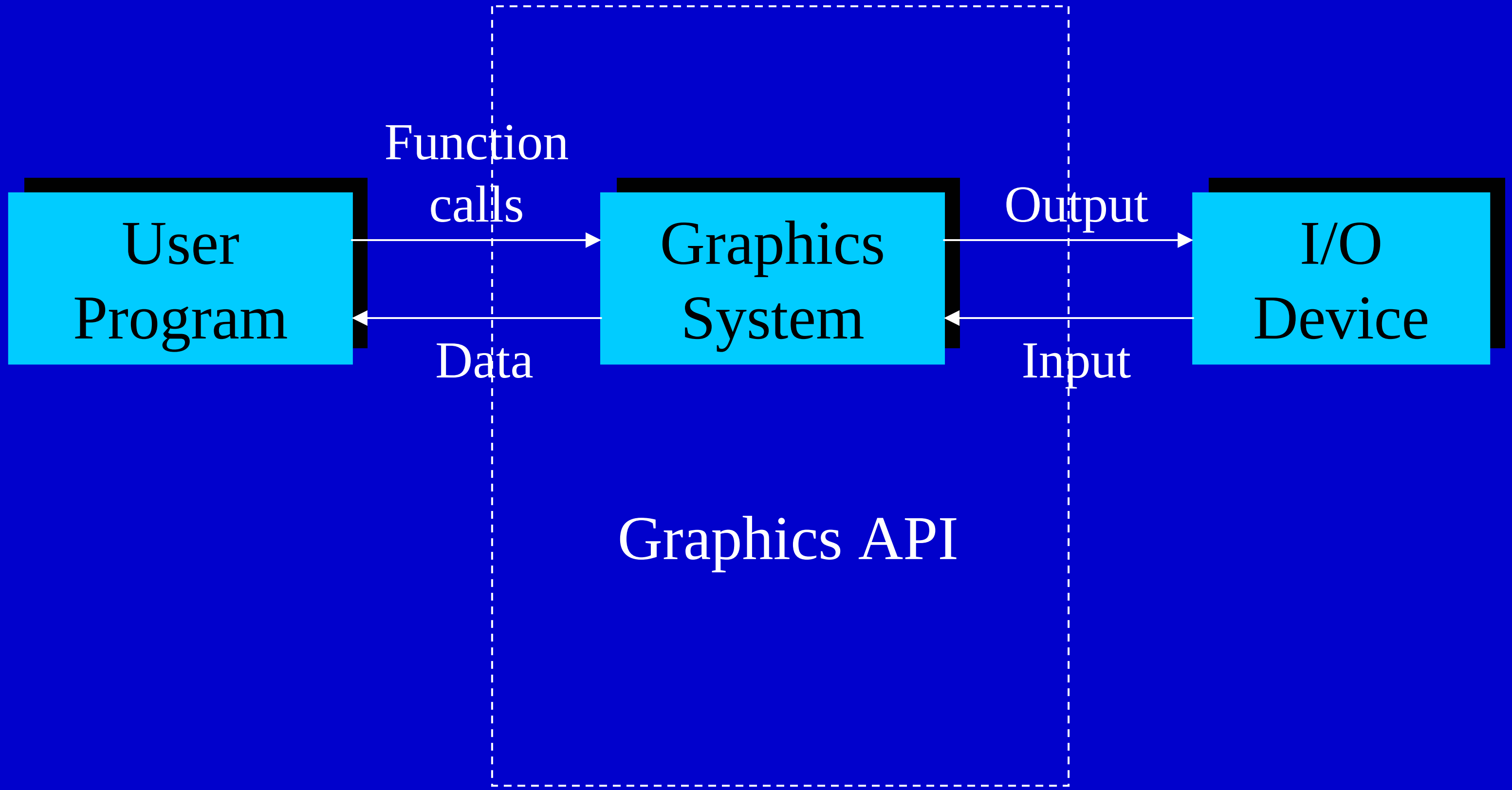
Graphics Programming

Introduction to OpenGL

Outline

- What is OpenGL
- OpenGL Rendering Pipeline
- OpenGL Utility Toolkits
- OpenGL Coding Framework
- OpenGL API

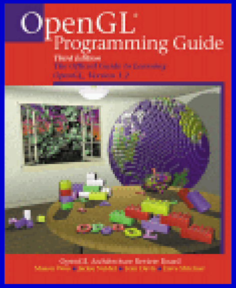
Graphics System

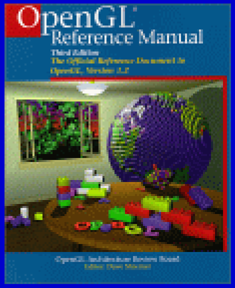


What is OpenGL

OpenGL

- A software interface to graphics hardware
- A 3D graphics rendering API (>120 functions)
- Hardware independent
- Very fast (a standard to be accelerated)
- Portable





http://www.opengl.org

A History of OpenGL

- Was SGI's Iris GL – "Open"**GL**
- "Open" standard allowing for wide range hardware platforms
- OpenGL v1.0 (1992)
- OpenGL v1.1 (1995)
- OpenGL v1.4 (latest)
- Governed by OpenGL Architecture Review Board (ARB)

"Mesa" – an Open source (http://www.mesa3d.org)

Graphics Process

Geometric Primitives

Image Primitives

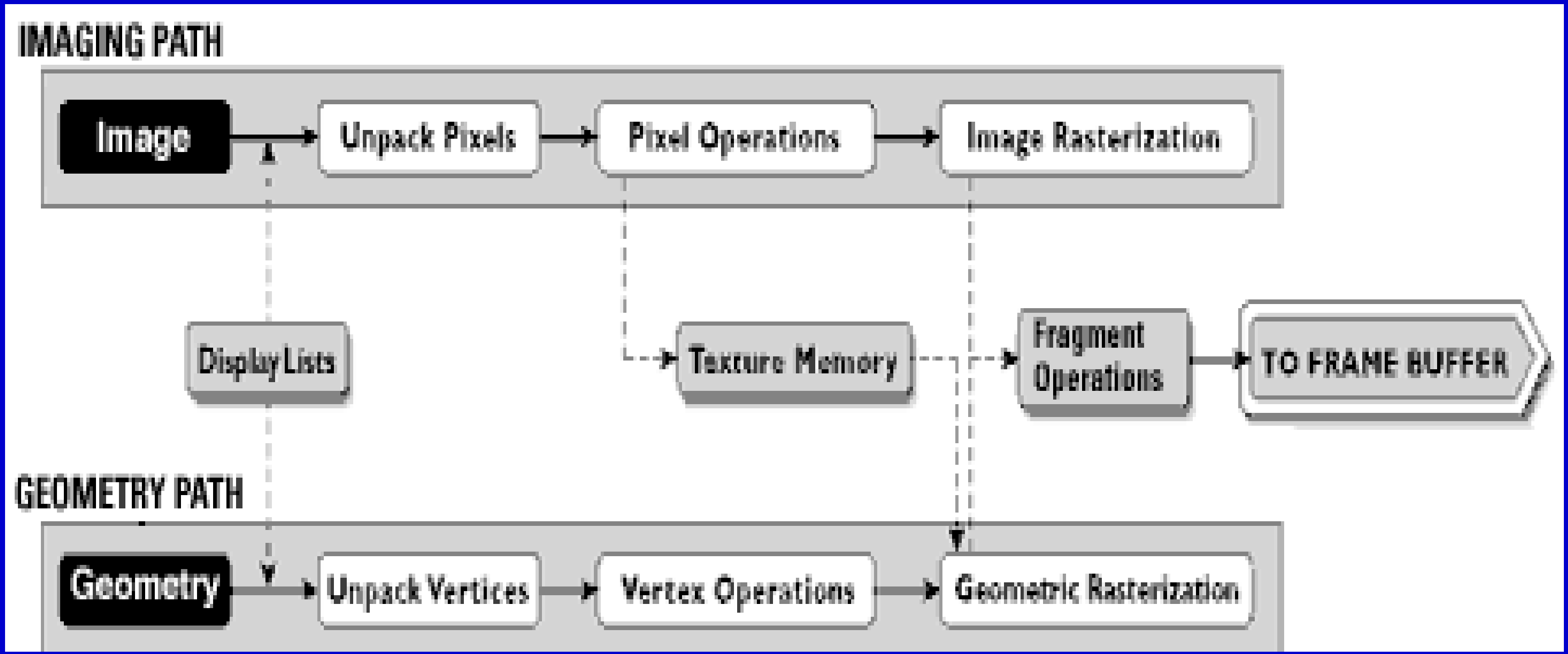
Rendering

Frame Buffer

+

=

OpenGL Architecture



OpenGL is Not a Language

It is a Graphics Rendering API

Whenever we say that a program is *OpenGL-based* or *OpenGL applications*, we mean that it is written in some programming language (such as C/C++) that makes calls to one or more of OpenGL libraries

Window Management

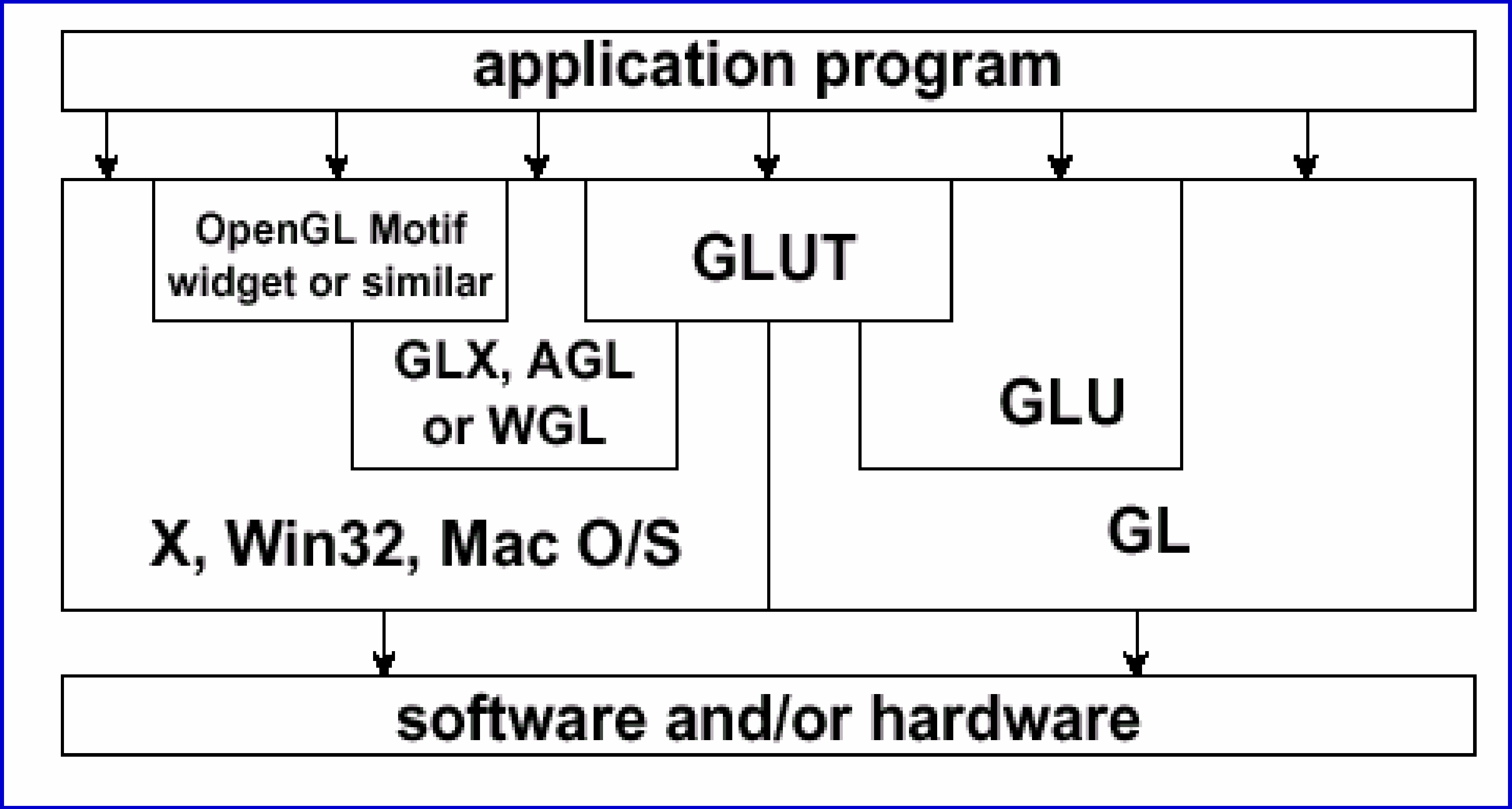
- OpenGL is window and operating system independent
- OpenGL does *not* include any functions for window management, user interaction, and file I/O
- Host environment is responsible for window management

