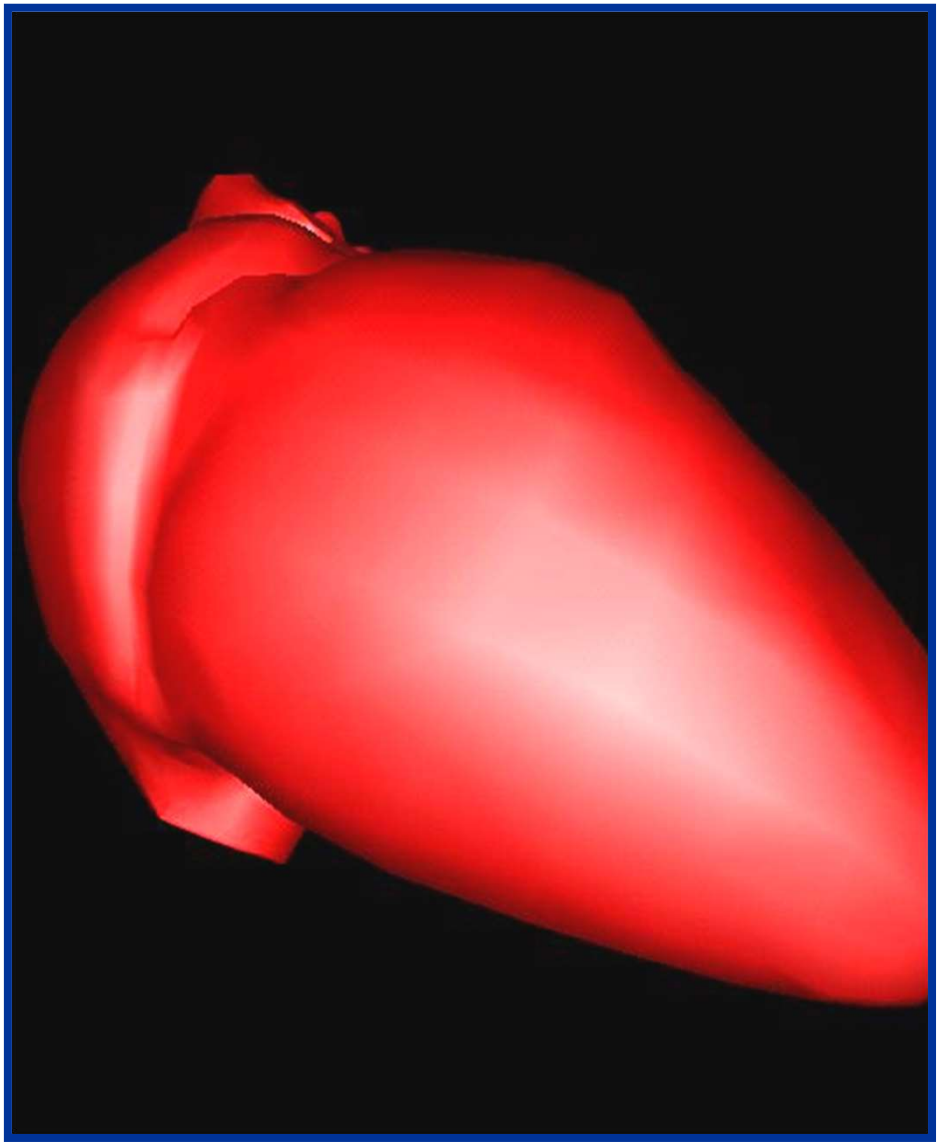


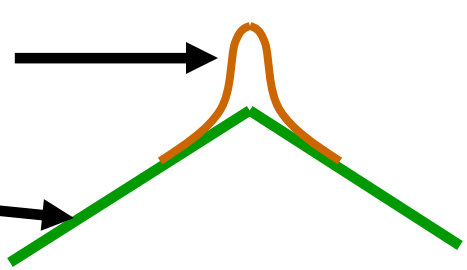
Beware of Mach Banding



Beware of Mach Banding

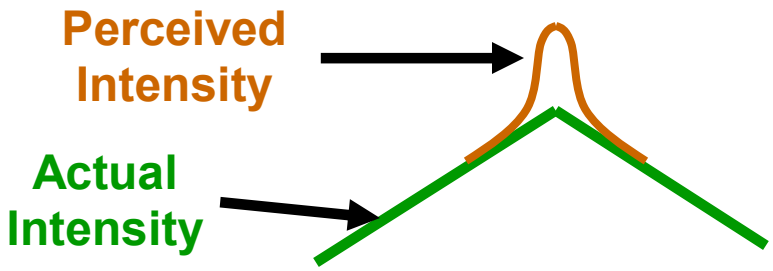






Beware of Mach Banding

Think of the Mach Banding problem as being similar to trying to round second base at a 90º angle.



Be Aware of Color Vision Deficiencies (CVD)

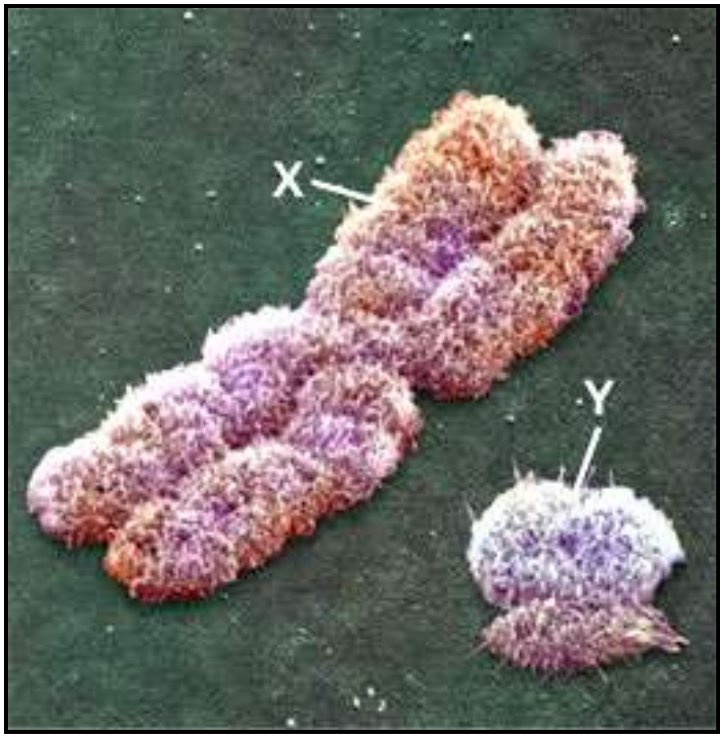
- In general, there is no such thing as total “color blindness”
- CVD affects ~10% of Caucasian men
- CVD affects ~4% of non-Caucasian men
- CVD affects ~0.5% of women
- The most common type of CVD is red-green
- Blue-yellow also exists

Resources for designing color schemes for people with color recognition deficiencies:

http://colorbrewer2.org
http://colororacle.org/usage.html
http://mkweb.bcgsc.ca/colorblind/

Why are more men affected by CVD than women?

It's because the red-green CVD defect is carried on the X Chromosome



http://www.bio.miami.edu/~cmallery/150/mendel/c7.15.X.Y.jpg

An XX with the defective gene on one X chromosome probably has a dominant non-defective gene on the other. An XY with a defective gene on one X chromosome has no other gene to "fix" it.

Be Aware of CVD:
Code Information Redundantly

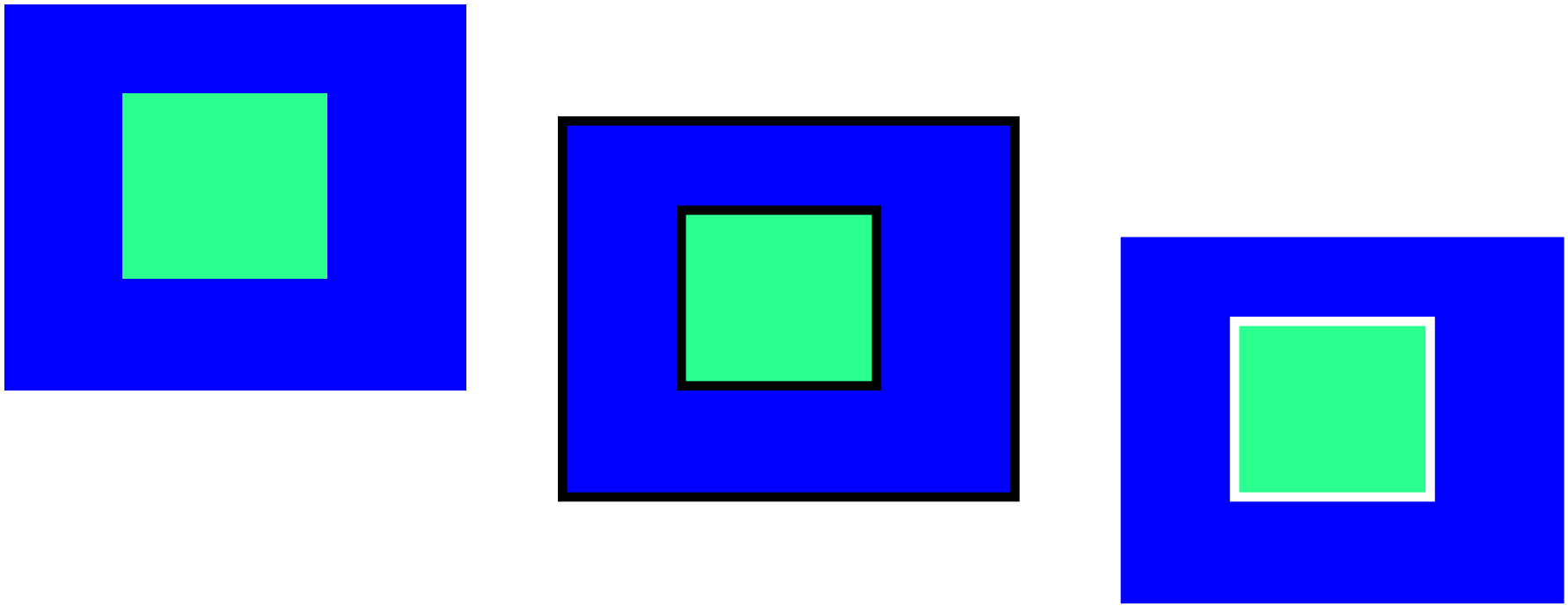
Four score and seven years ago, our fathers brought forth upon this continent a new nation...	Four score and seven years ago, our fathers brought forth upon this continent a new nation...	Four score and seven years ago, our fathers brought forth upon this continent a new nation...

Be Aware of CVD: Code Information Redundantly: Color + …

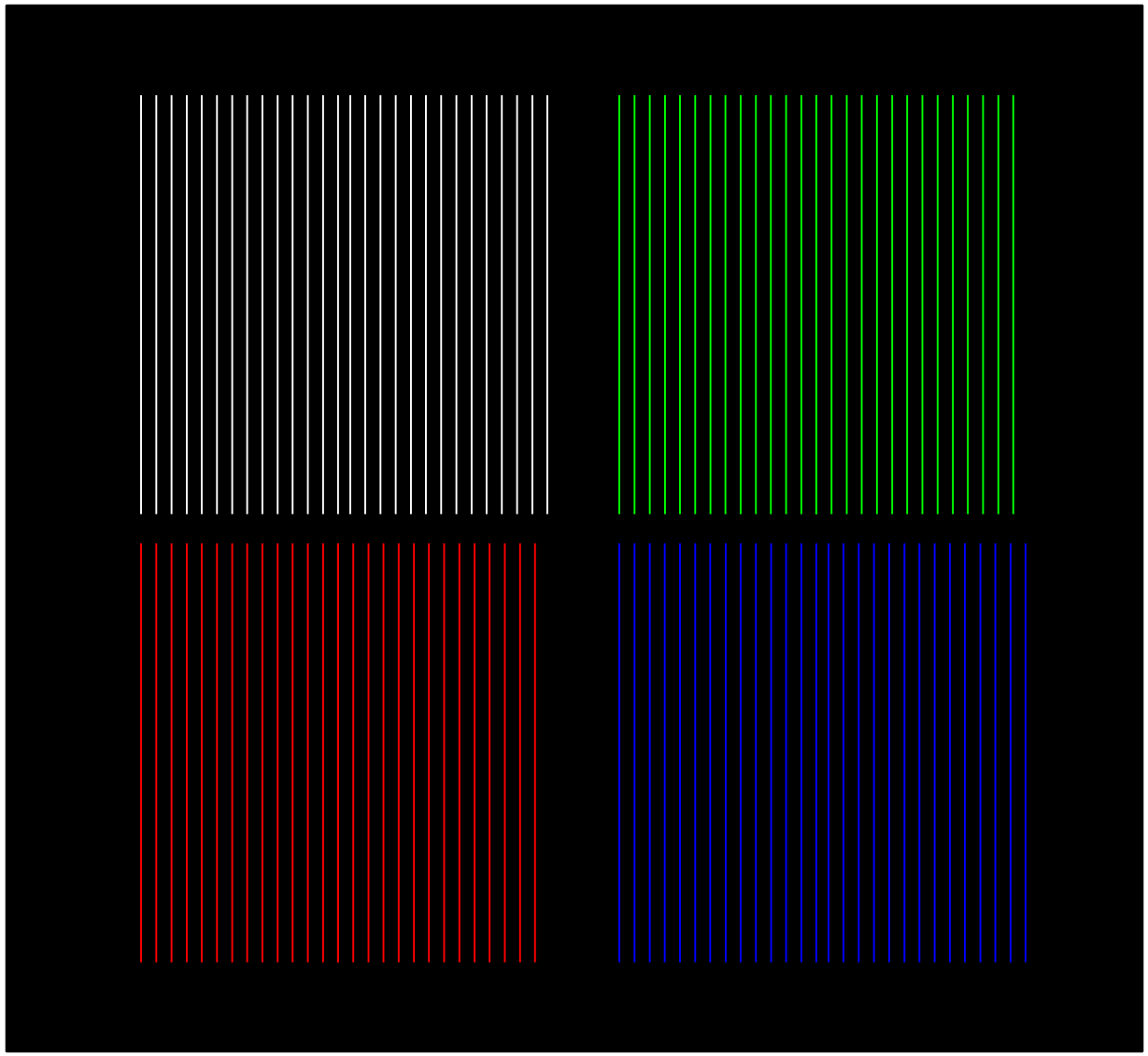
- **Different fonts**
- **Symbols**
- **Fill pattern**
- **Outline pattern**
- **Outline thickness**