Window Management API

We need additional libraries to handle window management

- GLX/AGL/WGL
  - glue between OpenGL and windowing systems
- GLUT
  - OpenGL Window Interface/Utility toolkit
- AUX
  - OpenGL Utility Toolkit

OpenGL API Hierarchy



OpenGL Division of Labor

- **GL**
  - “core” library of OpenGL that is platform independent
- **GLU**
  - an auxiliary library that handles a variety of graphics accessory functions
- **GLUT/AUX**
  - utility toolkits that handle window managements

Libraries and Headers

Library Name	Library File	Header File	Note
OpenGL	opengl32.lib (PC) -lgl (UNIX)	gl.h	"core" library
Auxiliary library	glu32.lib (PC) -lglu	glu.h	handles a variety of accessory functions
Utility toolkits	glut32.lib (PC) -lglut (UNIX) glaux.lib (PC) -lglaux (UNIX)	glut.h glaux.h	window managements

All are presented in the C language

Learning OpenGL with GLUT

- **GLUT** is a Window Manager (handles window creation, user interaction, callbacks, etc)
- Platform Independent
- Makes it *easy* to learn and write OpenGL programs without being distracted by your environment
- Not "final" code (Not meant for commercial products)

Environment Setup

- All of our discussions will be presented in C/C++ language
- Use GLUT library for window managements
- Files needed

  gl.h, glu.h, glut.h

  opengl32.lib, glu32.lib, glut32.lib
- Go to http://www.opengl.org download files
- Follow the Setup instruction to configure proper path

Usage

Include the necessary header files in your code

```
#include <GL/gl.h>          // "core", the only thing is required
#include <GL/glu.h>         // handles accessory functions
#include <GL/glut.h>        // handles window managements

void main( int argc, char **argv )
{
    .....
}
```

Only the "core" library (opengl32.lib, gl.h) are required

Usage

Link the necessary Libraries to your code

- Link **GL** library
  - Link opengl32.lib (PC), or -lgl (UNIX)
- Link **GLU** library
  - Link glu32.lib (PC), or -lglu (UNIX)
- Link **GLUT** library
  - Link glut32.lib (PC), or -lglut (UNIX)

OpenGL Data Types

To make it easier to convert OpenGL code from one platform to another, OpenGL defines its own data types that map to normal C data types

GLshort A[10];	⟷	short A[10];
GLdouble B;	⟷	double B;

OpenGL Data Types

OpenGL Data Type	Representation	As C Type
GLbyte	8-bit integer	signed char
GLshort	16-bit integer	short
GLint, GLsizei	32-bit integer	long
GLfloat	32-bit float	float
GLdouble	64-bit float	double
GLubyte, GLboolean	8-bit unsigned integer	unsigned char
GLushort	16-bit unsigned short	unsigned short
GLunit, GLenum, GLbitfield	32-bit unsigned integer	unsigned long

OpenGL Function Naming

OpenGL functions all follow a naming convention that tells you which library the function is from, and how many and what type of arguments that the function takes

<Library prefix><Root command><Argument count><Argument type>

OpenGL Function Naming

`glColor3f(…)`

gl library   Root command, # of arguments, type of arguments

gl means OpenGL

glu means GLU

glut means GLUT

f: the argument is *float* type

i: the argument is *integer* type

v: the argument requires a *vector*

Basic OpenGL Coding Framework

1. **Configure GL (and GLUT)**
   – Open window, Display mode, ……
2. **Initialize OpenGL state**
   – background color, light, View positions, ……
3. **Register callback functions**
   – Render, Interaction (keyboard, mouse), ……
4. **Event processing loop**
   – glutMainLoop()

A Sample Program

```
void main (int argc, char **argv)
{
    glutInit (&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize (500, 500);                      1
    glutCreateWindow (“My First Program");
    myinit ();                                          2
    glutDisplayFunc ( display );
    glutReshapeFunc ( resize );                         3
    glutKeyboardFunc ( key );
    glutMainLoop ();                                    4
}
```

1: Initializing & Creating Window

Set up window/display you're going to use

```
void main (int argc, char **argv)
{
    glutInit (&argc, argv);                                   // GLUT initialization
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);   // display model
    glutInitWindowSize (500, 500);                            // window size
    glutCreateWindow (“My First Program");        // create window
     ……
}
```

GLUT Initializing Functions

- **Standard GLUT initialization**

  glutInit (int argc, char ** argv)

- **Display model**

  glutInitDisplayMode (unsigned int mode)

- **Window size and position**

  glutInitWindowSize (int width, int height)

  glutInitWindowPosition(int x, int y)

- **Create window**

  glutCreateWindow (char *name);

2: Initializing OpenGL State

Set up whatever state you're going to use

```
void myinit(void)
{
    glClearColor(1.0, 1.0, 1.0, 1.0);       // background color
    glColor3f(1.0, 0.0, 0.0);               // line color
    glMatrixMode(GL_PROJECTION);            // followings set up viewing
    glLoadIdentity();
    gluOrtho2D(0.0, 500.0, 0.0, 500.0);
    glMatrixMode(GL_MODELVIEW);
}
```

Callback Functions

- **Callback Function**

  Routine to call when something happens

  - window resize, redraw, user input, etc

- **GLUT uses a callback mechanism to do its event processing**

GLUT Callback Functions

- **Contents of window need to be refreshed**
  glutDisplayFunc()
- **Window is resized or moved**
  glutReshapeFunc()
- **Key action**
  glutKeyboardFunc()
- **Mouse button action**
  glutMouseFunc()
- **Mouse moves while a button is pressed**
  glutMotionFunc()
- **Mouse moves regardless of mouse button state**
  glutPassiveMouseFunc()
- **Called when nothing else is going on**
  glutIdleFunc()

3: Register Callback Functions

Set up any callback function you're going to use

```
void main (int argc, char **argv)
{
    ……
    glutDisplayFunc ( display );        // display callback
    glutReshapeFunc ( resize );         // window resize callback
    glutKeyboardFunc ( key );           // keyboard callback

    ……
}
```

Rendering Callback

It's here that does all of your OpenGL rendering

```
void display( void )
{
    typedef GLfloat point2[2];
    point2 vertices[3]={{0.0, 0.0}, {250.0, 500.0}, {500.0, 0.0}};
    int i, j, k;    int rand();
    glClear(GL_COLOR_BUFFER_BIT);
    for( k=0; k<5000; k++)
        ……
}
```

Window Resize Callback

It's called when the window is resized or moved

```
void resize(int w, int h)
{
    glViewport(0,  0,  w,  h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    ……
    display();
}
```

Keyboard Input Callback

It’s called when a key is struck on the keyboard

```
void key( char mkey,  int x,  int y )
{
    switch( mkey )
    {
        case ‘q’ :
            exit( EXIT_SUCCESS );
            break;
        ……
}
```

Event Process Loop

This is where your application receives events, and schedules when callback functions are called

```
void main (int argc, char **argv)
{
    ……
    glutMainLoop();
}
```

Let's go Inside

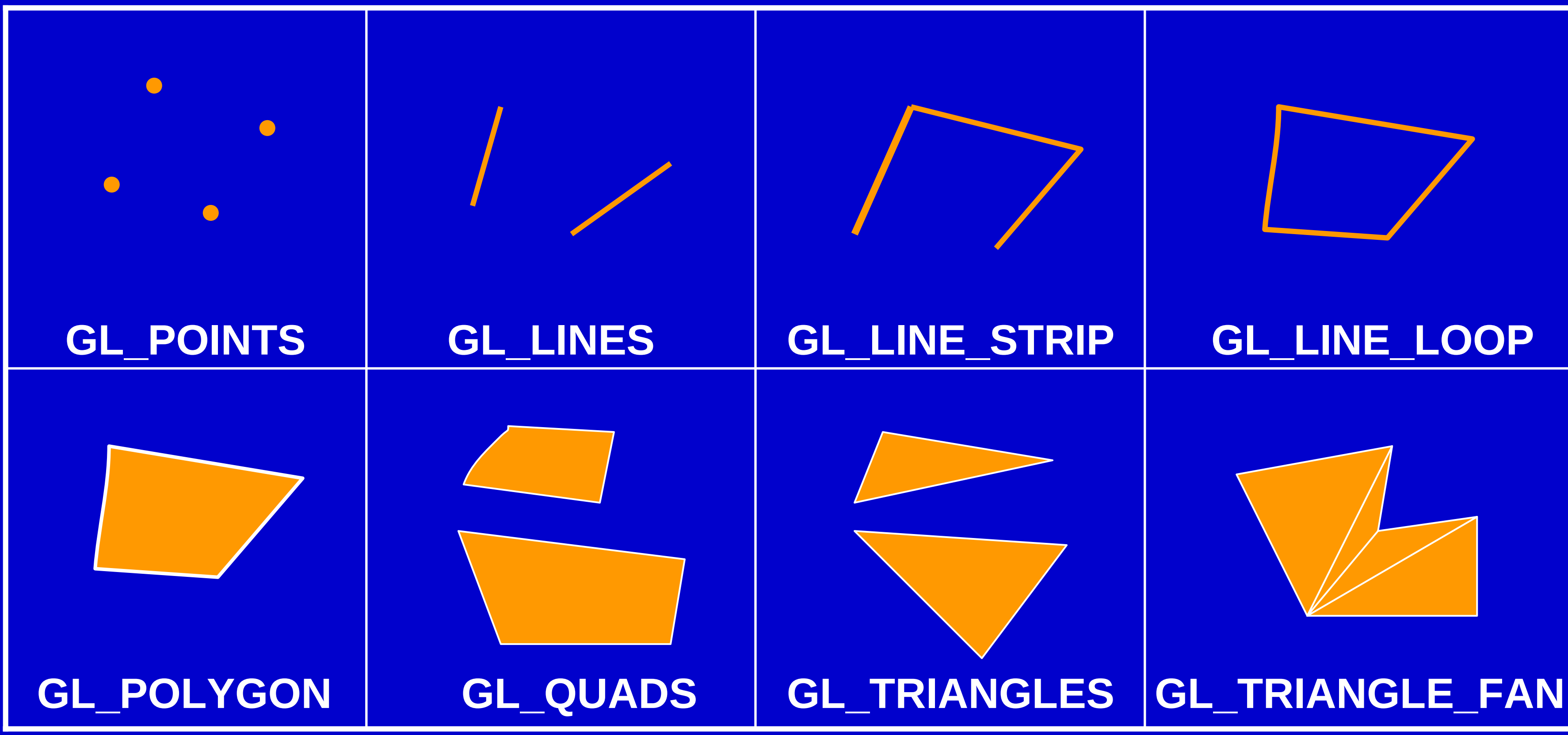
- OpenGL API
  - Geometric Primitives
  - Color Mode
  - Managing OpenGL's State
  - Transformations
  - Lighting and shading
  - Texture mapping

OpenGL API Functions

- **Primitives**
  - **A point, line, polygon, bitmap, or image**
- **Transformation**
  - **Rotation, size, perspective in 3D coordinate space**
- **Color mode**
  - **RGB, RGBA, Color index**
- **Materials lighting and shading**
  - **Accurately compute the color of any point given the material properties**
- **Buffering**
  - **Double buffering, Z-buffering, Accumulation buffer**
- **Texture mapping**

……

Geometric Primitives



All geometric primitives are specified by vertices

Geometry Commands

- **glBegin(GLenum type)**

  marks the beginning of a vertex-data list that describes a geometric primitives

- **glEnd (void)**

  marks the end of a vertex-data list

- **glVertex*(…)**

  specifies vertex for describing a geometric object

Specifying Geometric Primitives

```
glBegin( type );
    glVertex*(…);
    ……
    glVertex*(…);
glEnd();
```

type determines how vertices are combined

Example

```
void drawSquare (GLfloat *color)
{
    glColor3fv ( color );
    glBegin(GL_POLYGON);
        glVertex2f ( 0.0, 0.0 );
        glVertex2f ( 1.0, 0.0 );
        glVertex2f ( 1.1, 1.1 );
        glVertex2f ( 0.0, 1.0 );
    glEnd();
}
```

Primitives and Attributes

- **Draw what…**
  - Geometric primitives
    - points, lines and polygons
- **How to draw…**
  - Attributes
    - colors, lighting, shading, texturing, etc.