This also helps if someone makes a grayscale photocopy of your color hardcopy

Use a Black or White Line as the Boundary Between Colored Regions



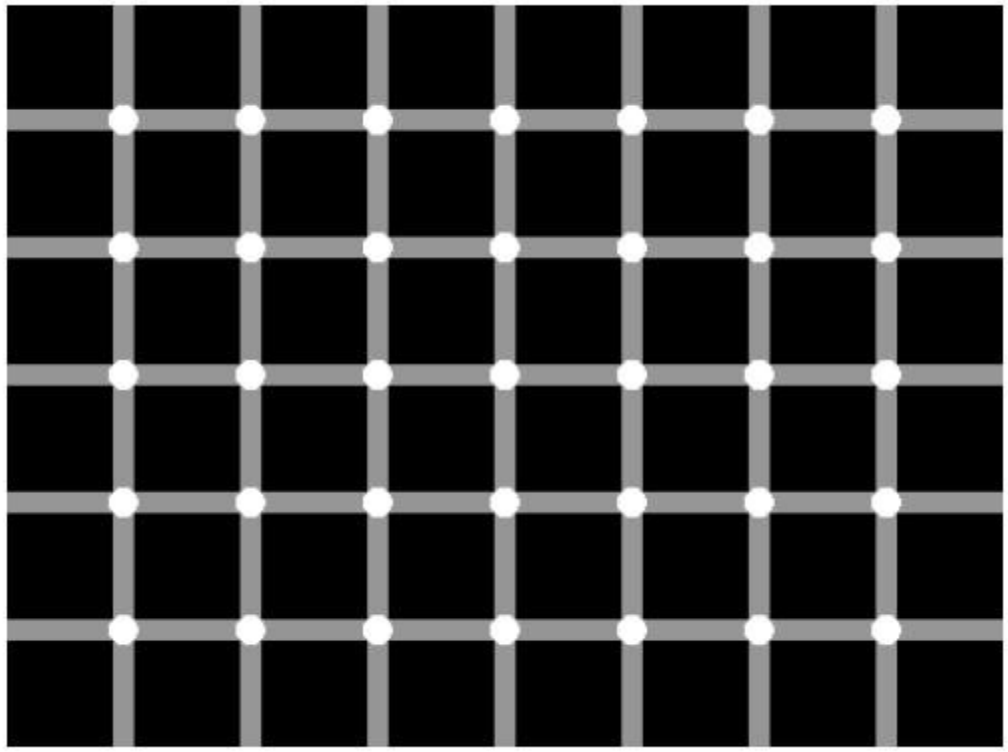
Do Not Display Fast-moving or High-detail Items in Color, Especially Blue

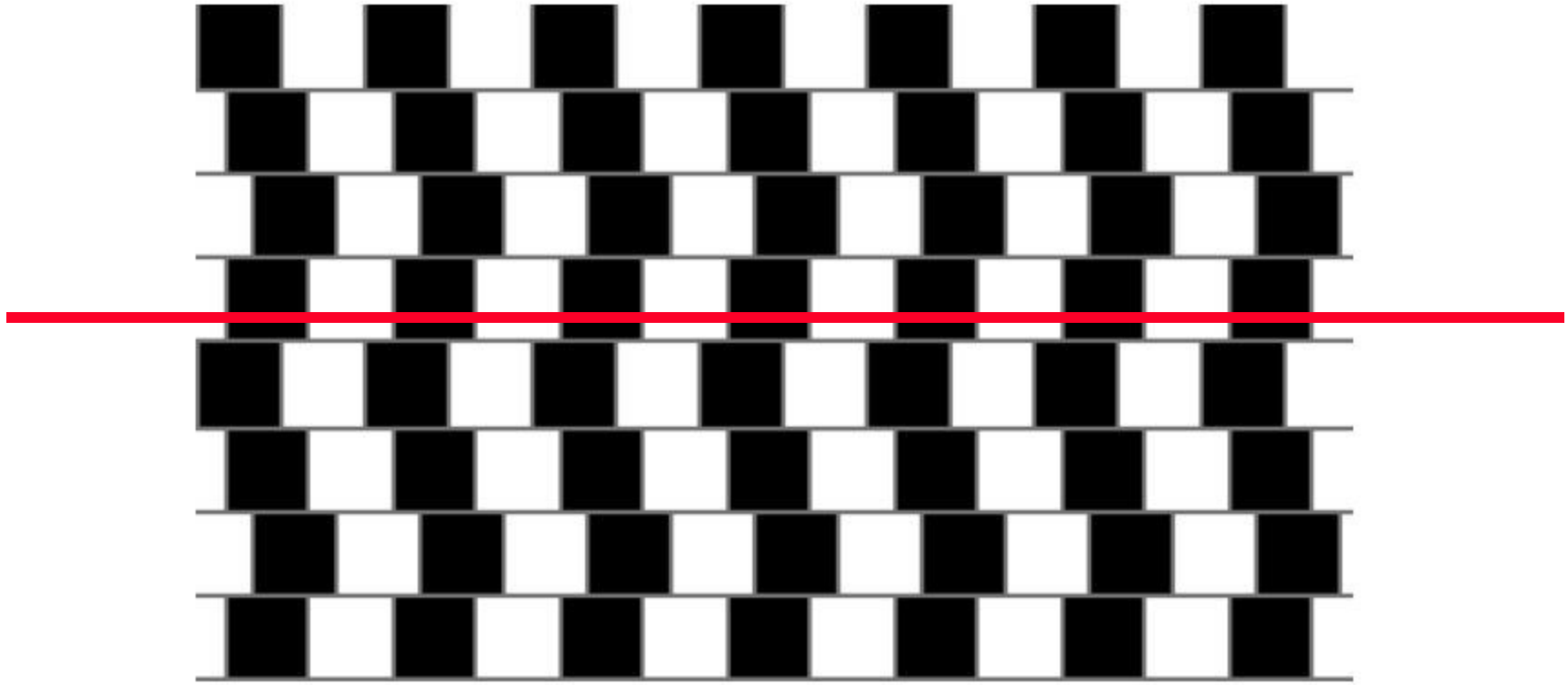


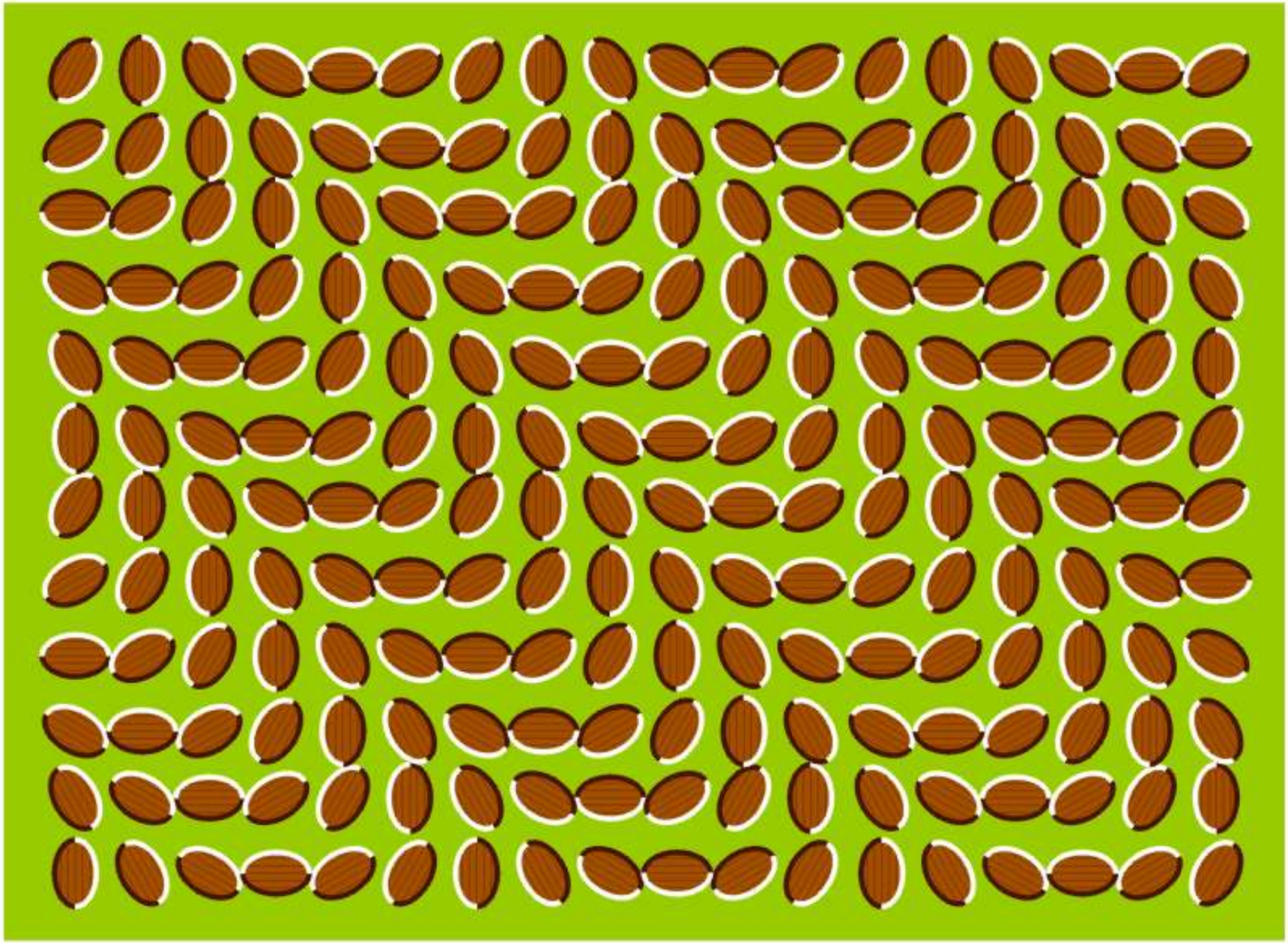
Watch the Use of Saturated Reds and Blues Together

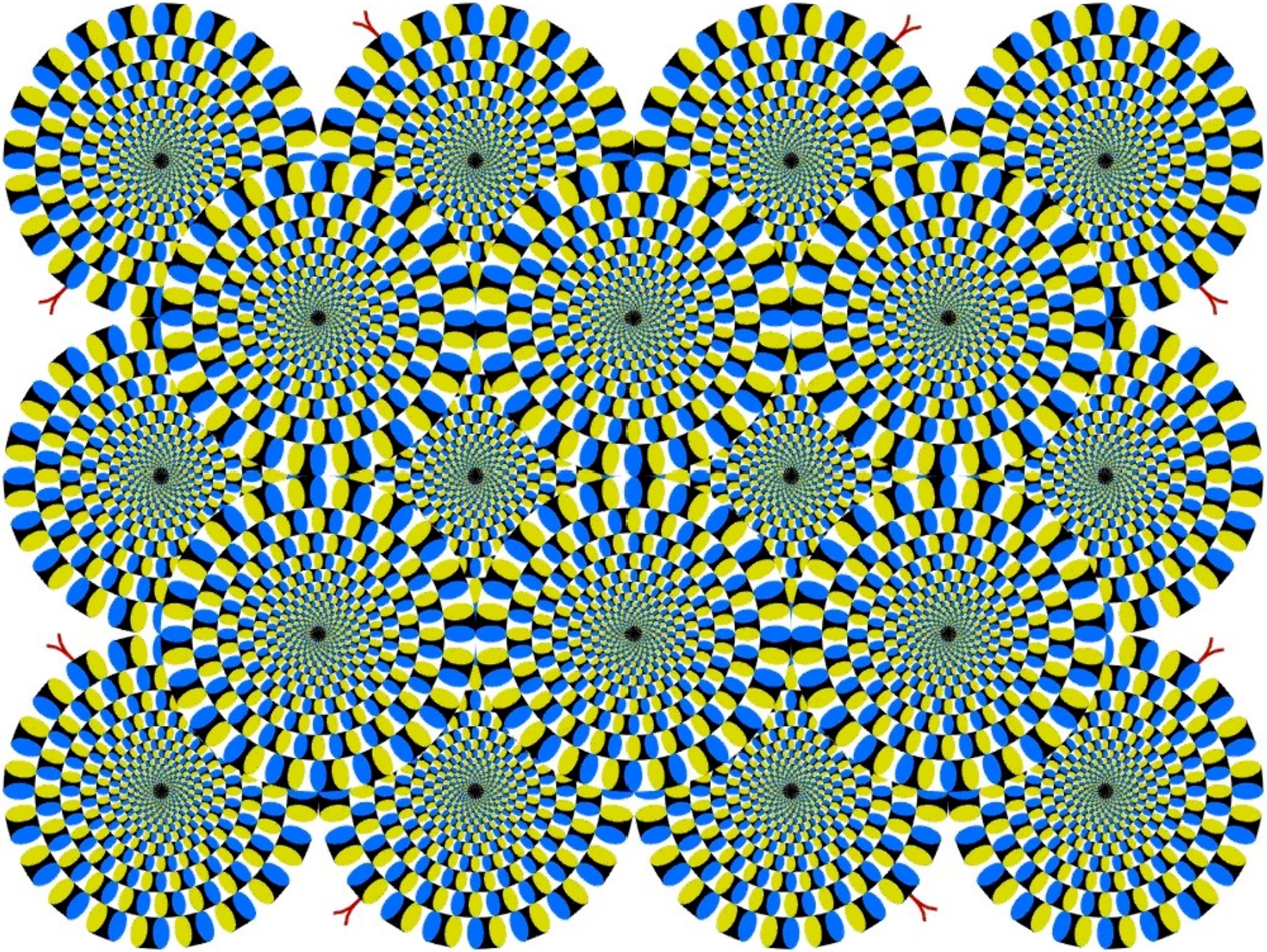
Reds and Blues are on opposite ends of the color spectrum. It is hard for your eyes to focus on both.

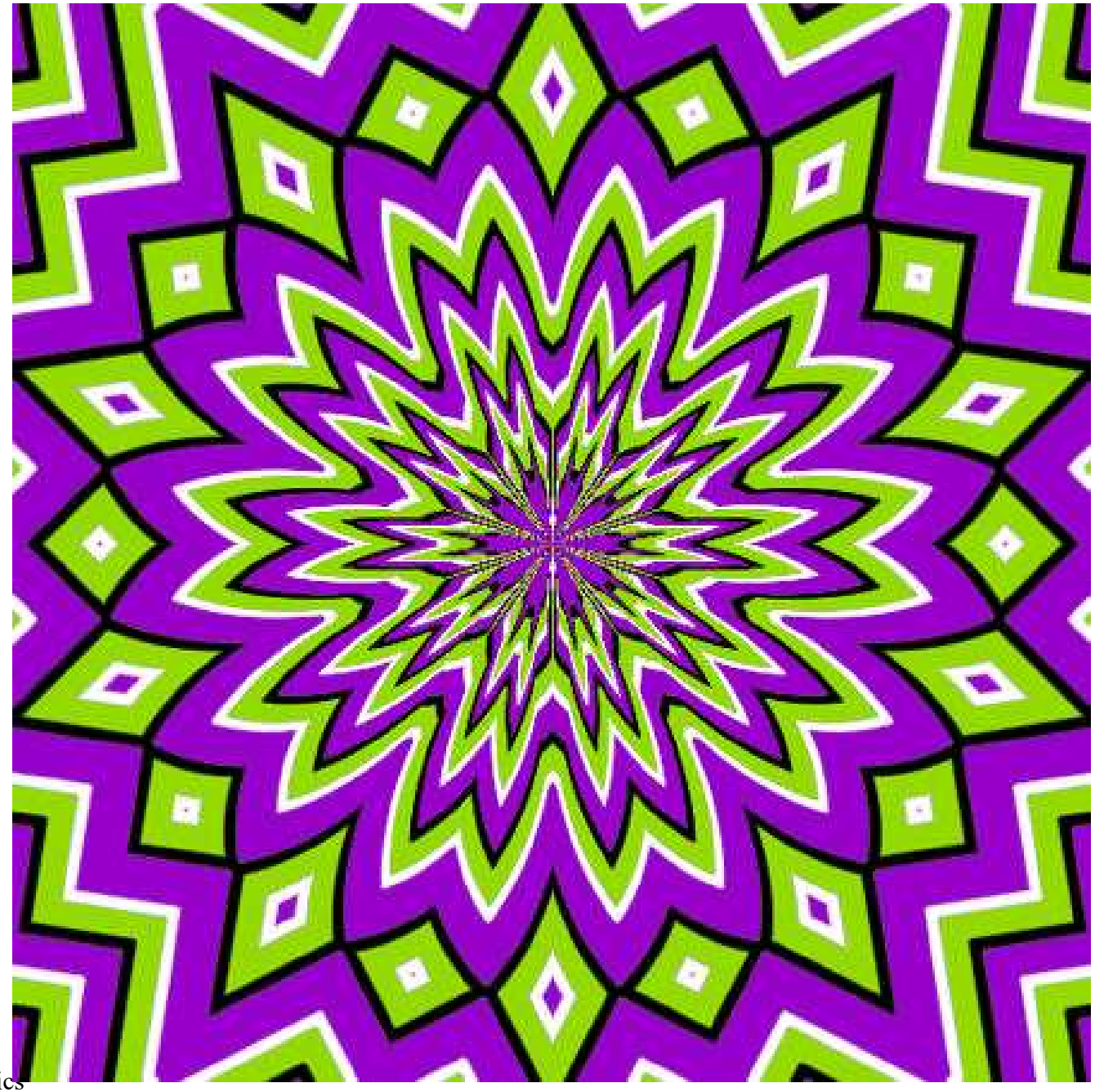
Beware of Lots of Other Stuff

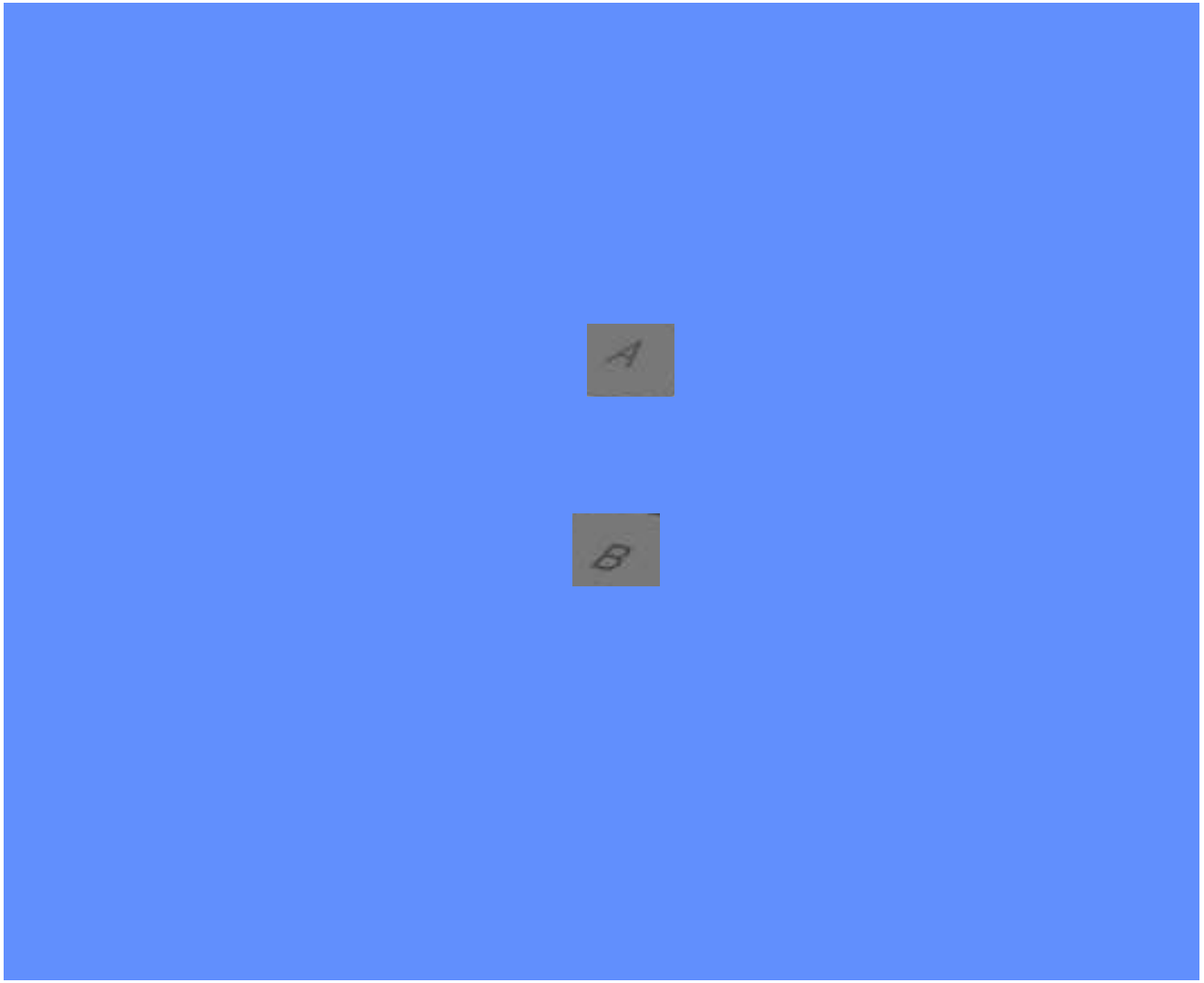


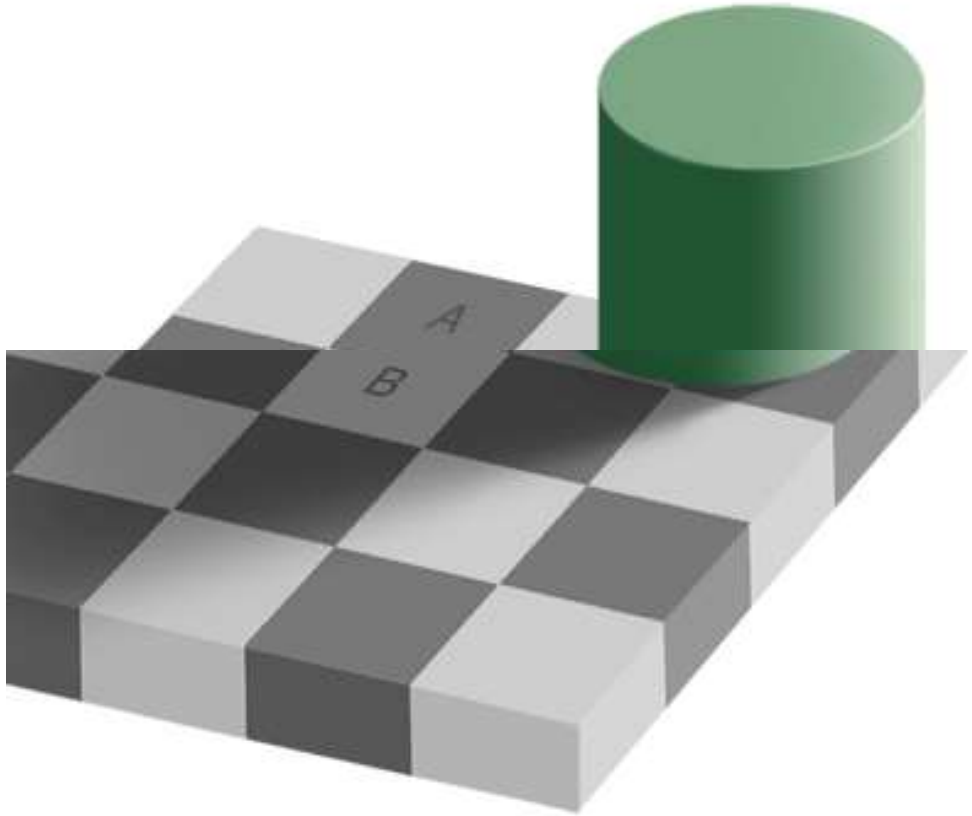









A
B


A
B

Good Color and Perception References

- Theresa-Marie Rhyne, *Applying Color Theory to Digital Media and Visualization*, Second Edition, CRC Press, 2025.
- Maureen Stone, *A Field Guide to Digital Color*, AK Peters, 2003.
- Roy Hall, *Illumination and Color in Computer Generated Imagery*, Springer-Verlag, 1989.
- R. Daniel Overheim and David Wagner, *Light and Color*, John Wiley & Sons, 1982.
- David Travis, *Effective Color Displays*, Academic Press, 1991.
- L.G. Thorell and W.J. Smith, *Using Computer Color Effectively*, Prentice Hall, 1990.
- Edward Tufte, *The Visual Display of Quantitative Information*, Graphics Press, 1983.
- Edward Tufte, *Envisioning Information*, Graphics Press, 1990.
- Edward Tufte, *Visual Explanations*, Graphics Press, 1997.
- Howard Resnikoff, *The Illusion of Reality*, Springer-Verlag, 1989.