Attributes

- An attribute is any property that determines how a geometric primitives is to be rendered

- Each time, OpenGL processes a vertex, it uses data stored in its internal attribute tables to determine how the vertex should be transformed, rendered or any of OpenGL's other modes

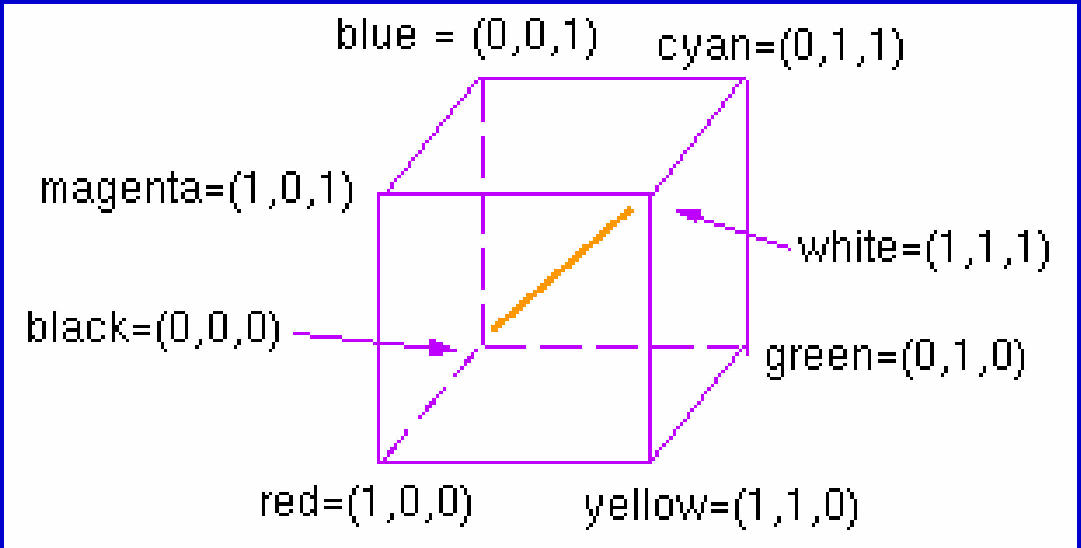
Example

```
glPointSize(3.0);
glShadeModel(GL_SMOOTH);
glBegin(GL_LINE);
    glColor4f(1.0, 1.0, 1.0, 1.0);
    glVertex2f(5.0, 5.0);
    glColor3f(0.0, 1.0, 0.0);
    glVertex2f(25.0, 5.0);
glEnd();
```

OpenGL Color

- There are two color models in OpenGL
  - RGB Color (True Color)
  - Indexed Color (Color map)
- The type of window color model is requested from the windowing system. OpenGL has no command to control

Color Cube



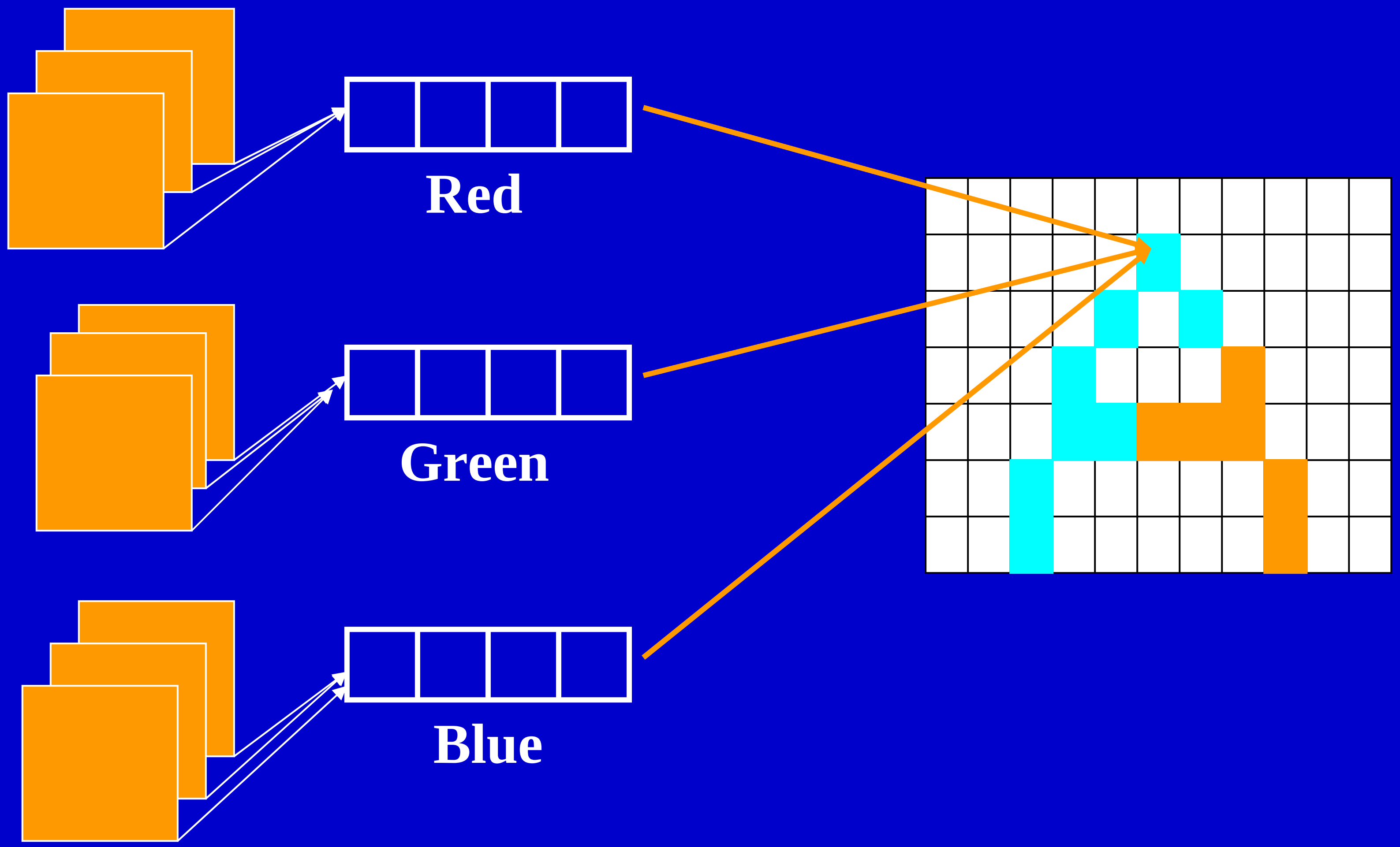
Three color theory

$$C = T_1 * \mathbf{R} + T_2 * \mathbf{G} + T_3 * \mathbf{B}$$

RGB Color

- R, G, B components are stored for each pixel
- With RGB mode, each pixel's color is independent of each other

RGB Color



How Many Colors?

Color number = $2^{\text{color\_depth}}$

For example:

4-bit color

2^4 = 16 colors

8-bit color

2^8 = 256 colors

24-bit color

2^{24}= 16.77 million colors

How Much Memory?

Buffer size = width * height *color depth

For example:

If width = 640, height = 480, color depth = 24 bits

Buffer size = 640 * 480 * 24 = 921,600 bytes

If width = 640, height = 480, color depth = 32 bits

Buffer size = 640 * 480 * 32 = 1,228,800 bytes

Alpha Component

Alpha value

A value indicating the pixels opacity

Zero usually represents totally transparent and the maximum value represents completely opaque

Alpha buffer

Hold the alpha value for every pixel

Alpha values are commonly represented in 8 bits, in which case transparent to opaque ranges from 0 to 255

RGB Color Commands

- **glColor*(…)**

  specifies vertex colors

- **glClearColor(r, g, b, a)**

  sets current color for cleaning color buffer

- **glutInitDisplayMode(mode)**

  specify either an RGBA window (GLUT_RGBA), or a color indexed window (`GLUT_INDEX`)

Example

```
glutInitDisplayMode (GLUT_RGBA);
glClearColor(1.0, 1.0, 1.0, 1.0);
void drawLine (GLfloat *color)
{
    glColor3fv ( color );
    glBegin(GL_LINE);
        glVertex2f ( 0.0, 0.0 );
        glVertex2f ( 1.0, 0.0 );
    glEnd();
}
```

Indexed Color

- Historically, color-index mode was important because it required less memory
- Use Color-map (lookup table)
- With color-index mode, each pixel with same index stored in its bit-planes shares the same color-map location

Color Lookup Table

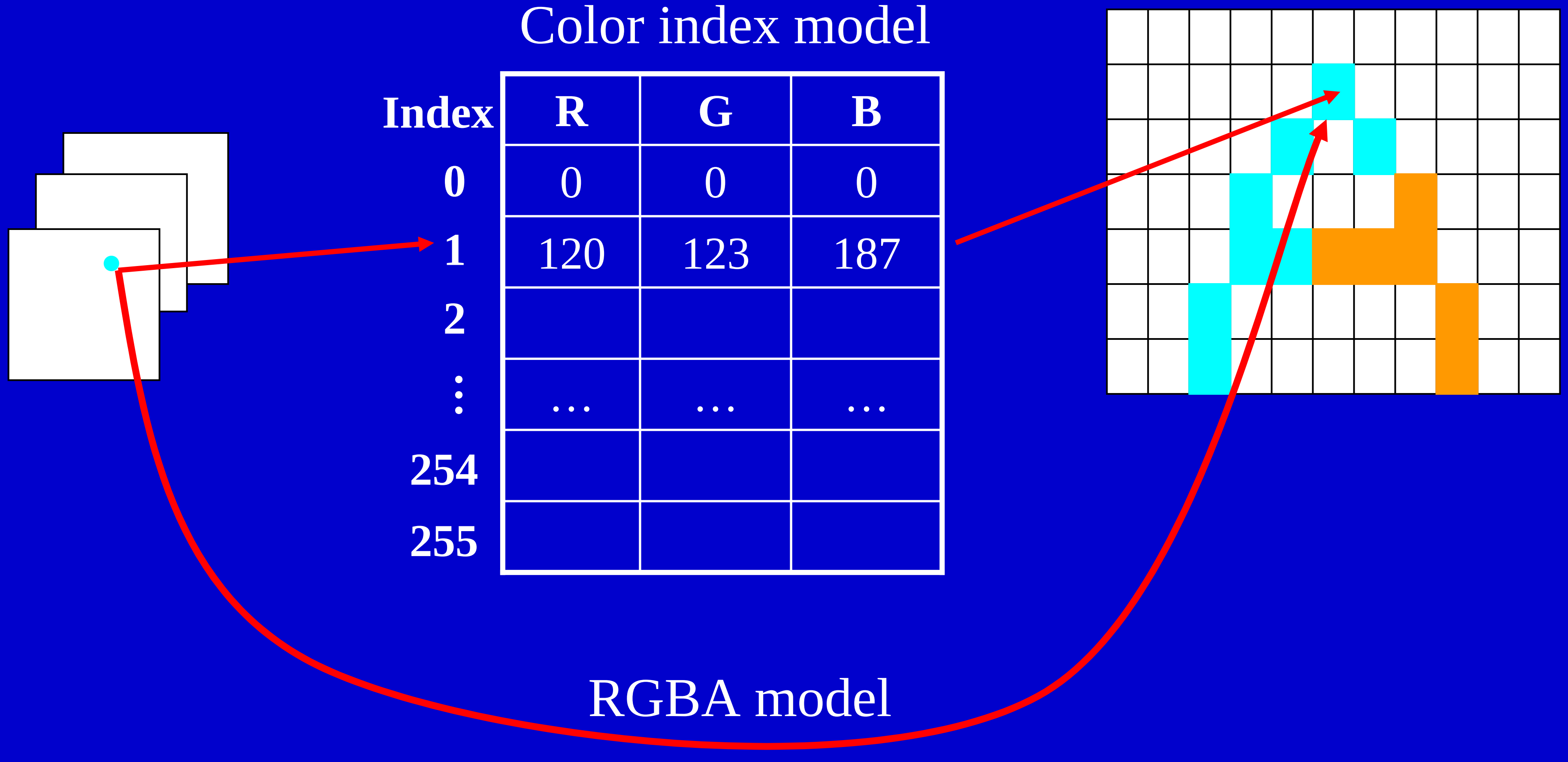
Index	Red	Green	Blue
0	0	0	0
1	120	123	187
⋮			
253	. . .	. . .	. . .
254			
255			