Sines and Cosines for Animating Computer Graphics

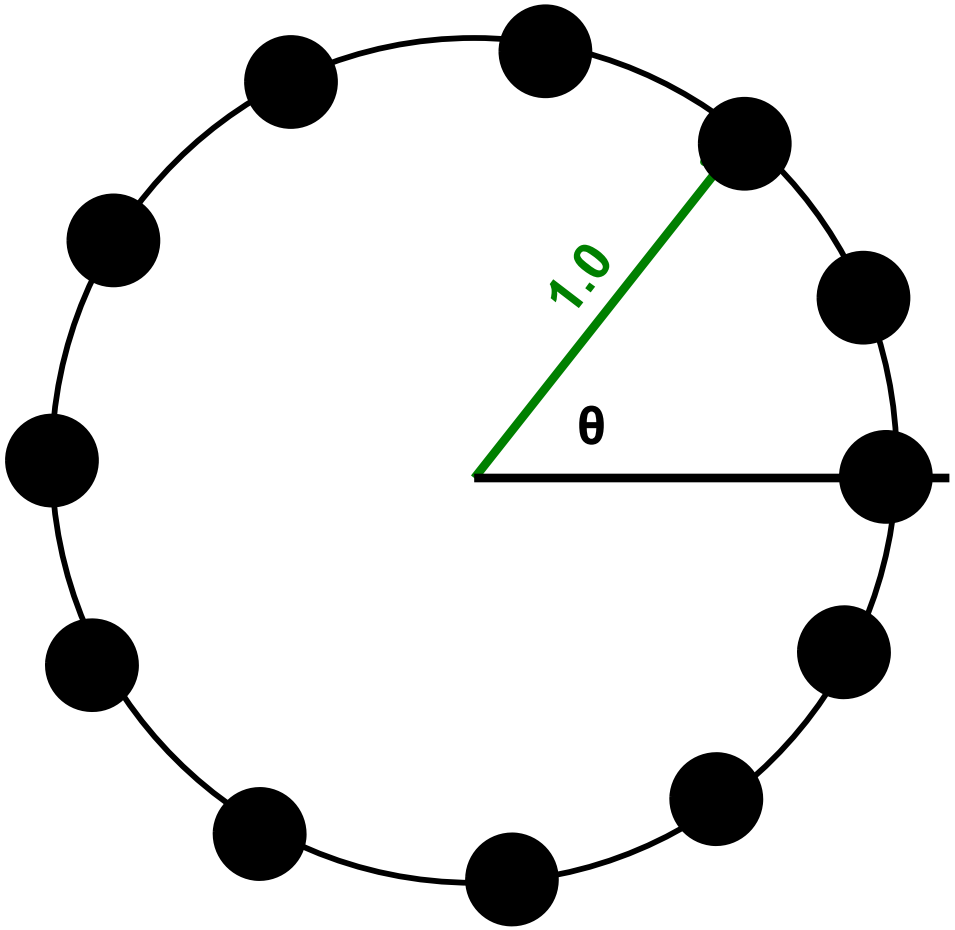
Oregon State University

Mike Bailey

mjb@cs.oregonstate.edu

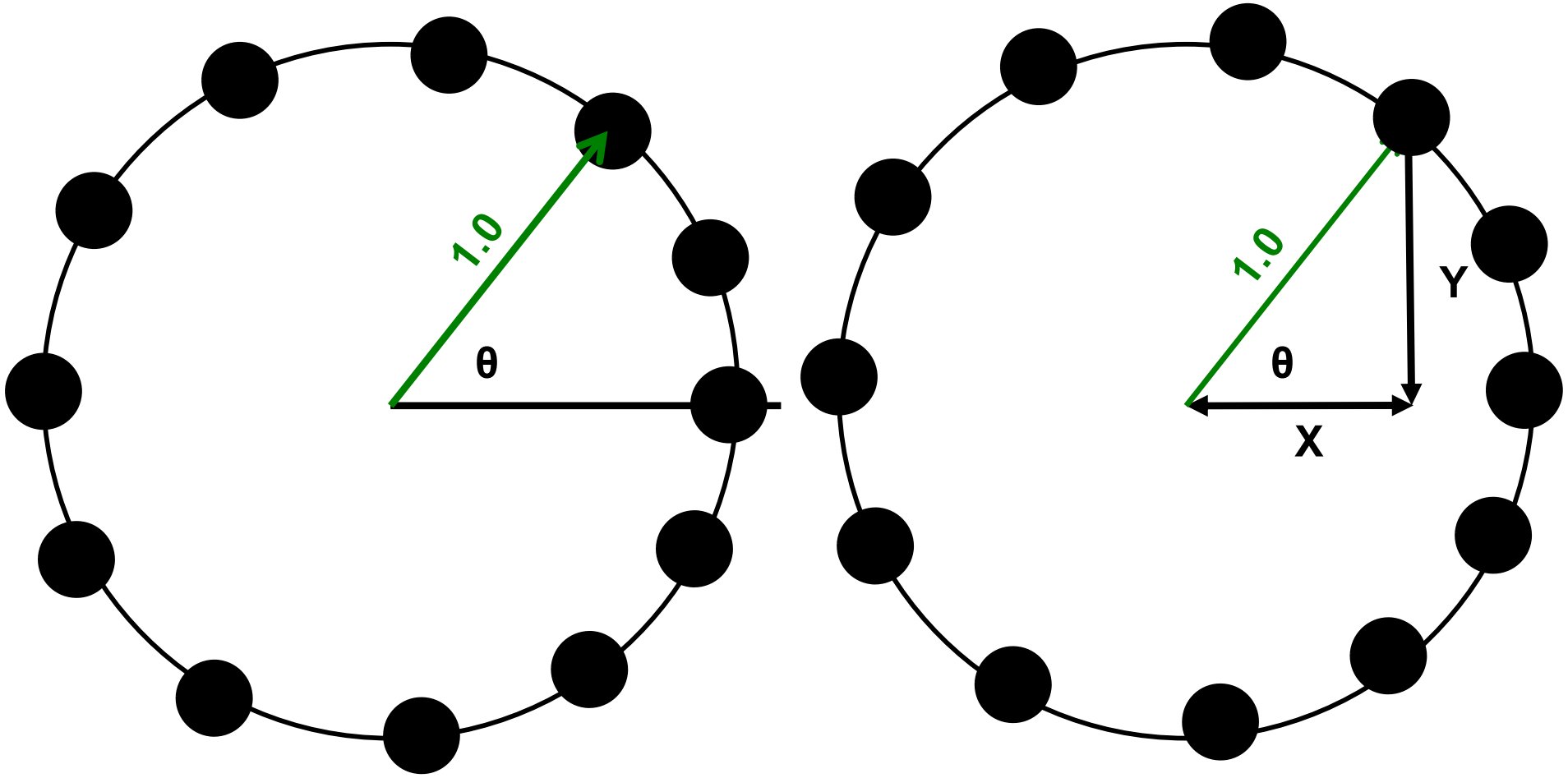
You Know about Sines and Cosines from Math, but They are Very Useful for Animating Computer Graphics

First, We Need to Understand Something about Angles:



If a circle has a radius of 1.0, then we can march around it by simply changing the angle that we call **θ.**

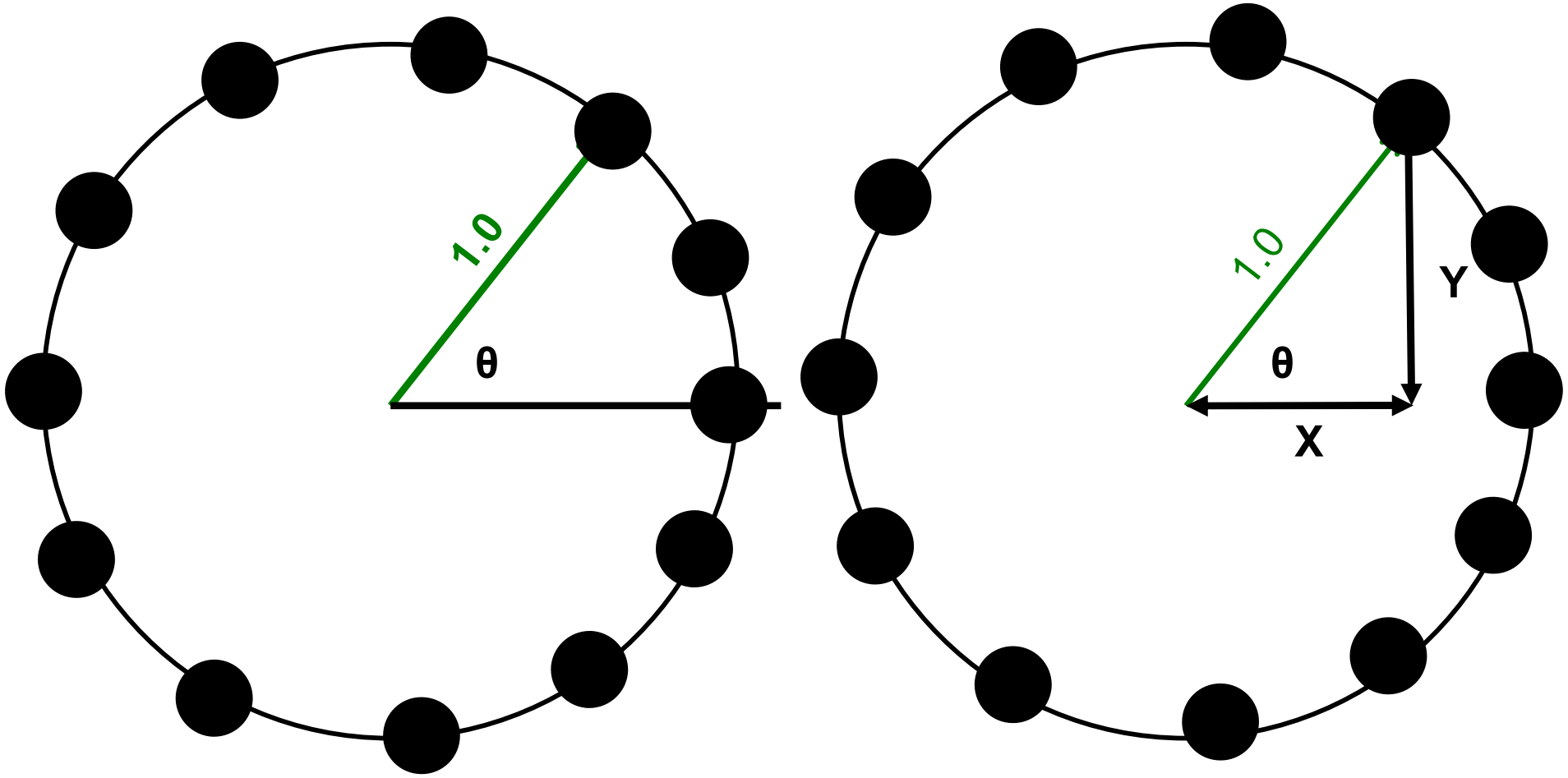
First, We Need to Understand Something about Angles



One of the things we notice is that each angle **θ** has a unique **X** and **Y** that goes with it.

These are different for each **θ**.

First, We Need to Understand Something about Angles



Fortunately, centuries ago, people developed tables of those X and Y values as functions of θ.

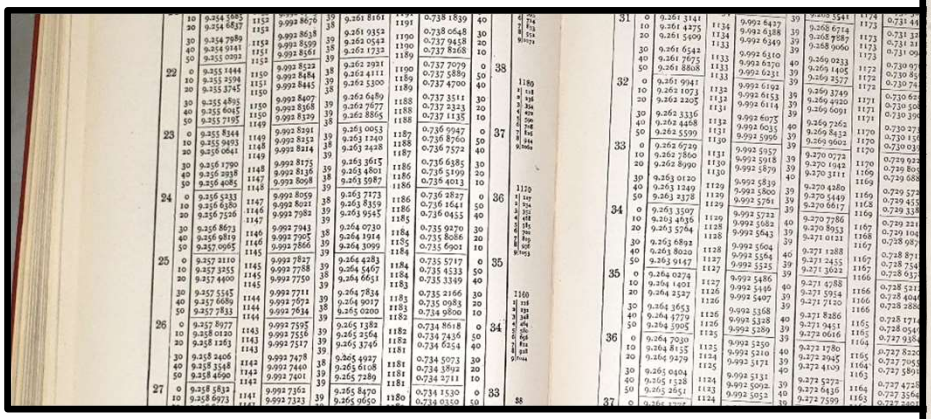
They called the X values **cosines** and the Y values **sines**. These are abbreviated cos and sin.

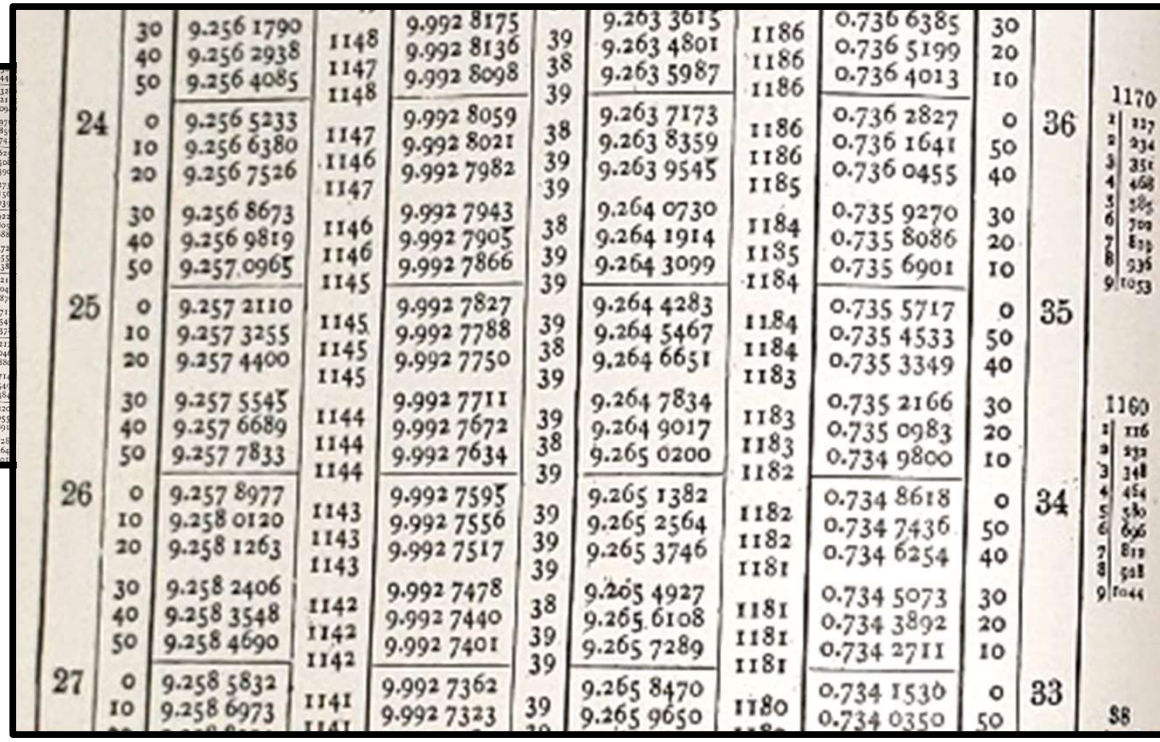
$cos\ \theta$=X

$sin\ \theta$=Y

How People used to Lookup Sines and Cosines – Yuch! Fortunately We Now Have Calculators and Computers

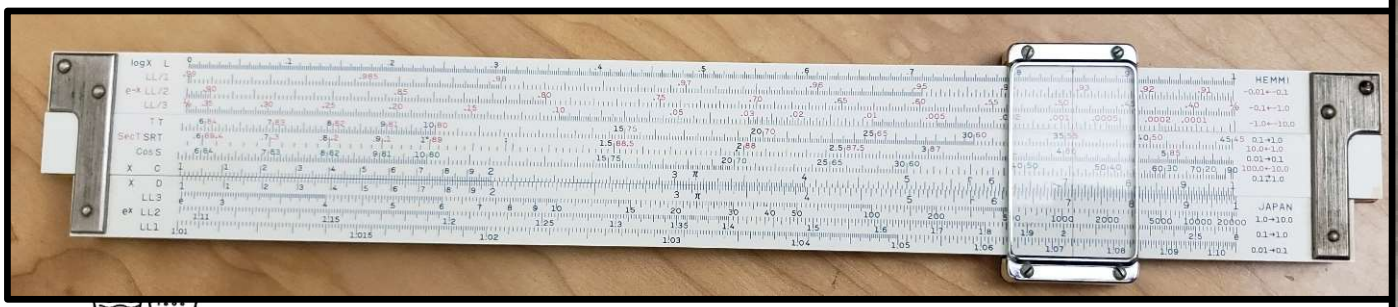
Book of sines and cosines

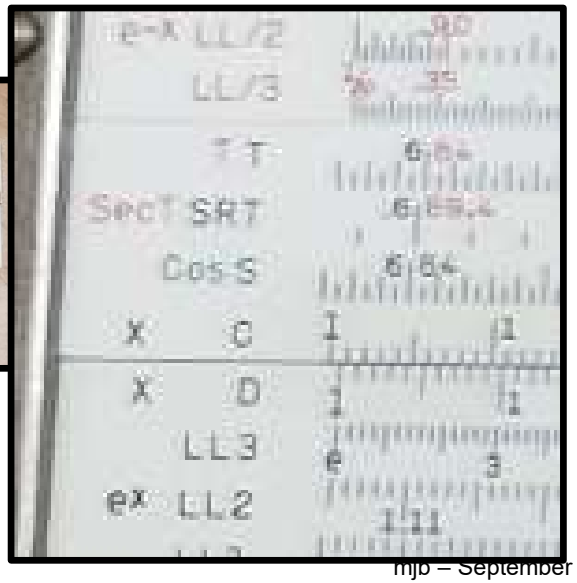




	30	9.256 1790	1148	9.992 8175	39	9.263 3615	1186	0.736 6385	30
	40	9.256 2938	1147	9.992 8136	38	9.263 4801	1186	0.736 5199	20
	50	9.256 4085	1148	9.992 8098	39	9.263 5987	1186	0.736 4013	10
24	0	9.256 5233	1147	9.992 8059	38	9.263 7173	1186	0.736 2827	0
	10	9.256 6380	1146	9.992 8021	39	9.263 8359	1186	0.736 1641	50
	20	9.256 7526	1147	9.992 7982	39	9.263 9545	1185	0.736 0455	40
	30	9.256 8673	1146	9.992 7943	38	9.264 0730	1184	0.735 9270	30
	40	9.256 9819	1146	9.992 7905	39	9.264 1914	1185	0.735 8086	20
	50	9.257 0965	1145	9.992 7866	39	9.264 3099	1184	0.735 6901	10
25	0	9.257 2110	1145	9.992 7827	39	9.264 4283	1184	0.735 5717	0
	10	9.257 3255	1145	9.992 7788	38	9.264 5467	1184	0.735 4533	50
	20	9.257 4400	1145	9.992 7750	39	9.264 6651	1183	0.735 3349	40
	30	9.257 5545	1144	9.992 7711	39	9.264 7834	1183	0.735 2166	30
	40	9.257 6689	1144	9.992 7672	38	9.264 9017	1183	0.735 0983	20
	50	9.257 7833	1144	9.992 7634	39	9.265 0200	1182	0.734 9800	10
26	0	9.257 8977	1143	9.992 7595	39	9.265 1382	1182	0.734 8618	0
	10	9.258 0120	1143	9.992 7556	39	9.265 2564	1182	0.734 7436	50
	20	9.258 1263	1143	9.992 7517	39	9.265 3746	1181	0.734 6254	40
	30	9.258 2406	1142	9.992 7478	38	9.265 4927	1181	0.734 5073	30
	40	9.258 3548	1142	9.992 7440	39	9.265 6108	1181	0.734 3892	20
	50	9.258 4690	1142	9.992 7401	39	9.265 7289	1181	0.734 2711	10
27	0	9.258 5832	1141	9.992 7362	39	9.265 8470	1180	0.734 1530	0
	10	9.258 6973	1141	9.992 7323	39	9.265 9650	1180	0.734 0350	50

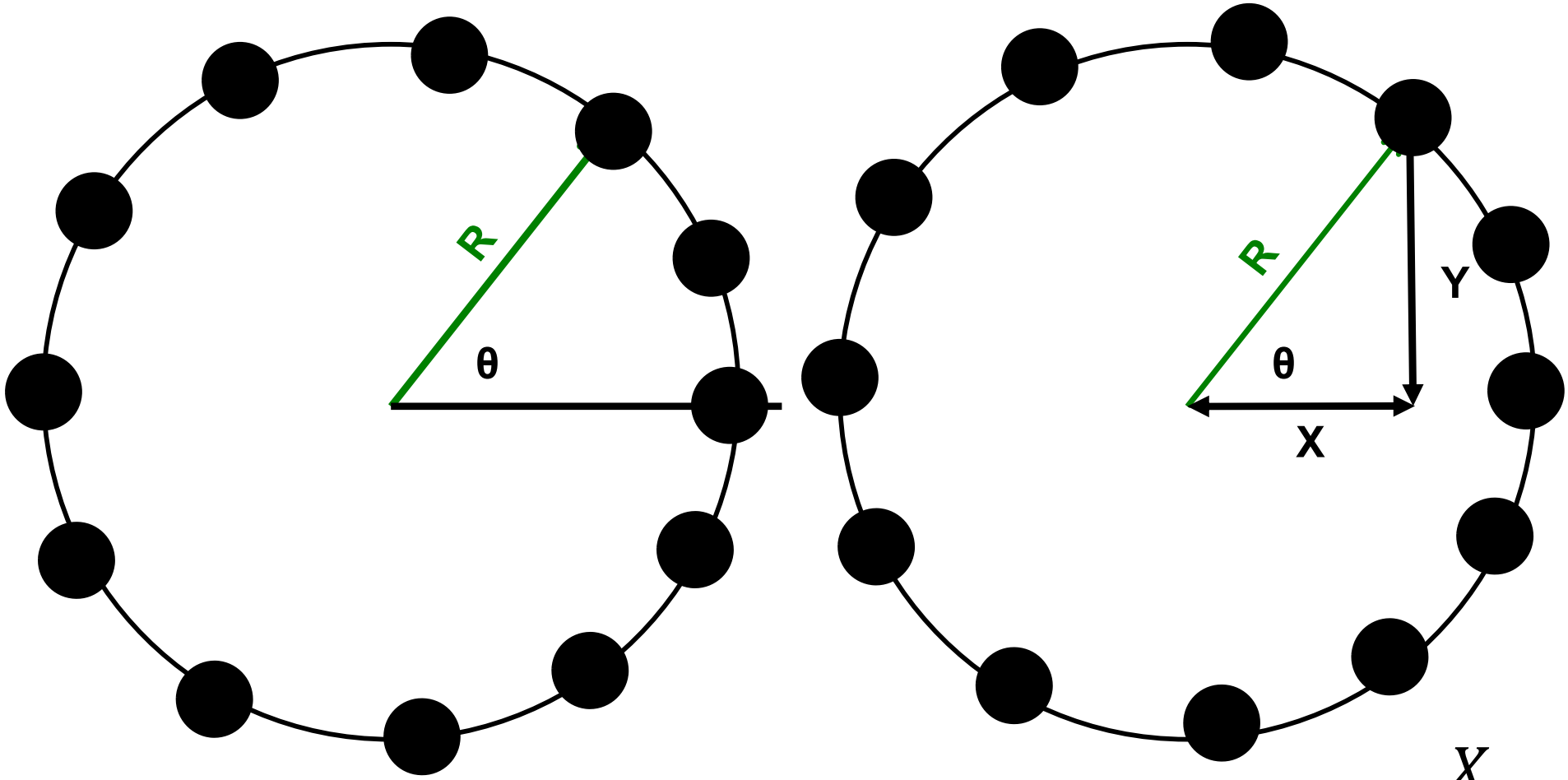
Slide rule





Oregon State University
Computer Graphics

First, We Need to Understand Something about Angles



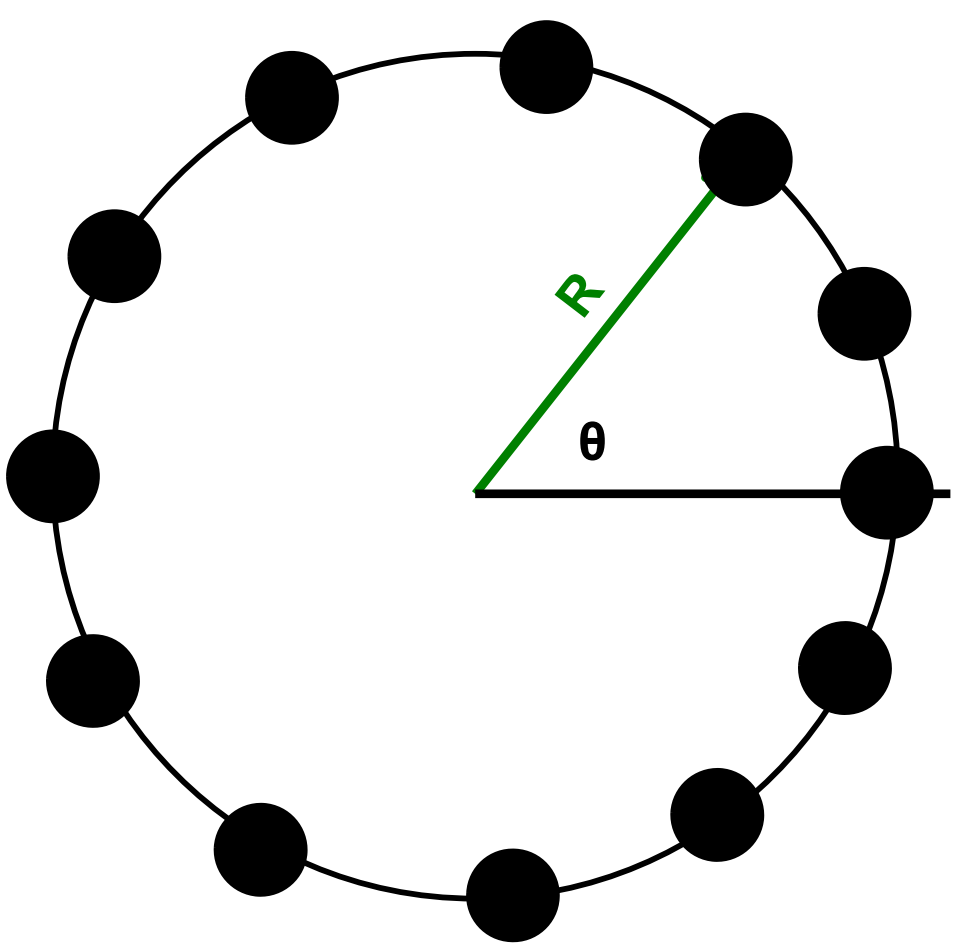
If we were to double the radius of the circle, all of the X's and Y's would also double.

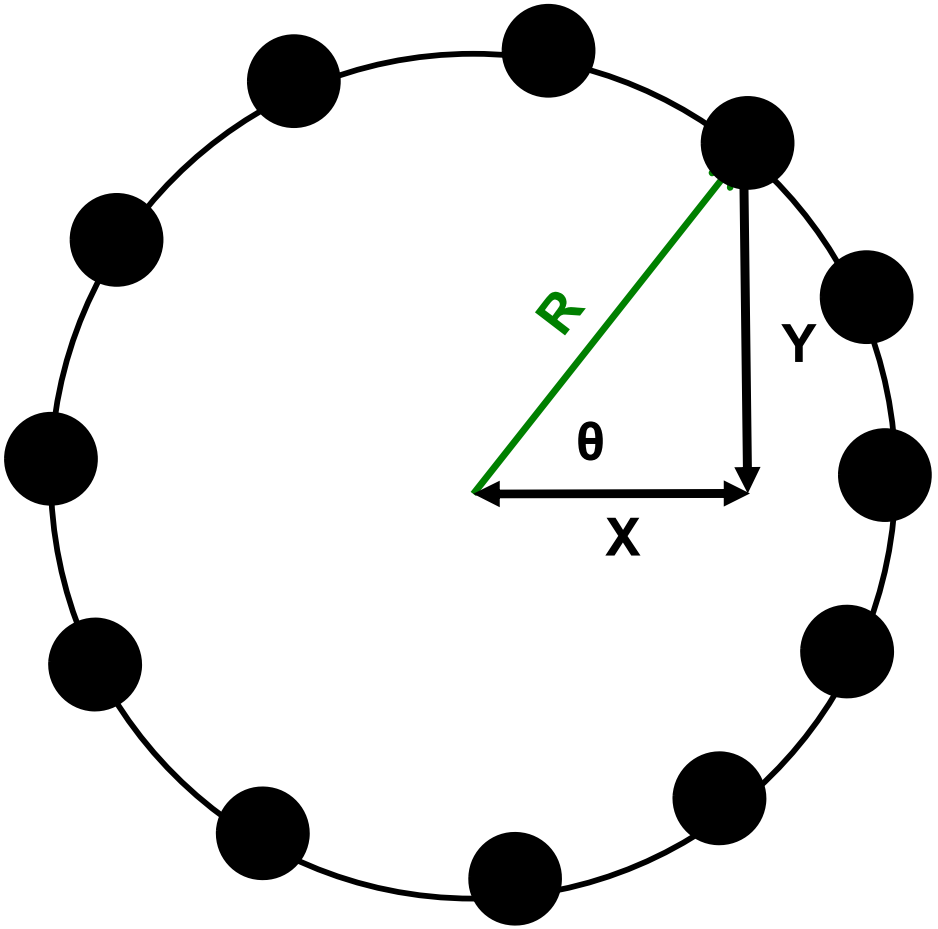
So, really the cos and sin are *ratios* of X and Y to the circle Radius

$$\cos\theta = \frac{X}{R}$$

$$\sin\theta = \frac{Y}{R}$$

First, We Need to Understand Something about Angles





So, if we know the circle Radius, and we march through a bunch of θ angles, we can determine all of the X's and Y's that we need to draw a circle.

$$\cos\theta = \frac{X}{R}$$

$$\sin\theta = \frac{Y}{R}$$

$$X = R * \cos\theta$$

$$Y = R * \sin\theta$$

Draw to this point

Thus, We Could Create Our Very Own Circle-Drawing Function

Circle center

Circle radius

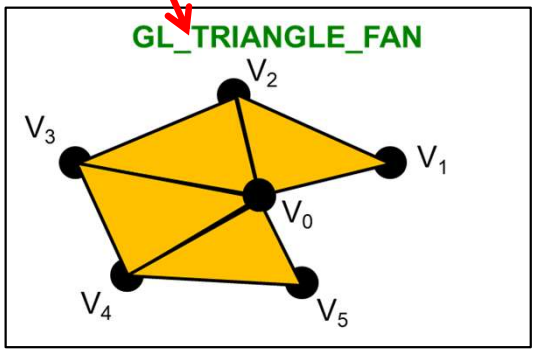
```
void
Circle( float xc, float yc, float r, int numsegs )
{
  float dang = 2.f * F_PI / (float)numsegs;
  float ang = 0.;
  glBegin( GL_TRIANGLE_FAN );
  glVertex3f( xc, yc, 0. );

  for( int i = 0; i <= numsegs; i++)
  {
    float x = xc + r * cosf(ang);
    float y = yc + r * sinf(ang);
    glVertex3f(  x, y, 0. );
    ang += dang;
  }

 glEnd( );
}
```

numsegs is the number of line segments making up the circumference of the circle.

numsegs=20 gives a nice circle.

5 gives a pentagon.
8 gives an octagon.
4 gives you a square. Etc.

2π is how many radians are in a full circle



The C/C++ sin( ) and cos( ) functions use double-precision floating point.

The C/C++ sinf( ) and cosf( ) functions use single-precision floating point, and are faster.

Why 2.*PI ?

```
float dang = 2.f*F_PI / (float)numsegs;
```

*We humans commonly measure angles in **degrees**, but science and computers like to measure them in something else called **radians**.*

There are 360° in a complete circle.
There are 2π radians in a complete circle.