8 bits 8 bits 8 bits

RGBA vs. Color Index Mode

Color index model

Index	R	G	B
0	0	0	0
1	120	123	187
2			
⋮	…	…	…
254			
255			



RGBA model

Color Index Commands

- **glIndex*(…)**

  specifies vertex colors

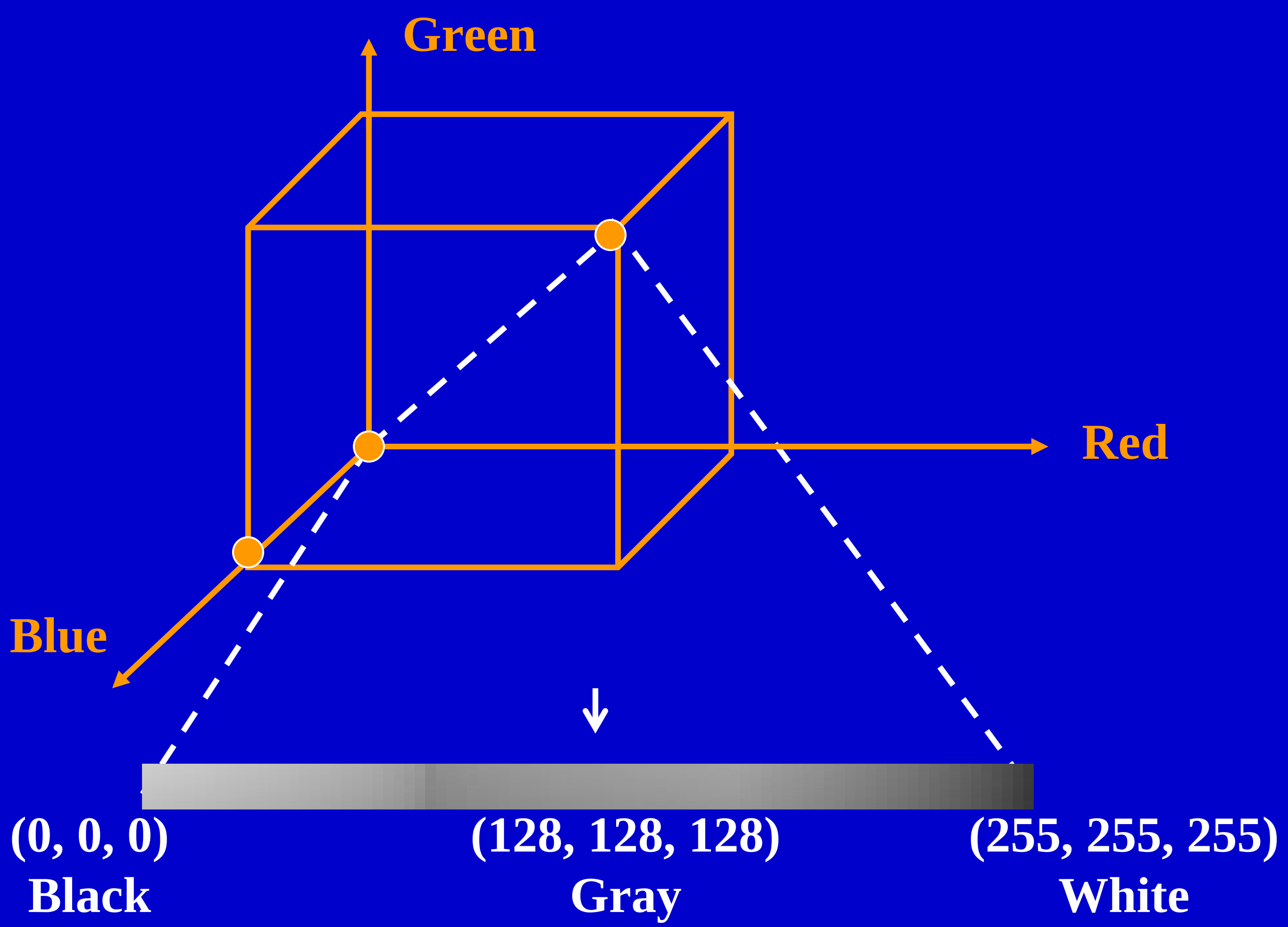
- **glClearIndex(Glfloat index)**

  sets current color for cleaning color buffer.

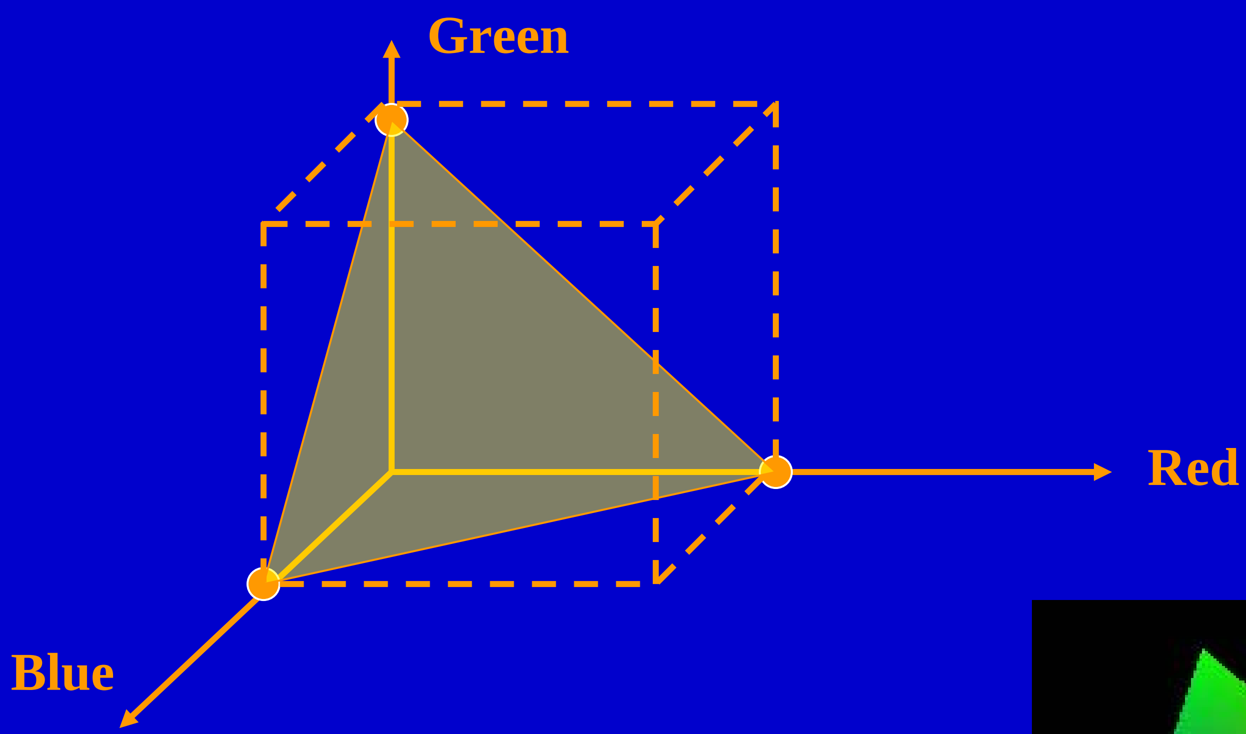
- **glutSetColor(int color, GLfloat r, GLfloat g, GLfloat b)**

  sets the entries in a color table for window.

Shading Model

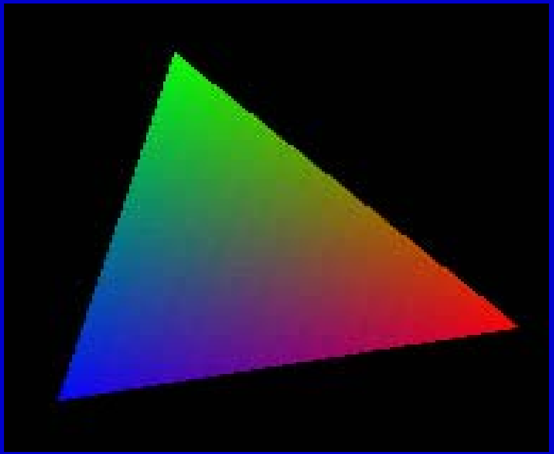


Shading Model



A triangle in RGB color space

Smooth shading in OpenGL



Shading Model Commands

- **glShadeModel(mode)**

  set the shading mode. The mode parameter can be GL_SMOOTH (the default) or GL_FLAT.

Flat shading: the color of one particular vertex of an independent primitive is duplicated across all the primitive's vertices to render that primitive.

Smooth shading: the color at each vertex is treated individually. The colors of interior pixels are interpolated.

OpenGL’s State Machine

In OpenGL, all rendering attributes are encapsulated in the OpenGL *State*

- rendering styles
- color
- shading
- lighting
- texture mapping

OpenGL’s State Management

- **Setting Vertex Attributes**
  - glPointSize(...) -- point size
  - glLineWidth(…) -- line width
  - glColor*(…) -- color
  - glNormal*(…) -- normal
  - glTexCoord*(…) -- texturing

Attributes

- **Controlling State Functions**
  - **glEnable(…)**
  - **glDisable(…)**

Controlling State Functions

- **OpenGL has many states and state variables can be changed to control rendering**

Ex.

- GL_LIGHTING
- GL_DEPTH_TEST
- GL_SHADE_MODEL
- GL_LINE_STIPPLE

……

Controlling State Functions

- **By default, most of the states are initially inactive. These states can be turn on/off by using:**