*The built-in cosf( ) and sinf( ) functions expect angles to be given in **radians**.*

To convert between the two:

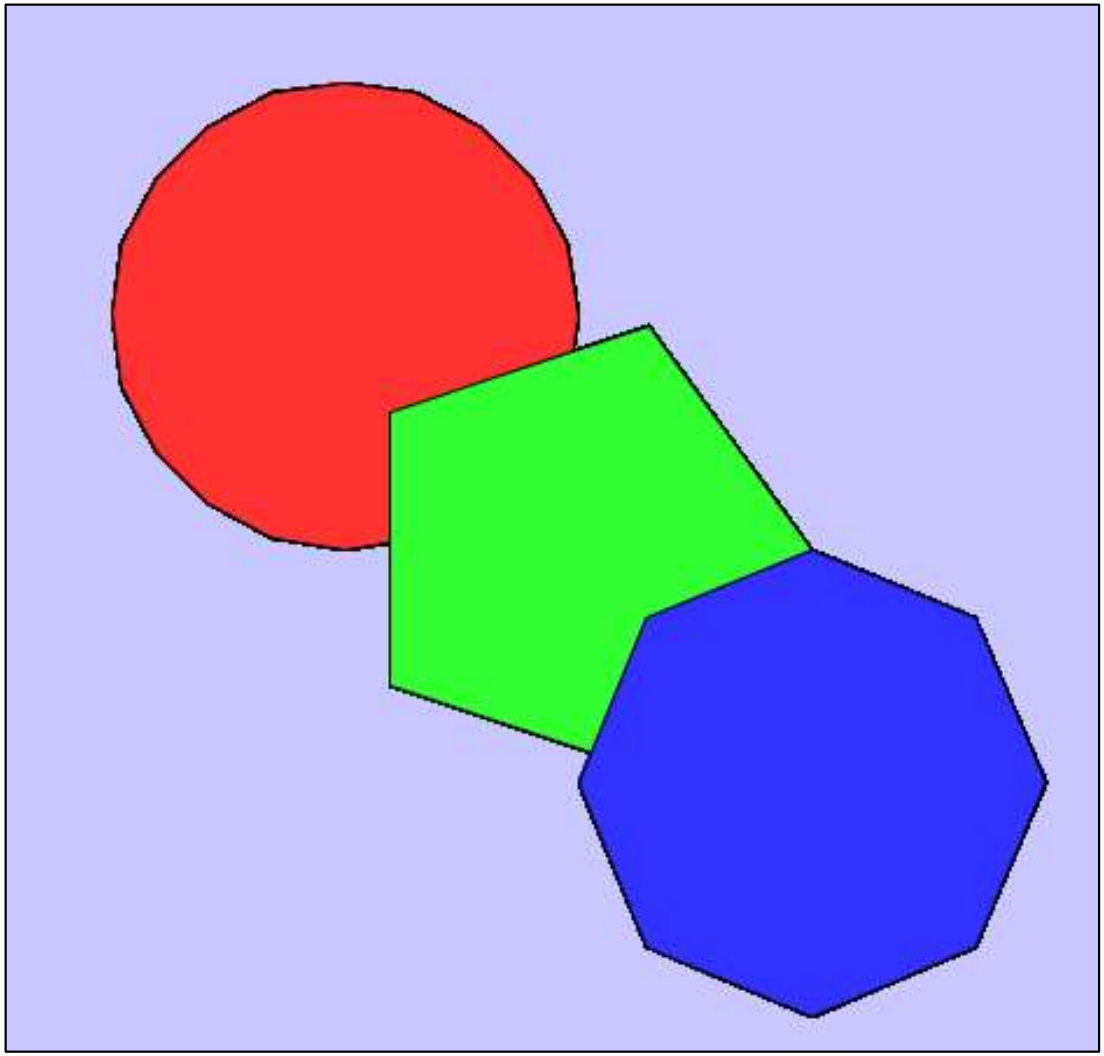
```
float rad  = deg * ( F_PI/180.f );
float deg = rad *  ( 180.f/F_PI );
```

Oregon State University
Computer Graphics

glRotatef( ) and gluPerspective( ) are the only two programming functions I can think of that use degrees. All others use radians!

Circles and Pentagons and Octagons, Oh My!

```
glColor3f( 1., 0., 0. );
Circle( 1.f, 3.f, 1.f, 20 )

glColor3f( 0., 1., 0. );
Circle( 2.f, 2.f, 1.f, 5 )

glColor3f( 0., 0., 1. );
Circle( 3.f, 1.f, 1.f, 8 )
```



Easy as π: M_PI vs. F_PI

The math.h include file has a definition of π that looks like this:

```
#define M_PI        3.14159265358979323846
```

Which will work just fine for whatever you need it for.

But, Visual Studio goes a little crazy complaining about mixing doubles (which is what M_PI is in) and floats (which is probably what you use most often). So, your sample code has these lines in it:

```
#define F_PI        ((float)(M_PI))
#define F_2_PI      ((float)(2.f*F_PI))
#define F_PI_2      ((float)(F_PI/2.f))
```

π

2π

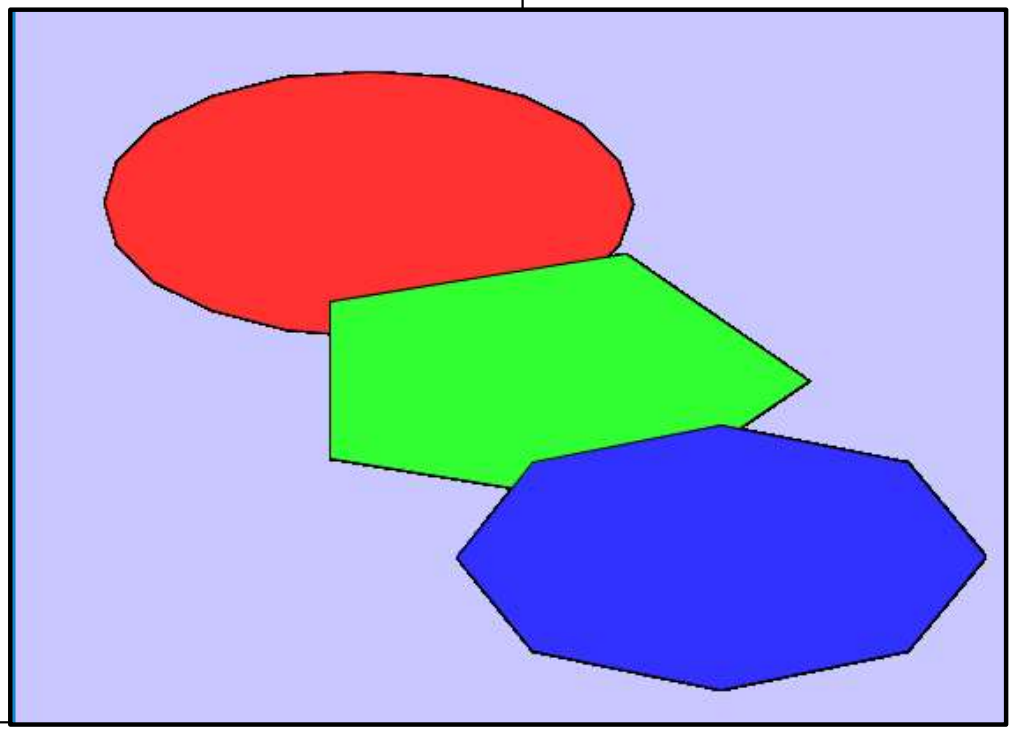
$\pi/2$

I use the *F_* version a lot because it keeps VS quiet. You can use either.

Oregon State University
Computer Graphics

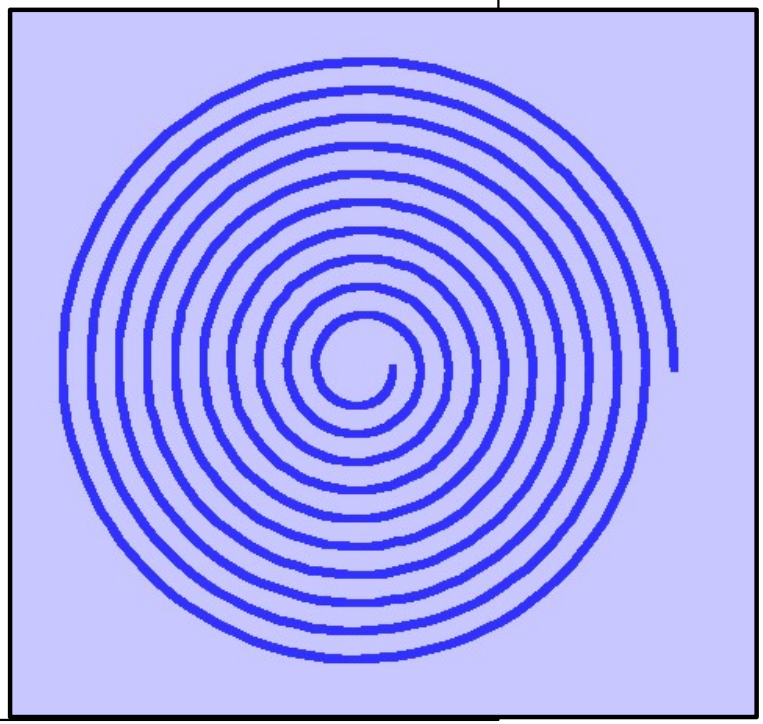
And, there is no reason the X and Y radii need to be the same…

```
void
Ellipse( float xc, float yc, float rx, float ry, int numsegs )
{
  float dang =  2.f * F_PI / (float)numsegs;
  float ang = 0.;
  glBegin( GL_TRIANGLE_FAN );
  glVertex3f( xc, yc, 0. );

  for( int i = 0; i <= numsegs; i++ )
  {
    float x = xc + rx * cosf(ang);
    float y = yc + ry * sinf(ang);
    glVertex3f( x, y, 0. );
    ang += dang;
  }

  glEnd( );
}
```



There is also no reason we can't gradually change the radius …

```
void
Spiral( float xc, float yc, float r0, float r1, int numsegs, int numturns )
{
  float dang = (float)numturns * 2.f * F_PI / (float)numsegs;
  float ang = 0.;
  glBegin( GL_LINE_STRIP );

  for( int i = 0; i <= numsegs; i++ )
  {
    float t = (float)i / (float)numsegs;      // 0.-1.
    float newrad = (1.-t)*r0 + t*r1;
                          // linearly interpolate from r0 to r1
    float x = xc + newrad * cosf(ang);
    float y = yc + newrad * sinf(ang);
    glVertex3f( x, y, 0. );
    ang += dang;
  }

  glEnd( );
}
```



Parametric Linear Interpolation (Blending)

What's this code all about?

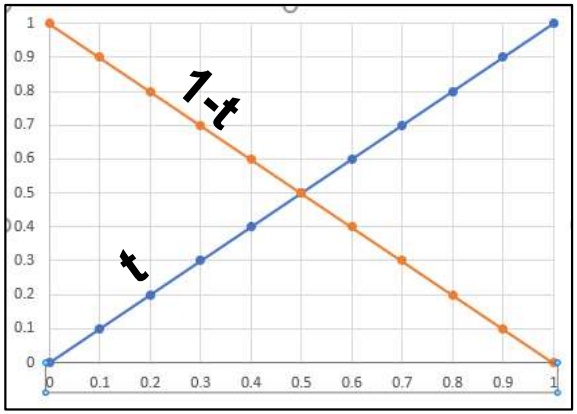
```
float t = (float)i / (float)numsegs;                    // 0.-1.
float newrad = (1.-t)*r0 + t*r1;
```

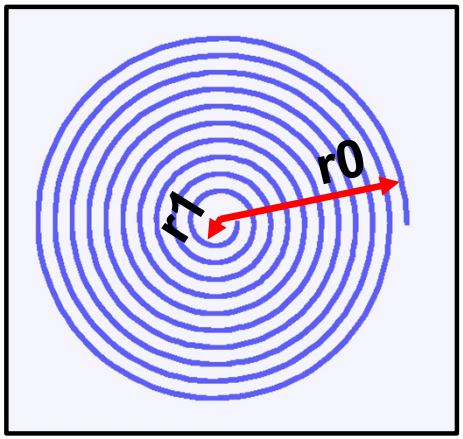
In computer graphics, we do a lot of linear interpolation between two input values. Here is a good way to do that:

1. Setup a float variable, ***t***, such that it ranges from 0. to 1.
   The line **float t = (float)i / (float)numsegs;** does this.

2. Step through as many ***t*** values as you want interpolation steps.
   The line **for(int i = 0; i <= numsegs; i++)** does this.

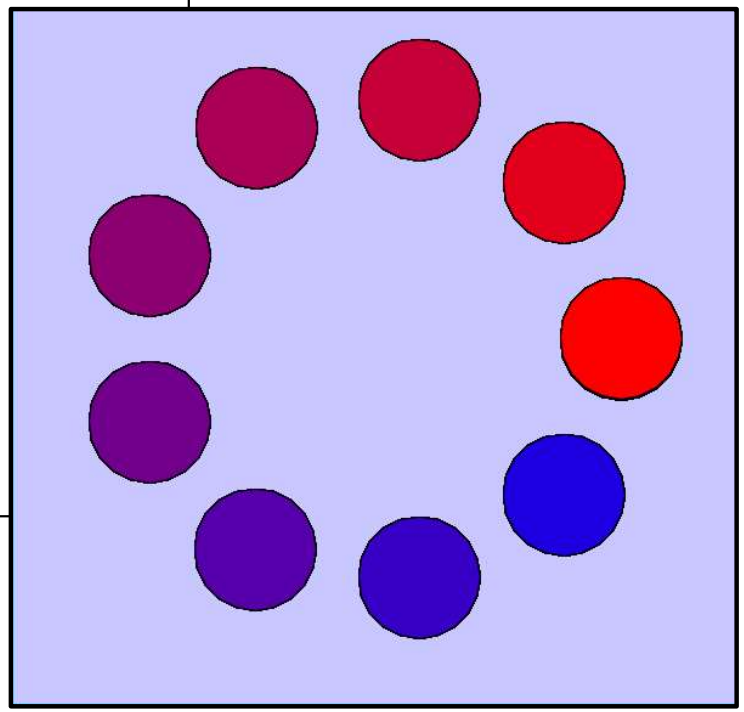
3. For each ***t***, multiply one input value by ***(1.-t)*** and multiply the other input value by **t** and add them together.
   The line **float newrad = (1.-t)*r0 + t*r1;** does this.





We Can Also Use This Same Idea to Arrange Things in a Circle and Linearly Blend Their Colors

```
int numObjects = 9;
float radius = 2.f;
float xc = 3.f;
float yc = 3.f;
int numSegs = 20;
float r = 50.f;
float dang = 2.f*F_PI / (float) ( numObjects - 1 );
float ang = 0.;
for( int i = 0; i < numObjects; i++ )
{
  float x = xc + radius * cosf(ang);
  float y = yc + radius * sinf(ang);
  float t = (float)i / (float)(numObjects-1);    // 0.-1.
  float red   = t;                               // ramp up
  float blue = 1.f - t;                          // ramp down
  glColor3f red, 0., blue );
  Circle( x, y, r, numSegs );
  ang += dang;
}
```



By Understanding what the Sine Function Looks Like, We Can Also Use it to Control Animations Based on Time

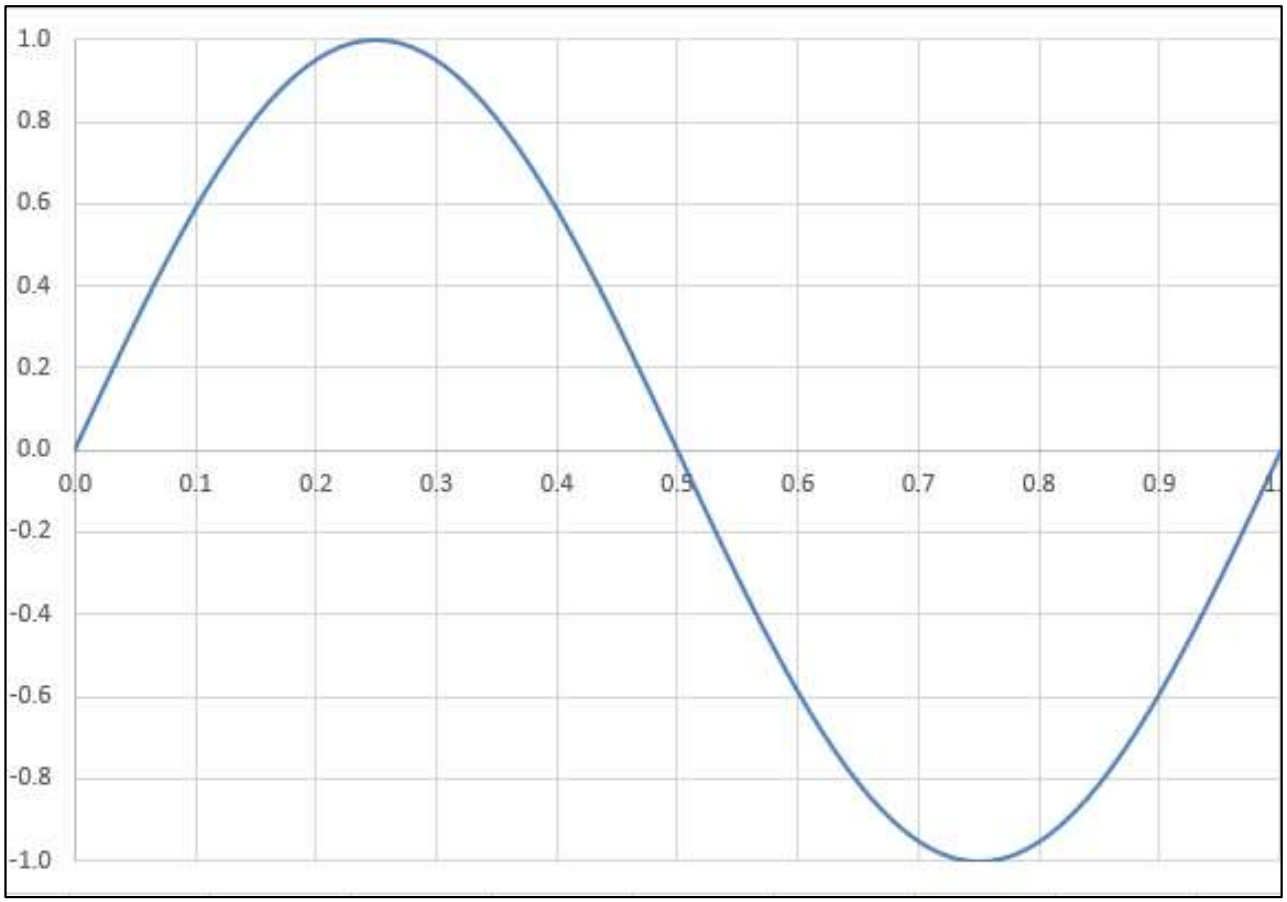
In your sample.cpp file, we have some code that looks like this:

```
float Time;                         // global variable intended to lie between [0.,1.)

. . .

const int MS_PER_CYCLE = 10000;     // 10000 milliseconds = 10 seconds

. . .

// in Animate( ):
        int ms = glutGet(GLUT_ELAPSED_TIME);
        ms %= MS_PER_CYCLE;
                        // makes the value of ms between 0 and MS_PER_CYCLE-1
        Time = (float)ms / (float)MS_PER_CYCLE;
                        // makes the value of Time between 0. and slightly less than 1.
```

By Understanding what the Sine Function Looks Like, We Can Also Use it to Control Animations Based on Time

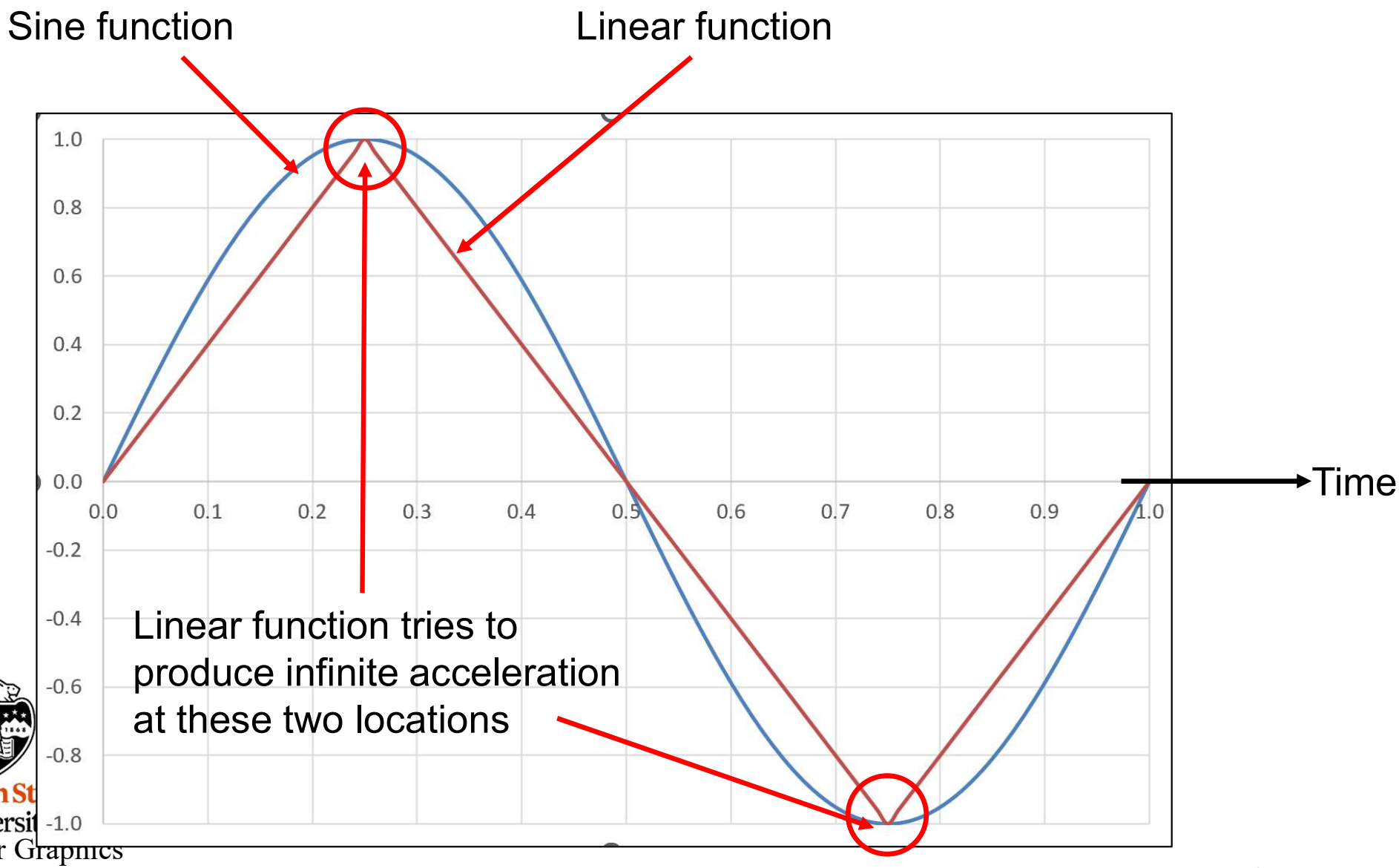
The sine function goes from -1. to +1., and does it very smoothly

$$y=sin(2.*\pi*Time)$$



By Understanding what the Sine Function Looks Like, We Can Also Use it to Control Animations Based on Time

Sine functions produce a smoother set of motions than linear functions do (that's why we use them):