- **glEnable (Glenum state)**

  turn on a state

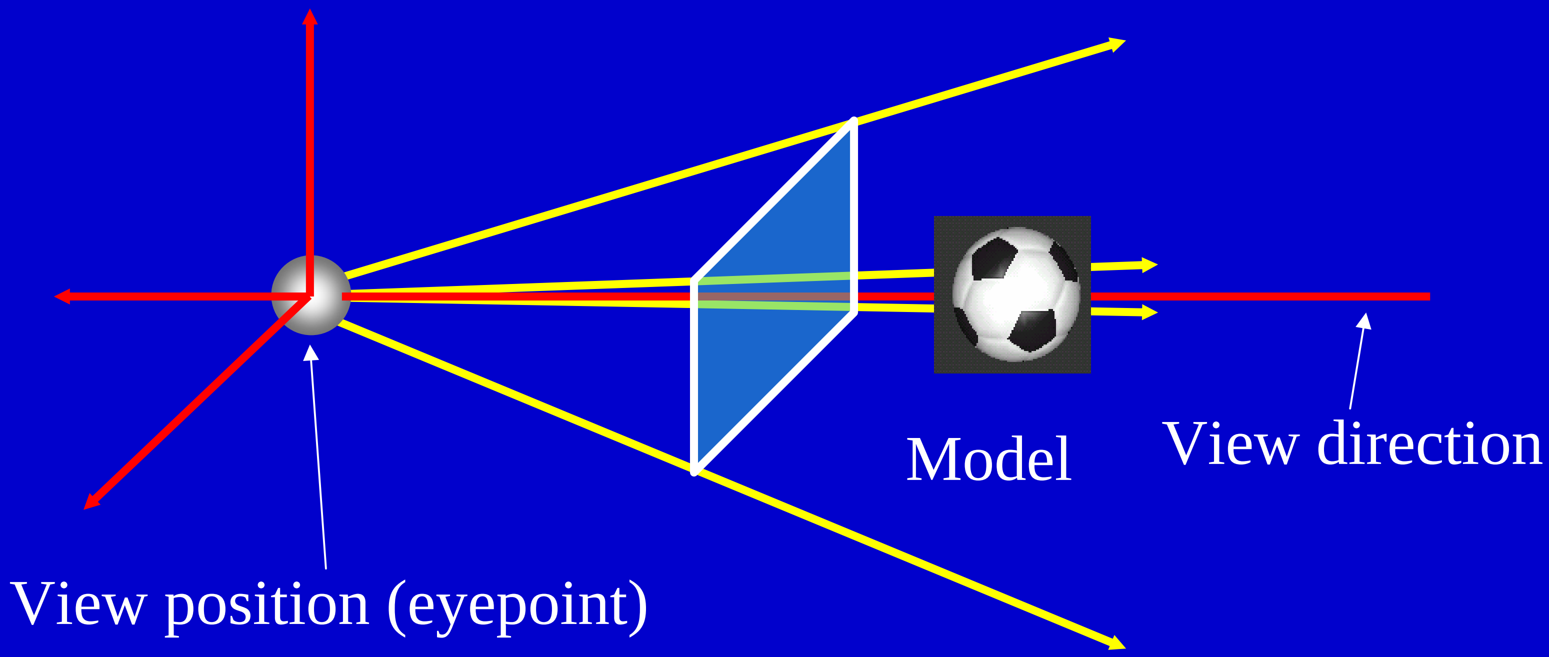
- **glDisable (Glenum state)**

  turn off a state

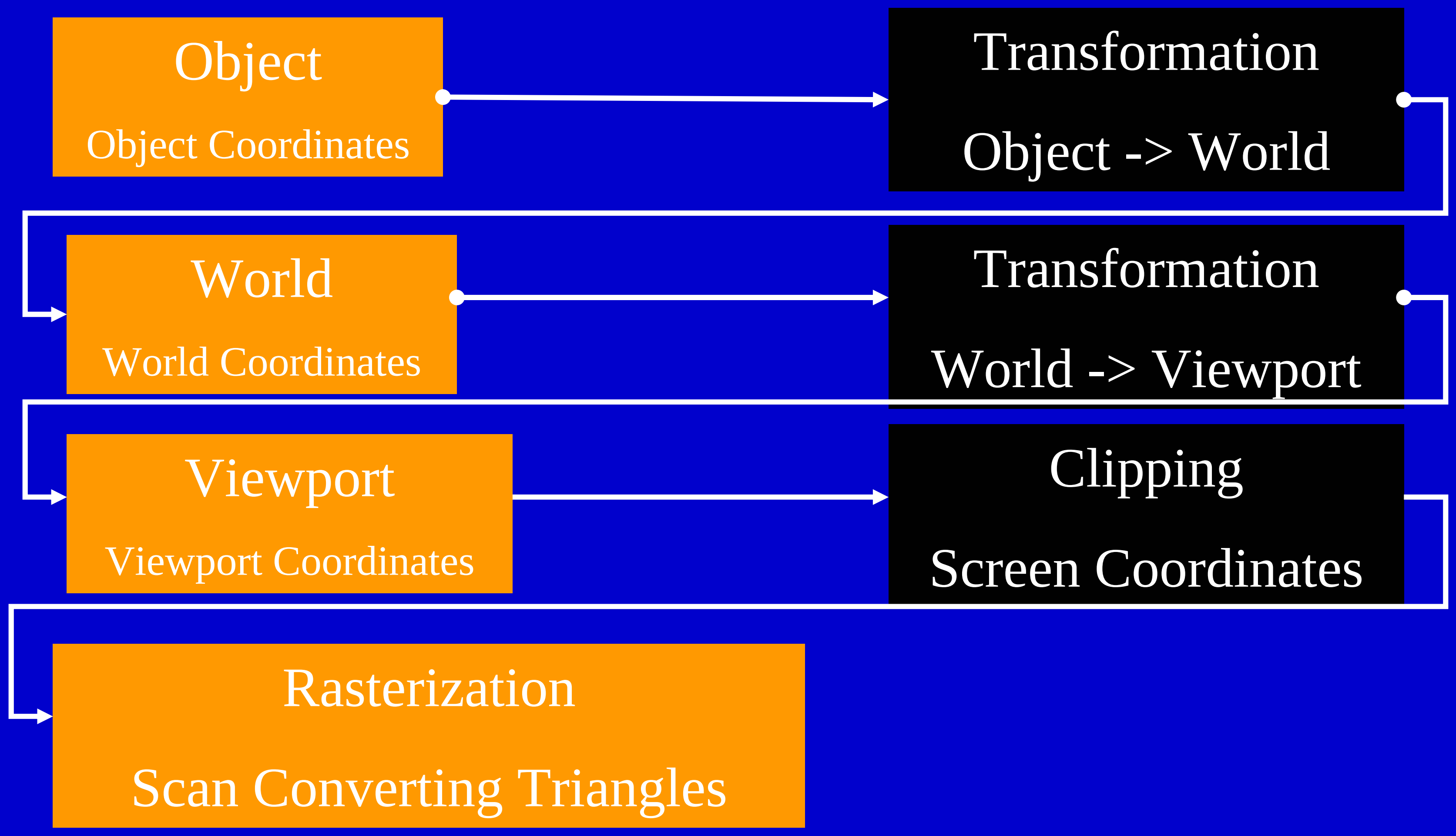
Example

```
glEnable(GL_LIGHTING);
glShadeModel(GL_SMOOTH);
glBegin(GL_LINE);
    glColor3f(1.0, 1.0, 1.0);
    glVertex2f(5.0, 5.0);
    glColor3f(0.0, 1.0, 0.0);
    glVertex2f(25.0, 5.0);
glEnd();
glDisable(GL_LIGHTING);
```

OpenGL Transformations



Graphics Pipeline



Camera Analogy

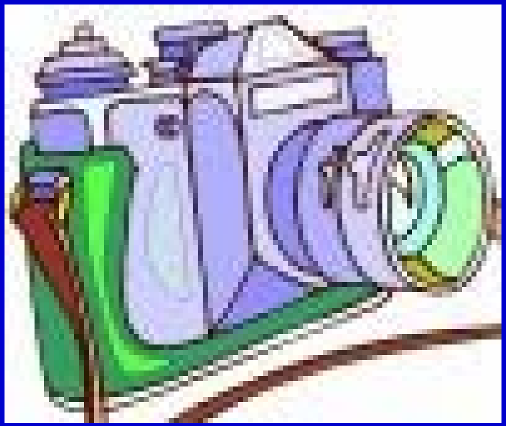
The graphics transformation process is analogous to taking a photograph with a camera

- Position camera

- Place objects

- Adjust camera

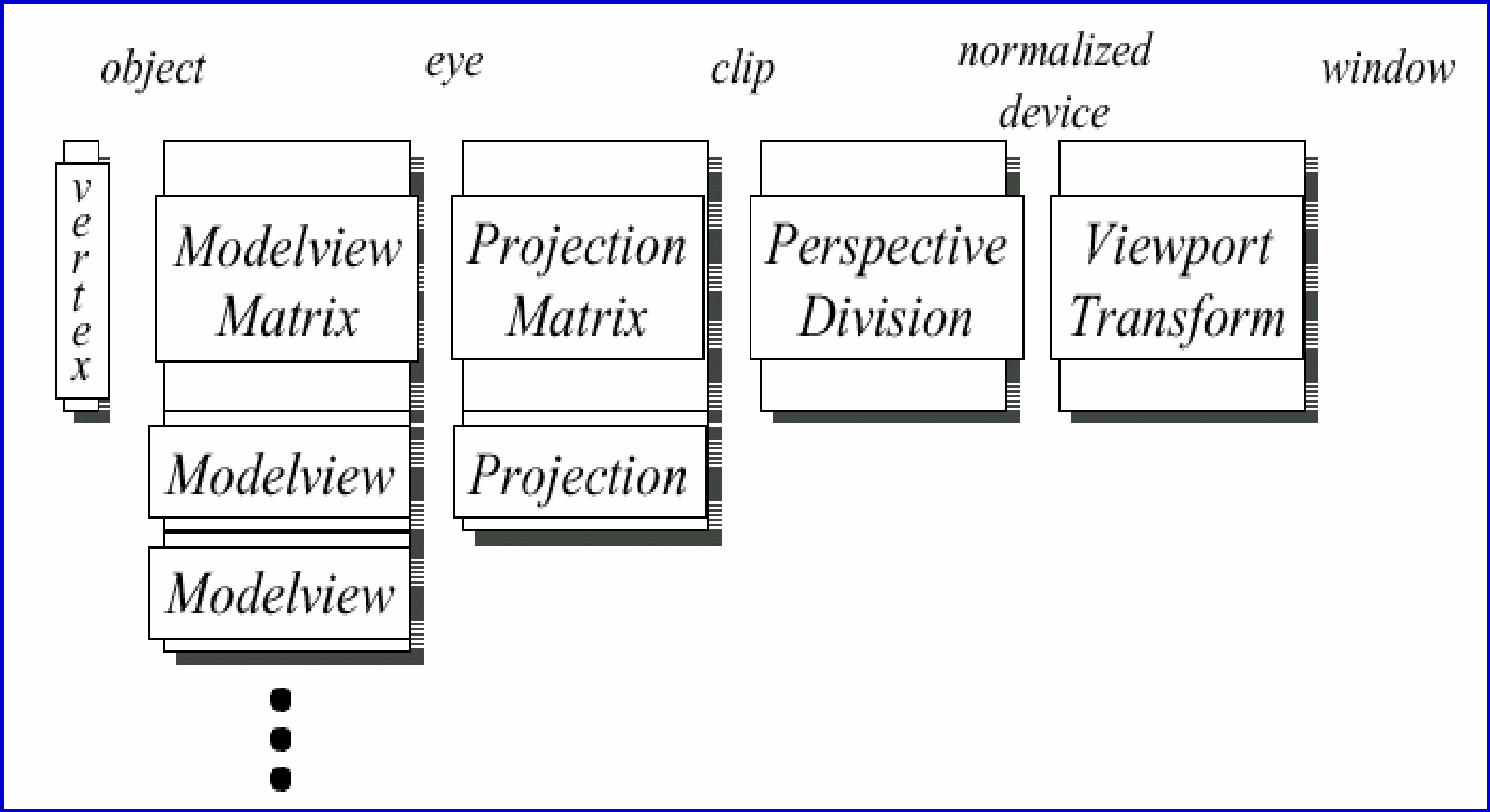
- Produce photograph



Transformations and Camera Analogy

- **Viewing transformation**
  - Positioning and aiming camera in the world.
- **Modeling transformation**
  - Positioning and moving the model.
- **Projection transformation**
  - Adjusting the lens of the camera.
- **Viewport transformation**
  - Enlarging or reducing the physical photograph.

OpenGL Transformation Pipeline



Transformations in OpenGL

- Transformations are specified by *matrix operations*. Desired transformation can be obtained by a sequence of simple transformations that can be concatenated together.
- Transformation matrix is usually represented by 4x4 matrix (homogeneous coordinates).
- Provides *matrix stacks* for each type of supported matrix to store matrices.

Programming Transformations

- **In OpenGL, the transformation matrices are part of the state, they must be defined *prior to* any vertices to which they are to apply.**
- **In modeling, we often have objects specified in their own coordinate systems and must use transformations to bring the objects into the scene.**
- **OpenGL provides *matrix stacks* for each type of supported matrix (model-view, projection, texture) to store matrices.**

Steps in Programming

- Define matrices:
  - Viewing/modeling, projection, viewport …
- Manage the matrices
  - Including matrix stack
- Composite transformations

Transformation Matrix Operation

- Current Transformation Matrix (*CTM*)
  - The matrix that is applied to any vertex that is defined subsequent to its setting.
- If change the CTM, we change the *state* of the system.
- CTM is a 4 x 4 matrix that can be altered by a set of functions.

Current Transformation Matrix

The CTM can be set/reset/modify (by post-multiplication) by a matrix

Ex:

C <= **M** // set to matrix **M**

C <= **CT** // post-multiply **by T**

C <= **CS** // post-multiply **by S**

C <= **CR** // post-multiply **by R**

Current Transformation Matrix

- Each transformation actually creates a new matrix that multiplies the CTM; the result, which becomes the new CTM.
- CTM contains the cumulative product of multiplying transformation matrices.

Ex:

If **C** <= **M**; **C** <= **CT**; **C** <= **CR**; **C** <= **CS**

Then **C** = **M T R S**

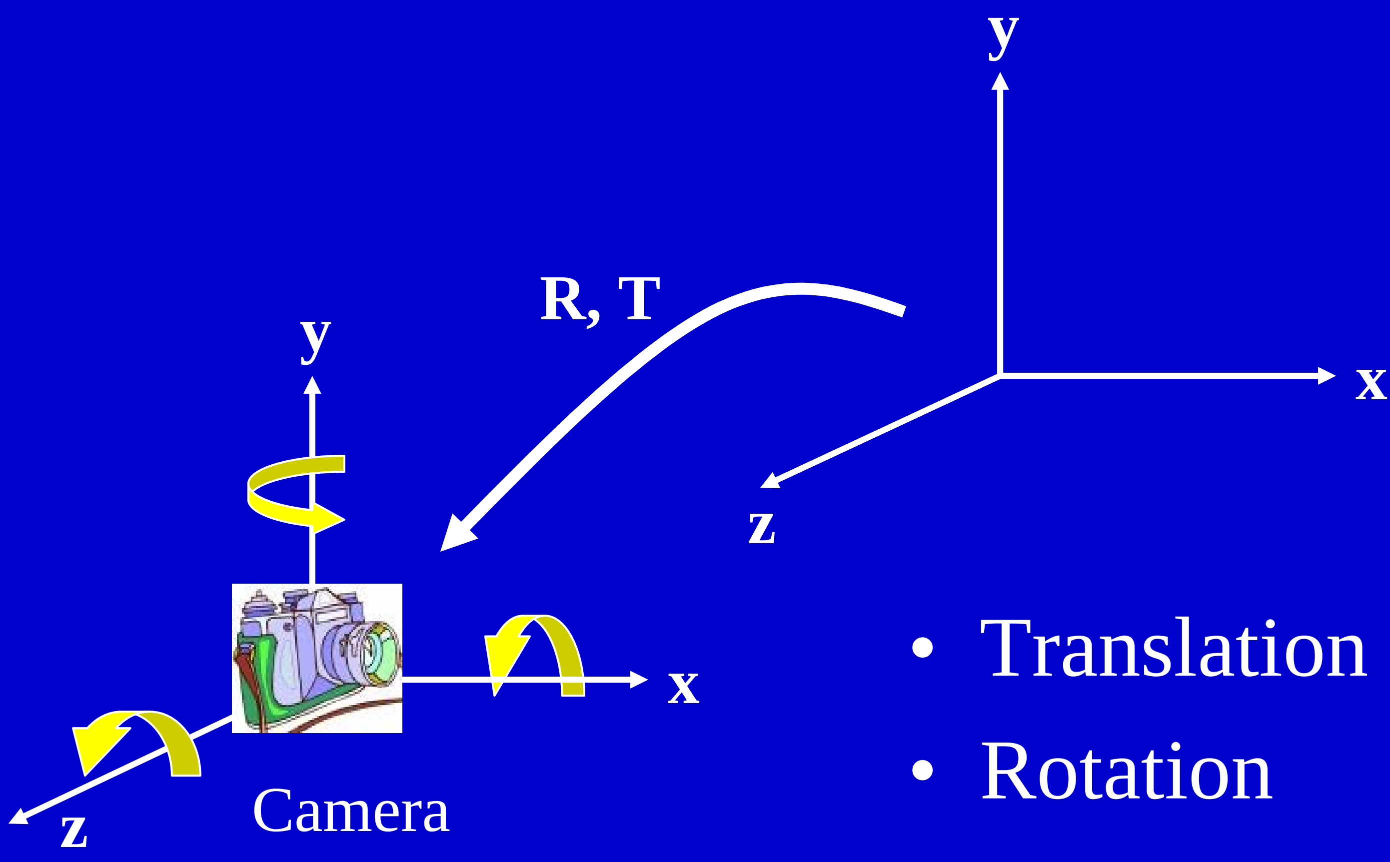
Viewing-Modeling Transformation

- If given an object, and I want to render it from a viewpoint, what information do I have to have?

  – Viewing position
  – Which way I am looking at
  – Which way is “up”

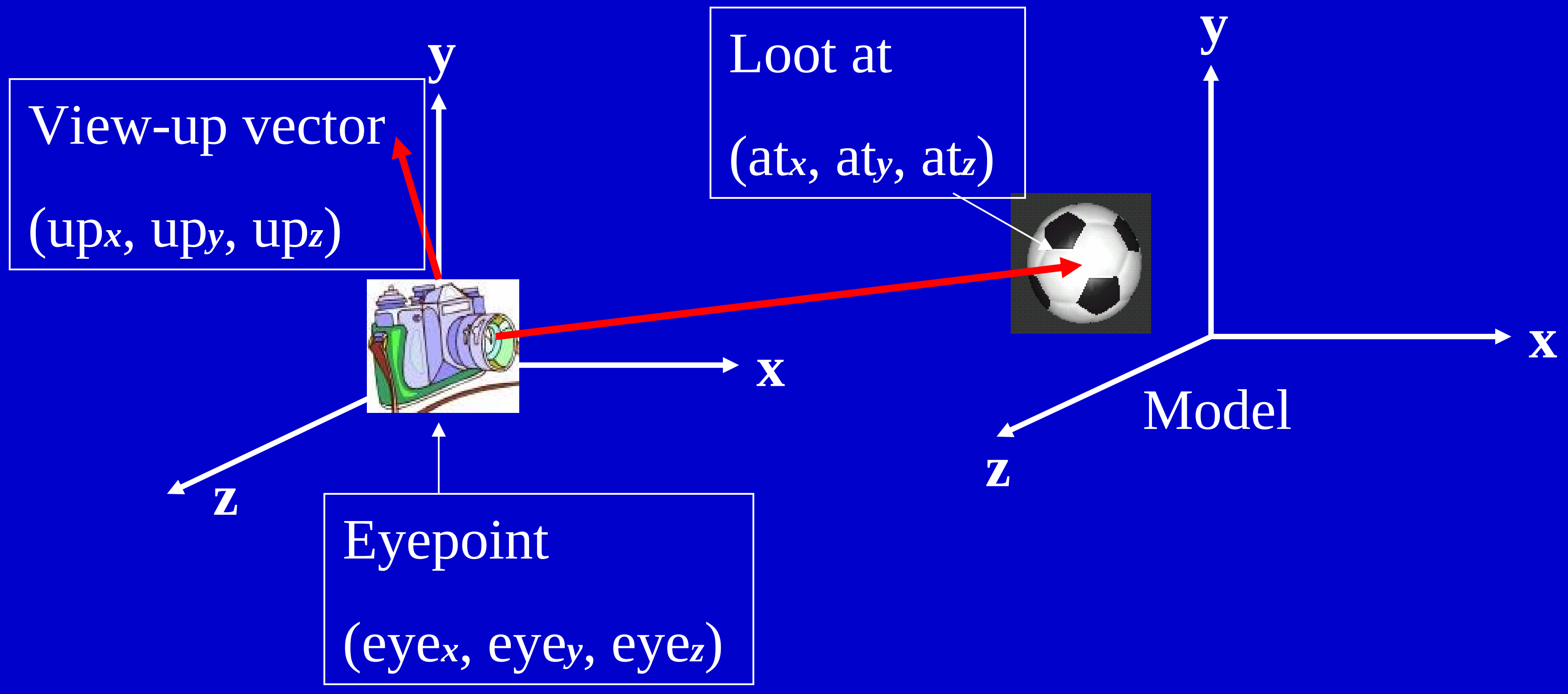
  …..

Viewing Position



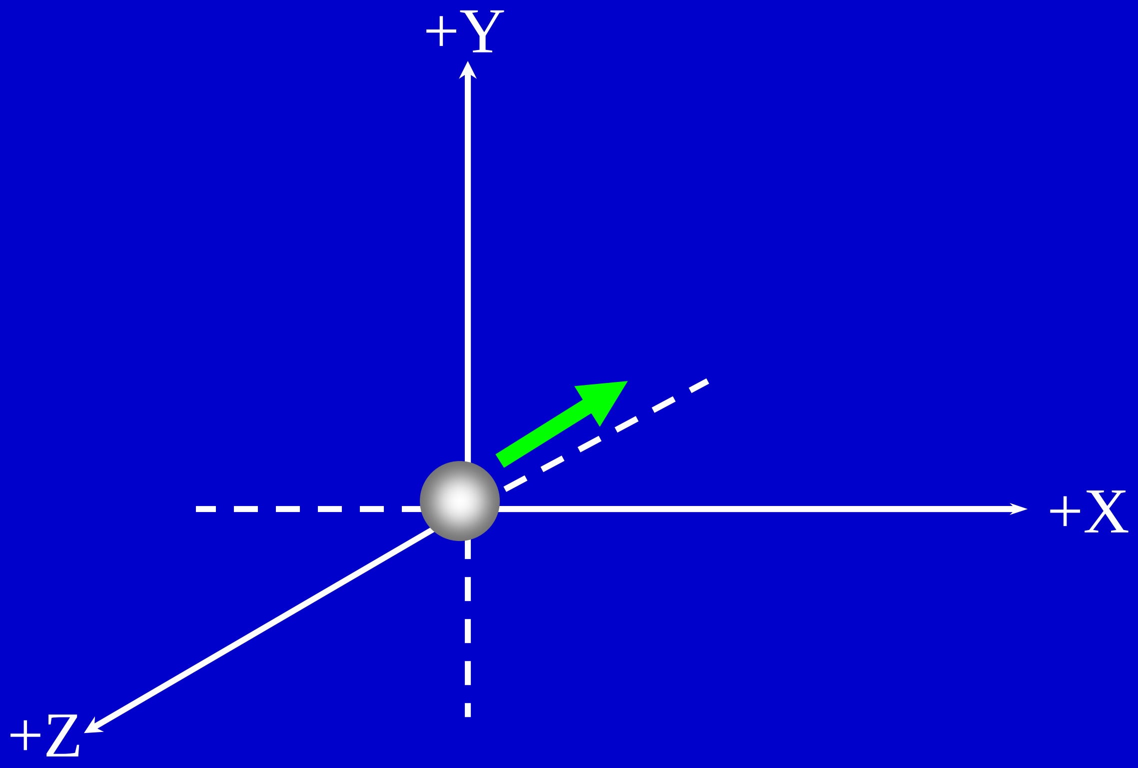
- Translation
- Rotation

Where I am and Looking at



Define Coordinate System

In the default position, the camera is at the origin, looking down the **negative** z-axis



If we use OpenGL

- **Look-At Function**

gluLookAt (eye$_x$, eye$_y$, eye$_z$, at$_x$, at$_y$, at$_z$, up$_x$, up$_y$, up$_z$ **)**

Define a viewing matrix and multiplies it to the right of the current matrix.

Ways to Specify Transformations

- In OpenGL, we usually have two styles of specifying transformations:
  - Specify matrices (**glLoadMatrix, glMultMatrix**)
  - Specify operations (**glRotate, glTranslate**)

Specifying Matrix

- Specify current matrix mode
- Modify current matrix
- Load current matrix
- Multiple current matrix

Specifying Matrix (1)

- **Specify current matrix mode**

glMatrixMode (*mode*)

Specified what transformation matrix is modified.

mode:

GL_MODELVIEW

GL_PROJECTION

Specifying Matrix (2)

- **Modify current matrix**

glLoadMatrix{fd} (Type ****m* **)**

Set the 16 values of current matrix to those specified by ***m***.

Note: ***m*** is the 1D array of 16 elements arranged by the *columns* of the desired matrix

Specifying Matrix (3)

- **Modify current matrix**

glLoadIdentity (void)

Set the currently modifiable matrix to the 4x4 identity matrix.

Specifying Matrix (4)

- **Modify current matrix**

glMultMatrix{fd} (Type ****m* **)**

Multiple the matrix specified by the 16 values pointed by ***m*** by the current matrix, and stores the result as current matrix.

Note: ***m*** is the 1D array of 16 elements arranged by the *columns* of the desired matrix

Specifying Operations

- Three OpenGL operation routines for modeling transformations:
  - Translation
  - Scale
  - Rotation

Recall

- **Three elementary 3D transformations**

Translation:

$$T(d_x, d_y, d_z) = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scale:

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

Recall

Rotation $R_x(\theta)$

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

Rotation $R_y(\theta)$

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

Rotation $R_z(\theta)$

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

Specifying Operations (1)

- **Translation**

glTranslate {fd} (TYPE *x*, TYPE *y*, TYPE *z*)

Multiplies the current matrix by a matrix that translates an object by the given ***x***, ***y***, ***z***.

Specifying Operations (2)

- **Scale**

glScale {fd} (TYPE *x*, TYPE *y*, TYPE *z*)

Multiplies the current matrix by a matrix that scales an object by the given ***x***, ***y***, ***z***.

Specifying Operations (3)

- **Rotate**

glRotate {fd} (TPE *angle*, TYPE *x*, TYPE *y*, TYPE *z*)

Multiplies the current matrix by a matrix that rotates an object in a counterclockwise direction about the ray from origin through the point by the given ***x***, ***y***, ***z***. The ***angle*** parameter specifies the angle of rotation in *degree*.

Example

Let’s examine an example:

- **Rotation about an arbitrary point**

Question:

Rotate a object for a 45.0-degree about the line through the origin and the point (1.0, 2.0, 3.0) with a fixed point of (4.0, 5.0, 6.0).