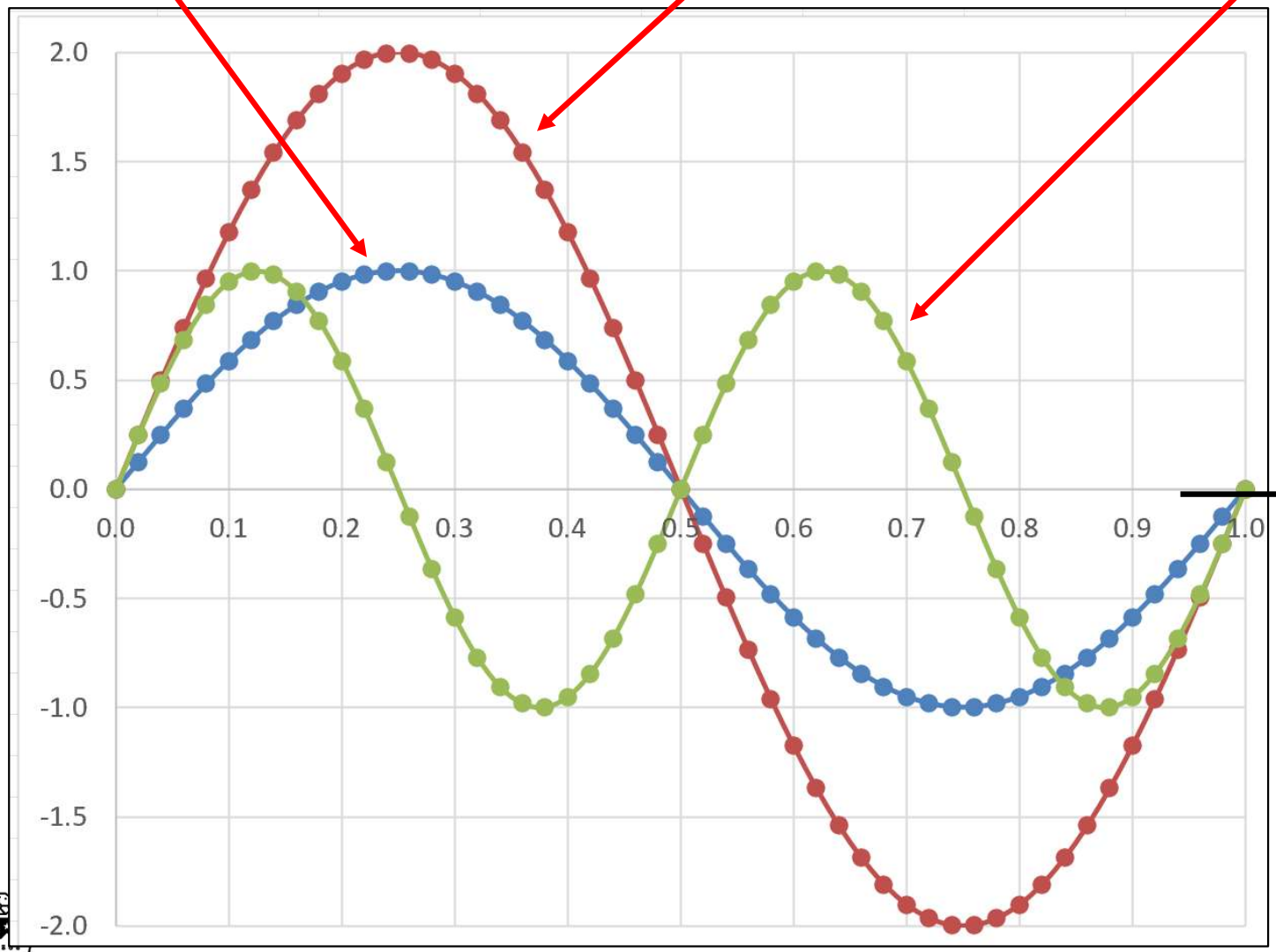


Increasing the Amplitude, Increasing the Frequency

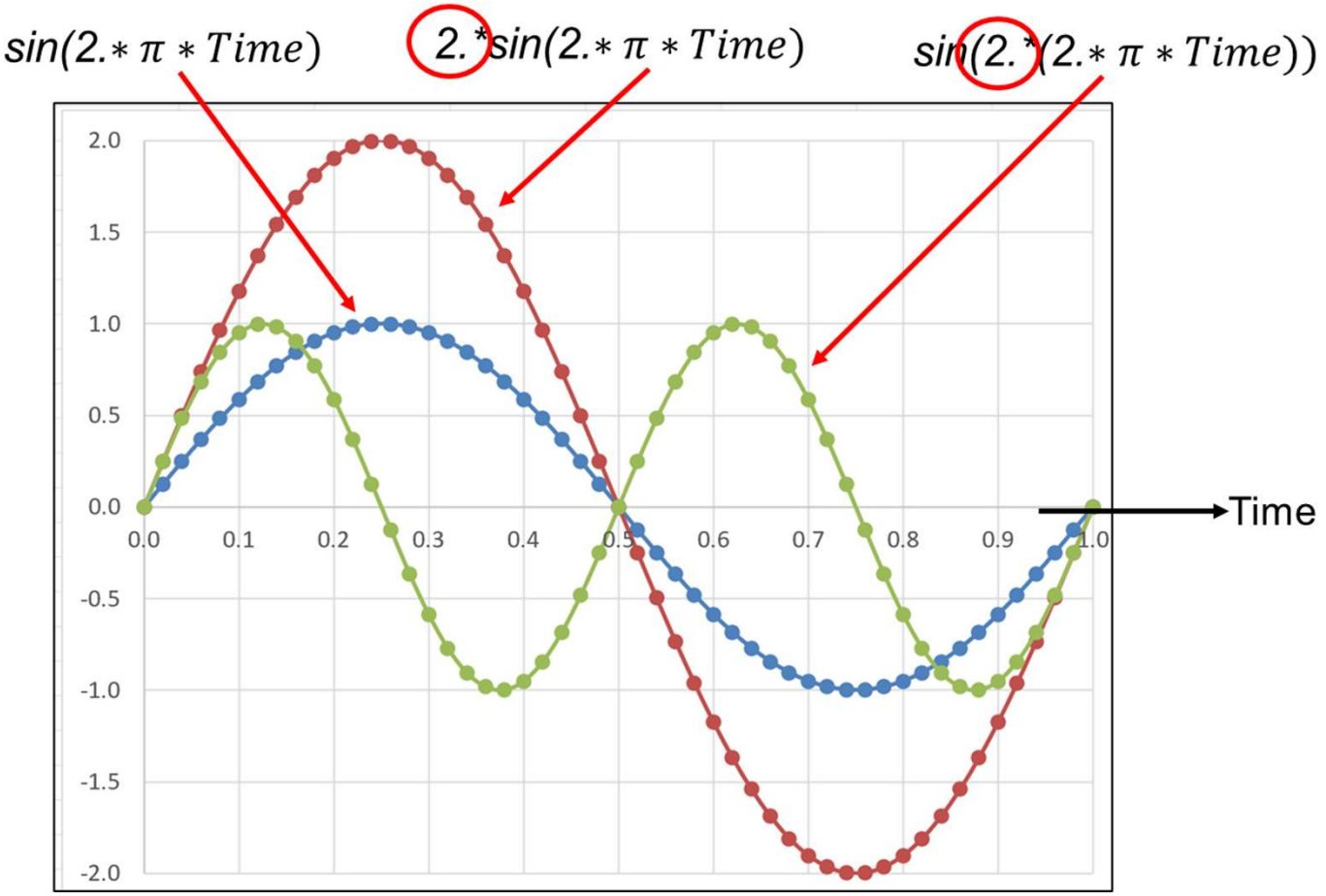
$sin(2.* \pi * Time)$

$2.*sin(2.* \pi * Time)$

$sin(2.*(2.* \pi * Time))$



Increasing the Amplitude, Increasing the Frequency



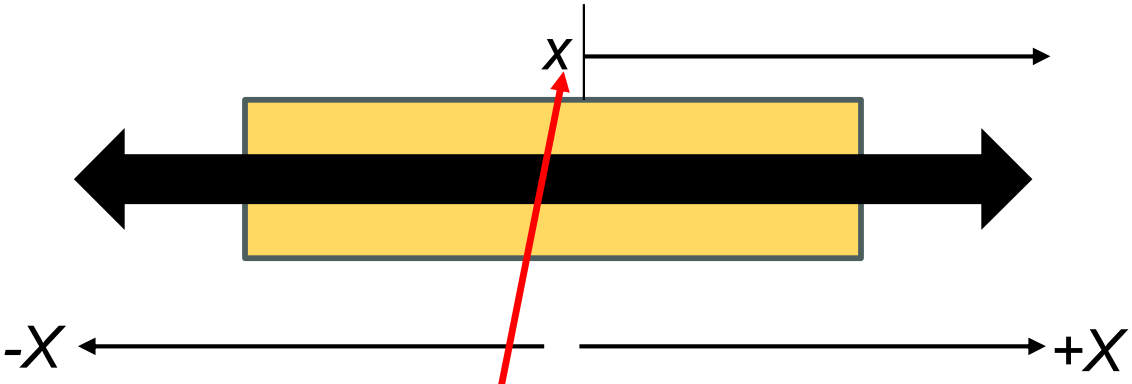
$$A*\sin(F*(2.*\pi*Time))$$

Changing this number changes the Amplitude

Changing this number changes the Frequency

Oscillating Motion

Let's say you want a block to oscillate back and forth in x:

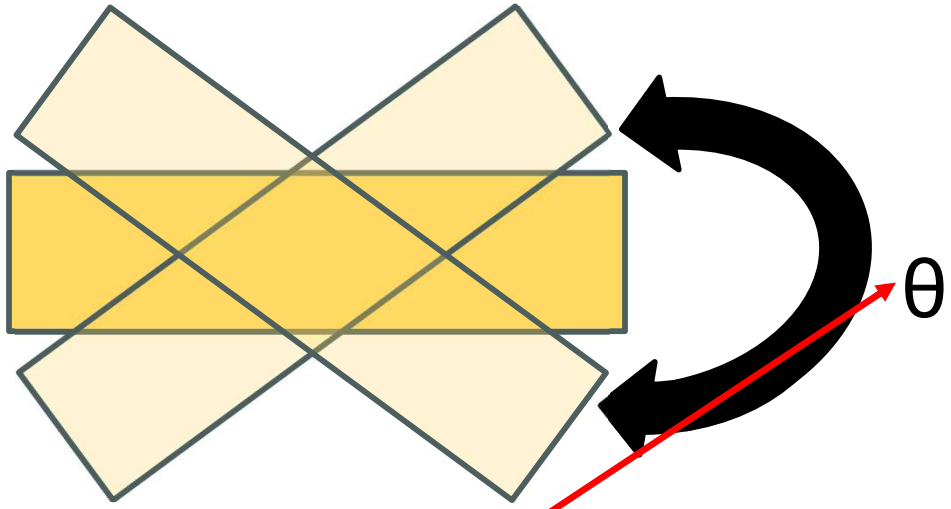


This code would cause it to do that:

```
// in Display( ):
        float x = X*sin(F*(2.* π * Time) )
        . . .
        glTranslatef( x, 0., 0. );
        glCallList( BlockList );
```

Rocking Motion

Let's say you want a block to rock back and forth:



This code would cause it to do that:

```
// in Display( ):
        float theta = 45.f * sin(F *(2.* π * Time) )
        . . .
        glRotatef( theta,    0., 0., 1. );
        glCallList( BlockList );
```

OBJ Files

Oregon State University

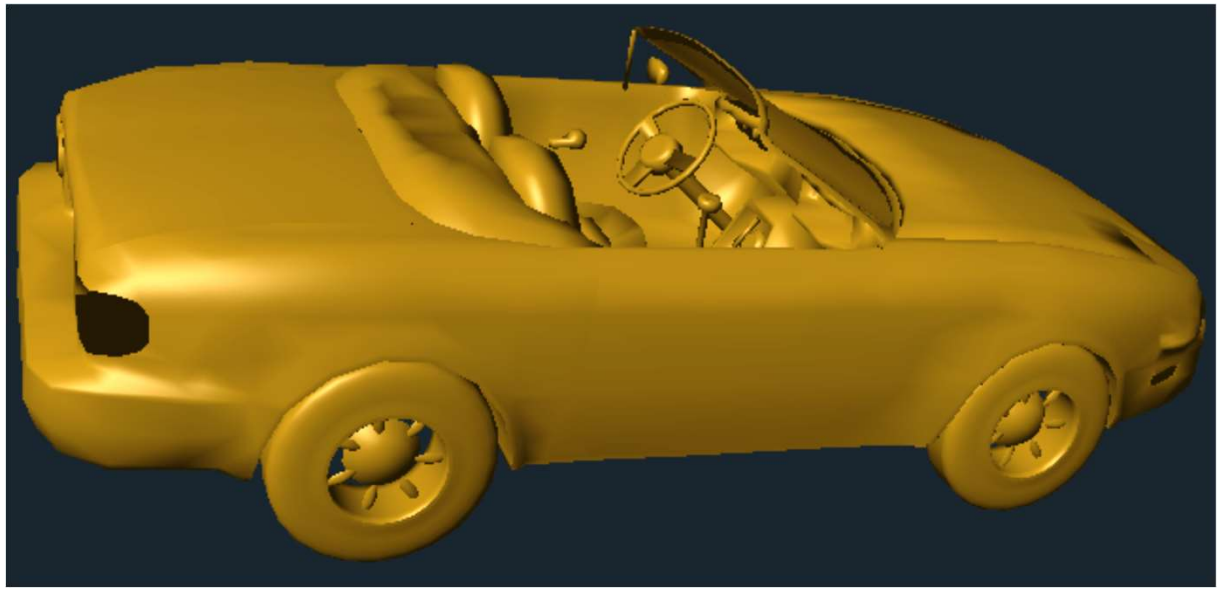
Mike Bailey

mjb@cs.oregonstate.edu

Oregon State University
Computer Graphics

OBJ Files

An OBJ file is a way to transmit 3D geometry information from one program to your OpenGL program. As there are thousands of free OBJ files out there (and a lot of paid ones too), this is a great way to get fun geometry into your program without you having to create it yourself.



The Parts of an OBJ File

v 2.229345 -0.992723 -0.862826
v 2.292449 -0.871852 -0.882400
v 2.410367 -0.777999 -0.841105
v 2.407309 -0.974980 -0.805091
. . .

Vertices

vt 0.202747 0.304978
vt 0.201052 0.414168
vt 0.137383 0.357003
vt 0.263749 0.402974
vt 0.102404 0.424003
. . .

Per-vertex Texture Coordinates
(if there are none of these, you cannot texture this model)

vn 0.628361 -0.426126 -0.650830
vn 0.437900 -0.250054 -0.863549
vn 0.709718 -0.453838 -0.538824
vn 0.720876 -0.356662 -0.594247
. . .

Per-vertex Normals
(if there are none of these, you can only apply per-face lighting to this model)

f 11/11/26 12/15/25 13/19/27
f 13/19/28 12/15/29 15/24/30
f 12/15/29 14/28/31 15/24/30
f 15/24/32 14/28/33 16/34/34
f 16/34/35 14/28/36 18/33/37

Faces
(for each vertex of a face, the numbers are vertex number **/** texcoord-number **/** normal-number)
(to make it more confusing, the texcoord number and the normal number don't need to be there)
(to make it even more confusing, these indices are 1-based, not 0-based)

OBJ File Samples

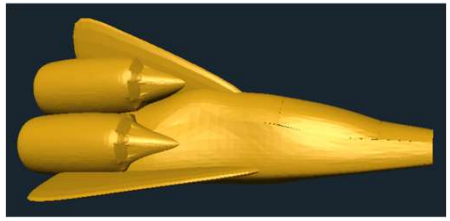
cow.obj

dino.obj

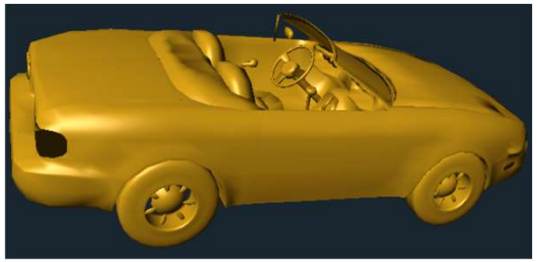
deer.obj

cat.obj

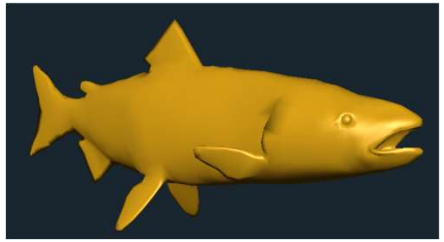
skeleton.obj



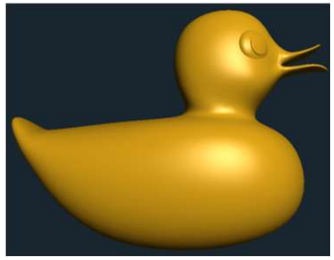
spaceship.obj



car.obj



salmon.obj



ducky.obj

OBJ File Samples

You can find these and more at: http://cs.oregonstate.edu/~mjb/cs557/Obj/

Or, look at the end of the Class Resources Page

Or, Google the phrase: ***Free OBJ Files***

Or, my favorite go-to site for free models (and paid ones) is: https://www.turbosquid.com/

We are Giving You a Function to Load an Obj File into Your Program

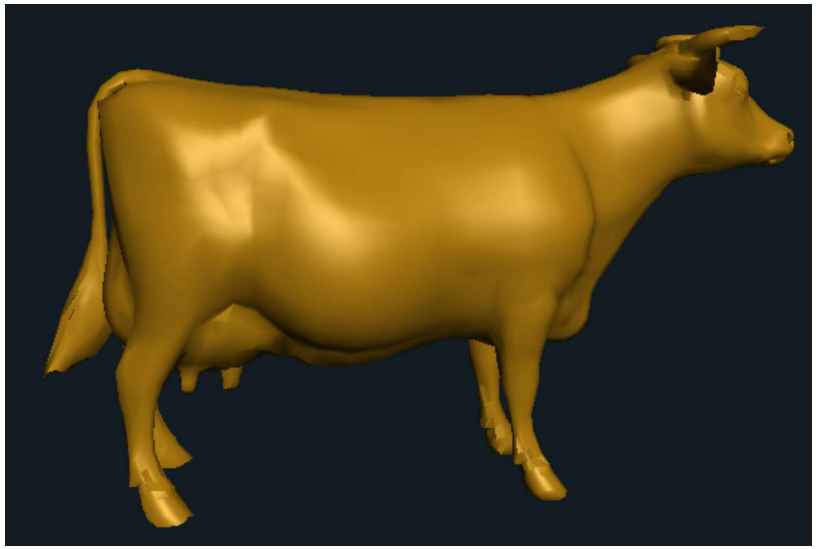
The code for this is in the file: **loadobjfile.cpp**

I usually use this by reading the .obj object into a display list, like this:

```
// create a global variable:
GLuint  DL;

. . .

// do this in InitLists( ):

DL = glGenLists( 1 );
glNewList( DL, GL_COMPILE );
        LoadObjFile( "spaceship.obj" );
glEndList( );

. . .

// do this in Display( ):

glCallList( DL );
```

But, you need to use Lighting!

You are advised not to use these models until we have covered OpenGL lighting!



With lighting – cool!

Without lighting – blech!