Rotation About an Arbitrary Point

1. Translate object through vector $-\boldsymbol{V}$.

   T(-4.0, -5.0, -6.0)

2. Rotate about the origin through angle θ.

   R(45.0)

3. Translate back through vector $\boldsymbol{V}$

   T(4.0, 5.0, 6.0)

$$\boldsymbol{M = T(V)\, R(\theta)\, T(-V)}$$

OpenGL Implementation

```
glMatrixMode (GL_MODEVIEW);
glLoadIdentity ();
glTranslatef (4.0, 5.0, 6.0);
glRotatef (45.0, 1.0, 2.0, 3.0);
glTranslatef (-40.0, -5.0, -6.0);
```

Order of Transformations

- The transformation matrices appear in *reverse* order to that in which the transformations are applied.
- *In OpenGL, the transformation specified most recently is the one applied first.*

Order of Transformations

- **In each step:**

  C <= I

  C <= CT(4.0, 5.0, 6.0)

  C <= CR(45, 1.0, 2.0, 3.0)

  C < = CT(-4.0, -5.0, -6.0)

- **Finally**

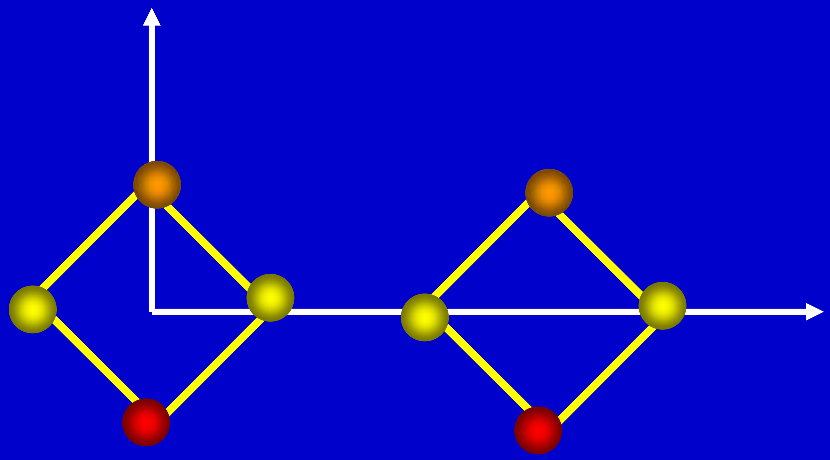
C = T(4.0, 5.0, 6.0) CR(45, 1.0, 2.0, 3.0) CT(-4.0, -5.0, -6.0)

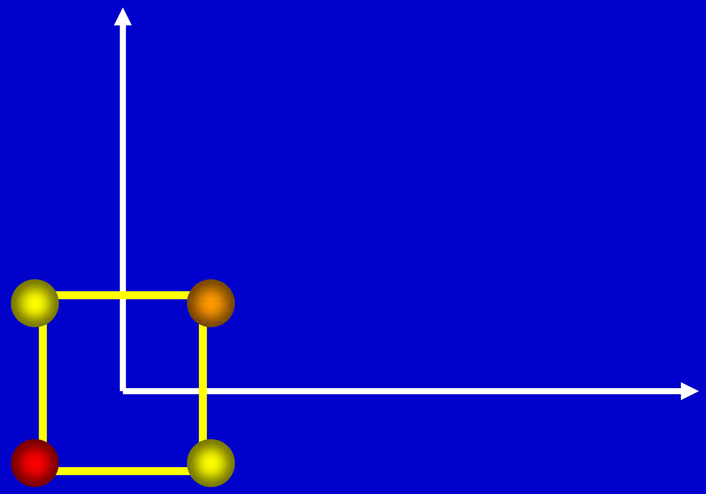
Write it

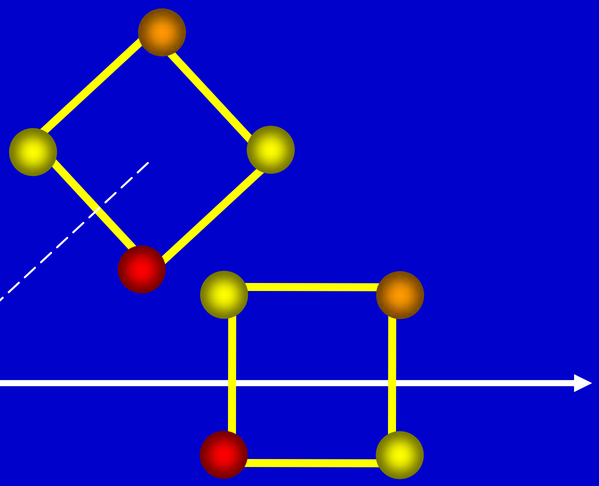
Read it

Matrix Multiplication is Not Commutative

First rotate, then translate =>







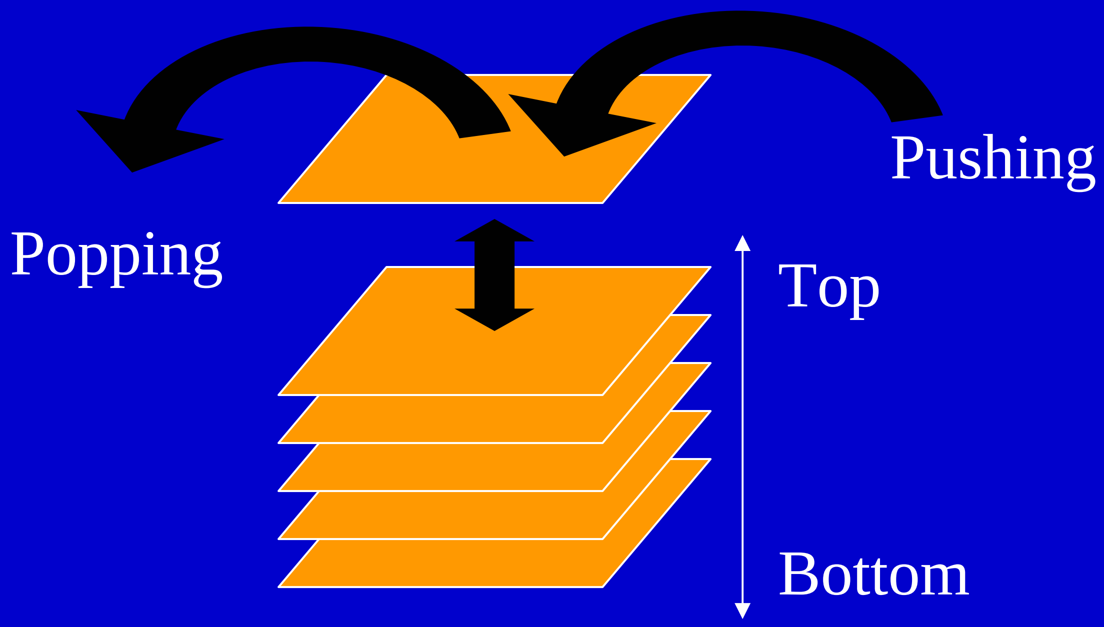
First translate, then rotate =>

Matrix Stacks

- OpenGL uses matrix stacks mechanism to manage transformation hierarchy.
- OpenGL provides matrix stacks for each type of supported matrix to store matrices.
  - Model-view matrix stack
  - Projection matrix stack
  - Texture matrix stack

Matrix Stacks

- Current matrix is always the topmost matrix of the stack
- We manipulate the current matrix is that we actually manipulate the topmost matrix.
- We can control the current matrix by using push and pop operations.



Manipulating Matrix Stacks (1)

- ***Remember where you are***

glPushMatrix (void)

Pushes all matrices in the current stack *down one level*. The topmost matrix is copied, so its contents are duplicated in both the top and second-from-the top matrix.

Note: current stack is determined by **glMatrixModel()**

Manipulating Matrix Stacks (2)

- ***Go back to where you were***

glPopMatrix (void)

Pops the top matrix off the stack, destroying the contents of the popped matrix. What was the second-from-the top matrix becomes the top matrix.

Note: current stack is determined by **glMatrixModel()**

Manipulating Matrix Stacks (3)

- **The depth of matrix stacks are implementation-dependent.**
- **The Modelview matrix stack is guaranteed to be at least 32 matrices deep.**
- **The Projection matrix stack is guaranteed to be at least 2 matrices deep.**

glGetIntegerv (Glenum *pname*, Glint **parms*)

Pname:

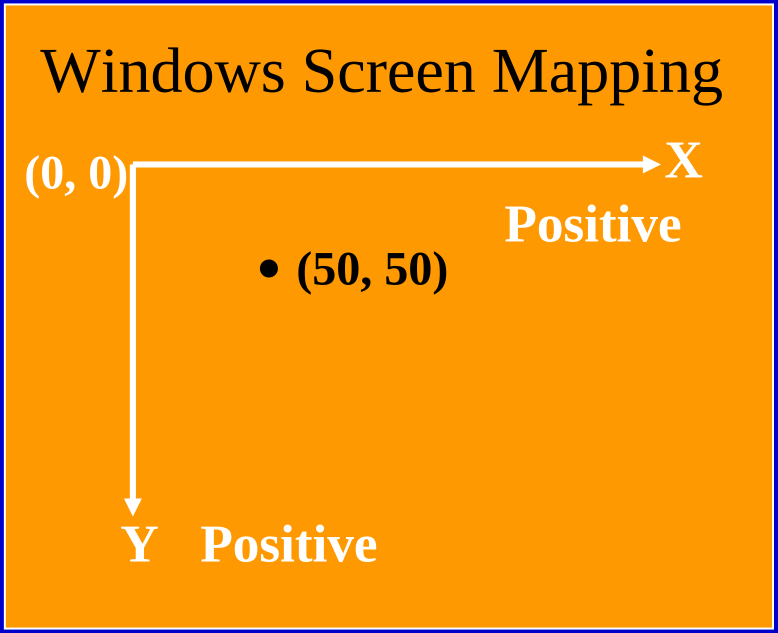
GL_MAX_MODELVIEW_STACT_DEPTH

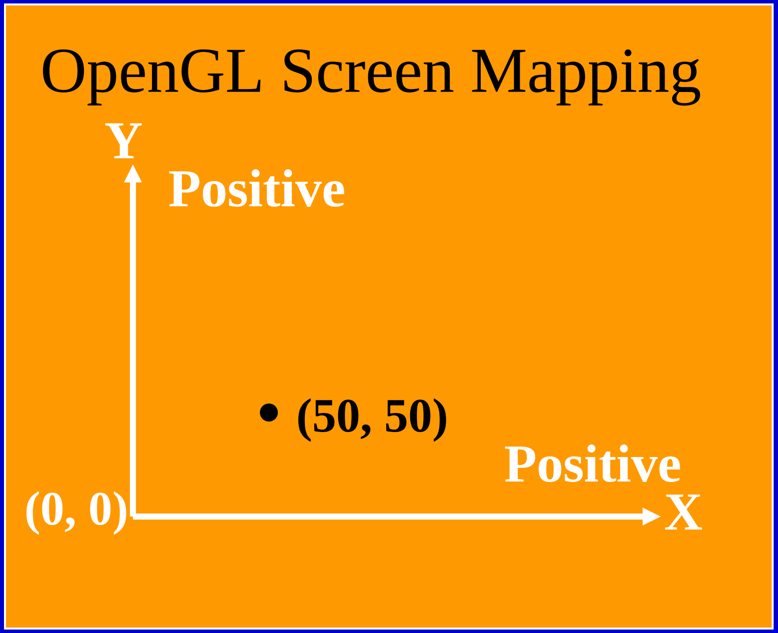
GL_MAX_PROJECTION_STACT_DEPTH

Projection Transformation

- Projection & Viewing Volume
- Projection Transformation
- Viewpoint Transformation

OpenGL and Windows Screen

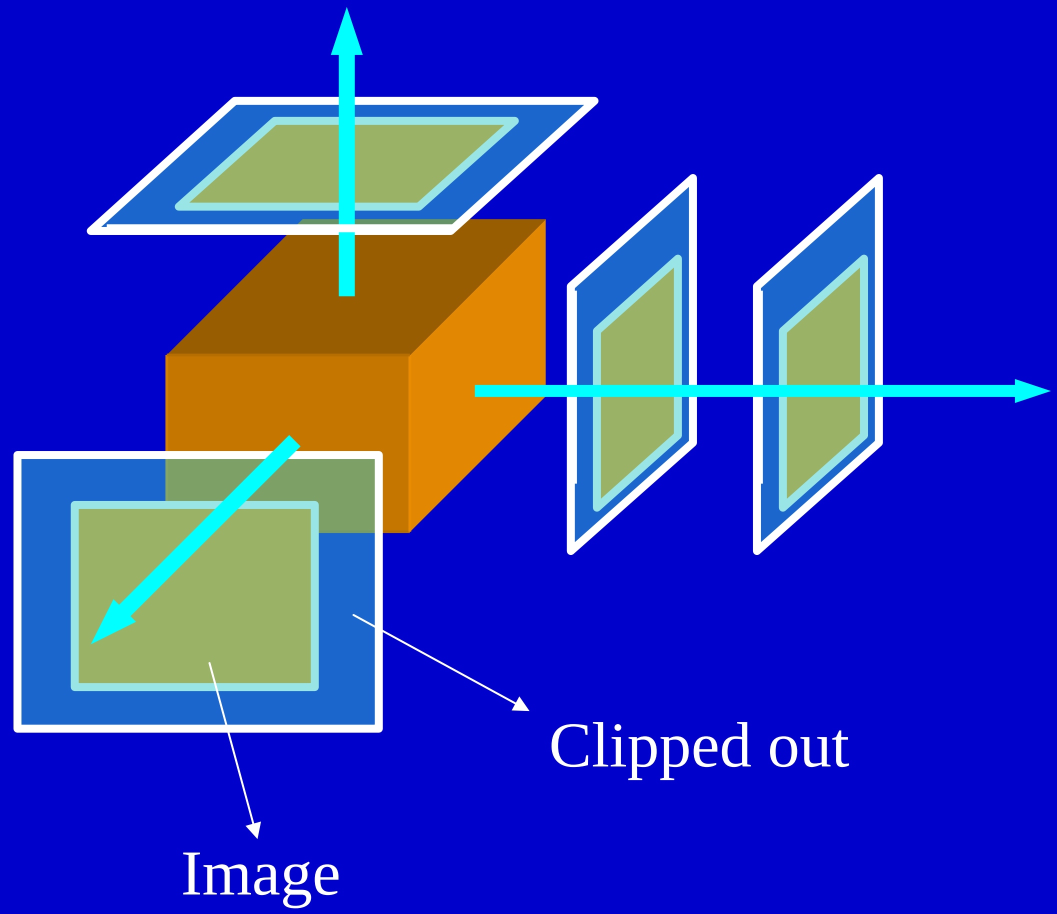




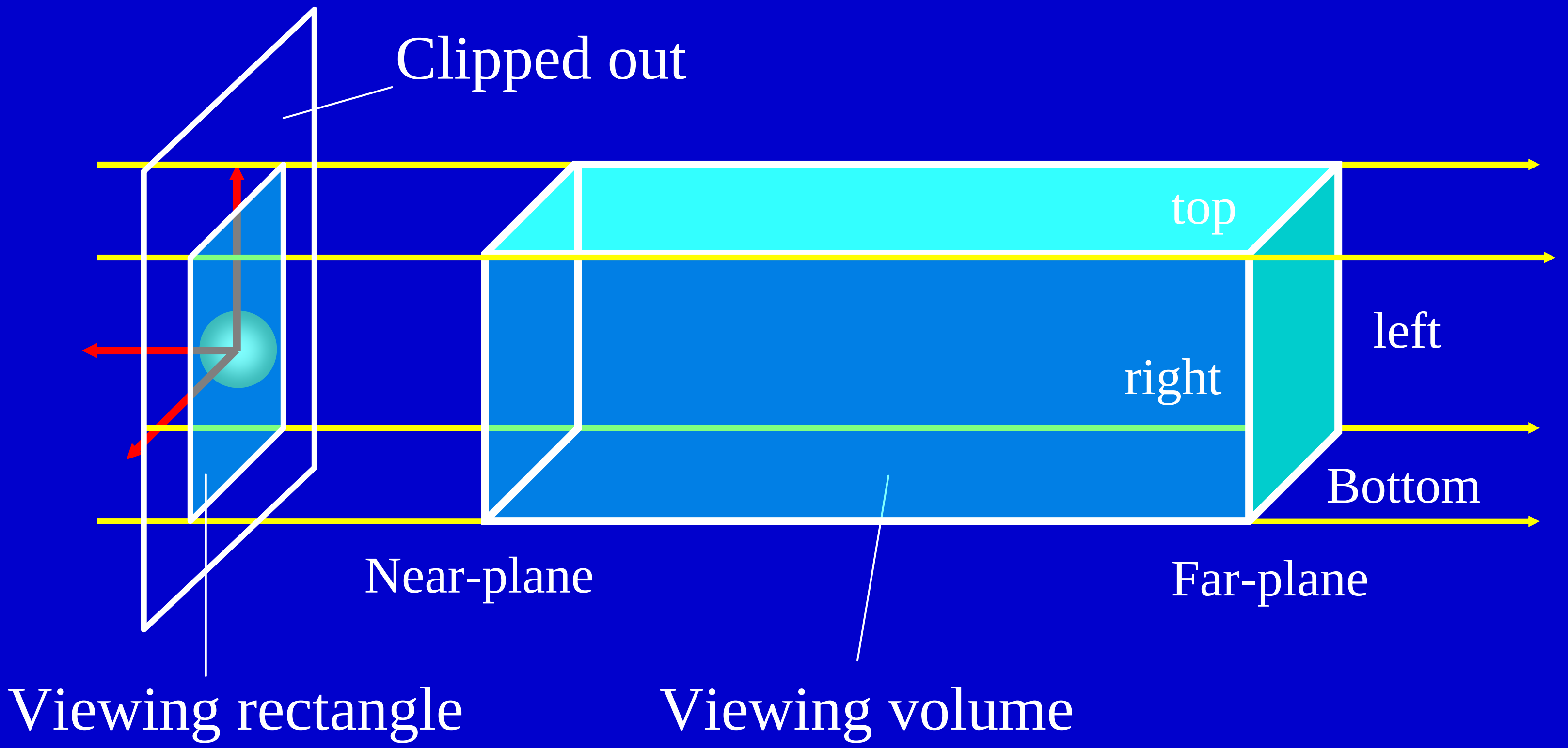
Remember: the Y coordinates of OpenGL screen is the opposite of Windows screen. But same as in the XWindows system.

Orthographic Projection

- **Vertexes of an object are projected towards *infinity***
- **Points projected outside view volume are clipped out**
- **Distance does not change the apparent size of an object**



Orthographic Viewing Volume



Orthographic Projection Commands

- **glOrtho(left, right, bottom, top, zNear, zFar)**

  Creates a matrix for an orthographic viewing volume and multiplies the current matrix by it

- **gluOrtho2D(left, right, bottom, top)**

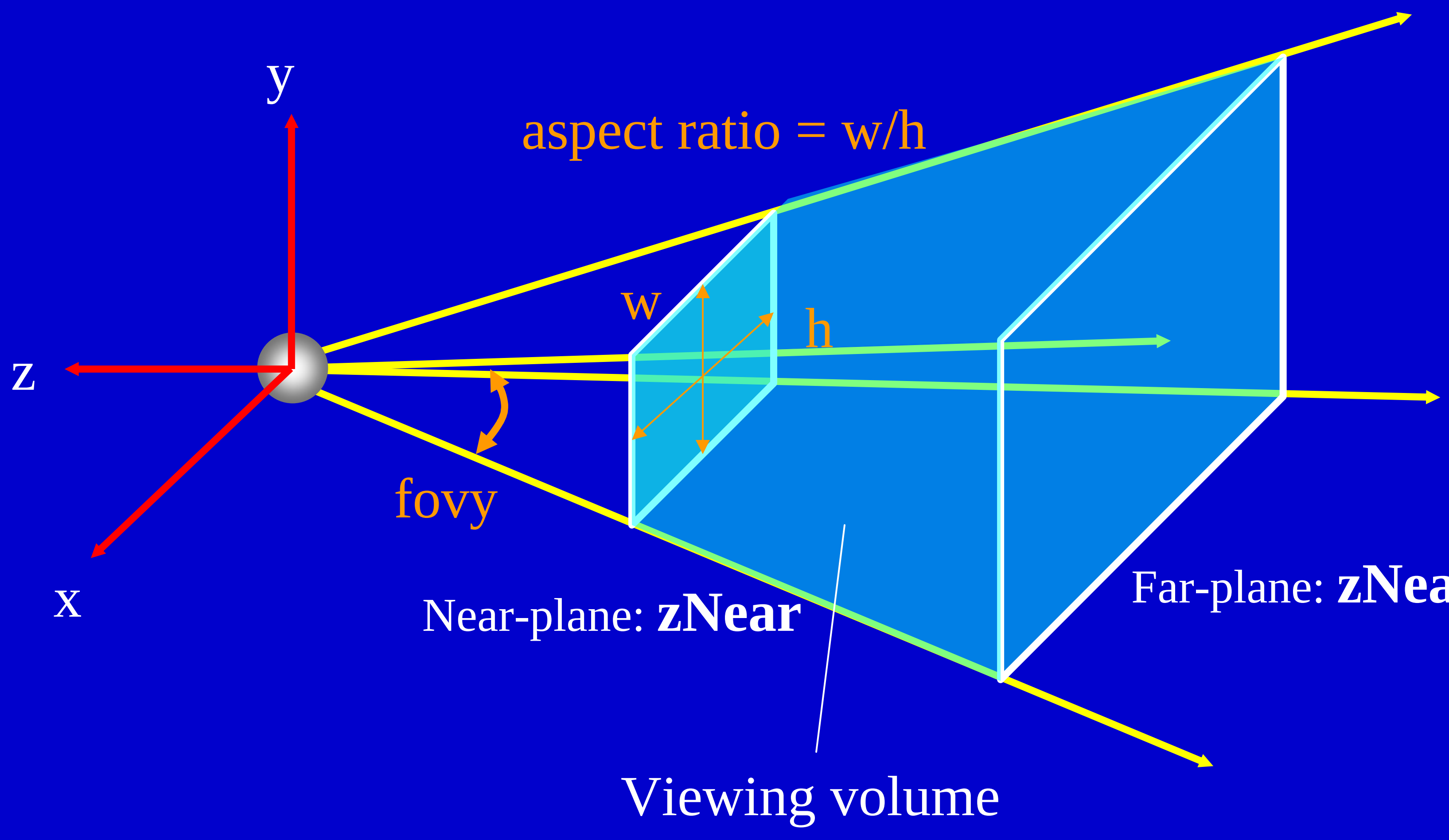
  Creates a matrix for projecting 2D coordinates onto the screen and multiplies the current matrix by it

Orthographic Projection (Example)

Define a 500x500 viewing rectangle with the lower-left corner of the rectangle at the origin of the 2D system

```
glMatrixMode(GL_PROJECTION)
glLoadIdentity();
gluOrtho2D(0.0, 500.0, 0.0, 500.0);
glMatrixMode(GL_MODELVIEW);
```

Perspective Projection Volume



Perspective Projection Commands

glFrustum(left, right, bottom, top, zNear, zFar)

Creates a matrix for a perspective viewing frustum and multiplies the current matrix by it.

Perspective Projection Commands

gluPerspective(fovy, aspect, zNear, zFar)

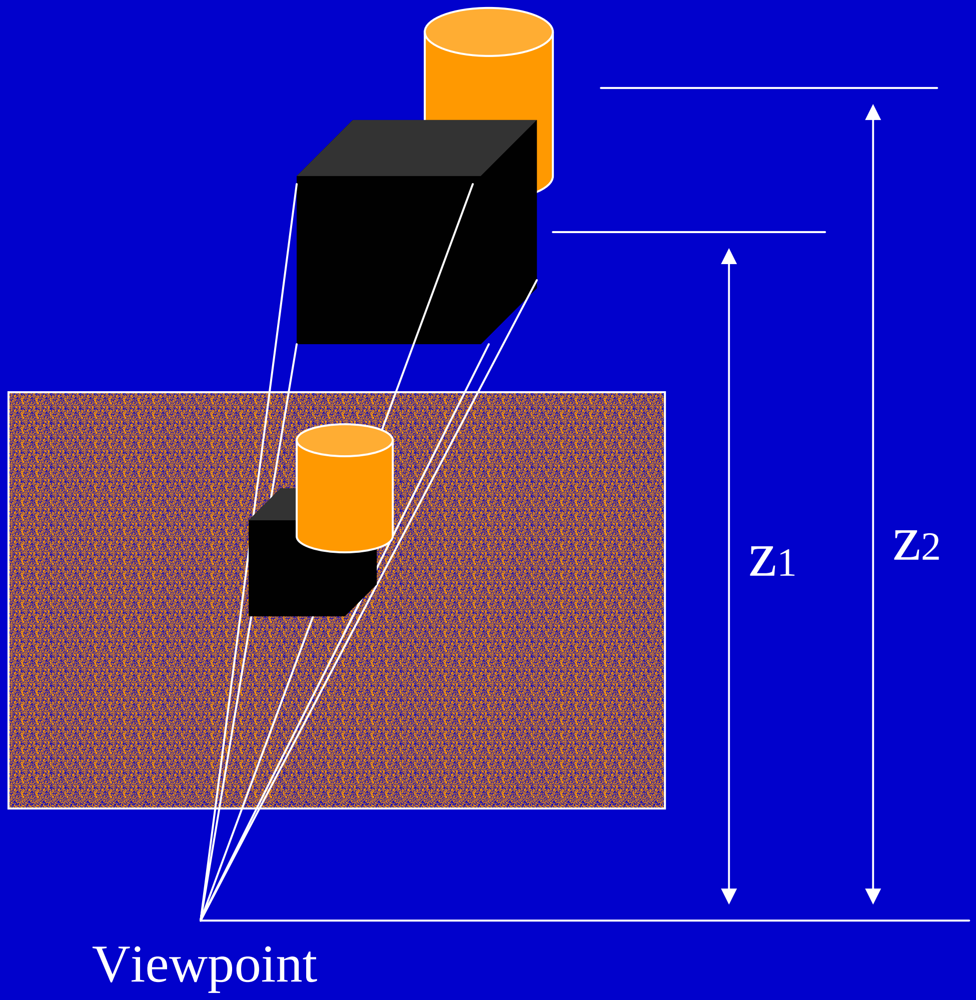
Creates a matrix for an perspective viewing frustum and multiplies the current matrix by it.

Note: ***fovy*** is the **field of view** (fov) between the top and bottom planes of the clipping volume. ***aspect*** is the aspect ratio

Hidden-Surface Removal

z-buffer algorithm

- Image-space check
- The worst-case complexity is proportional to the number of polygons
- Requires a depth or **z buffer** to store the information as polygons are rasterized



Hidden-Surface Removal

glEnable(GL_DEPTH_TEST)

glDisable(GL_DEPTH_TEST)

Enable/disable z (depth) buffer for hidden-surface removal

Remember to Initialize

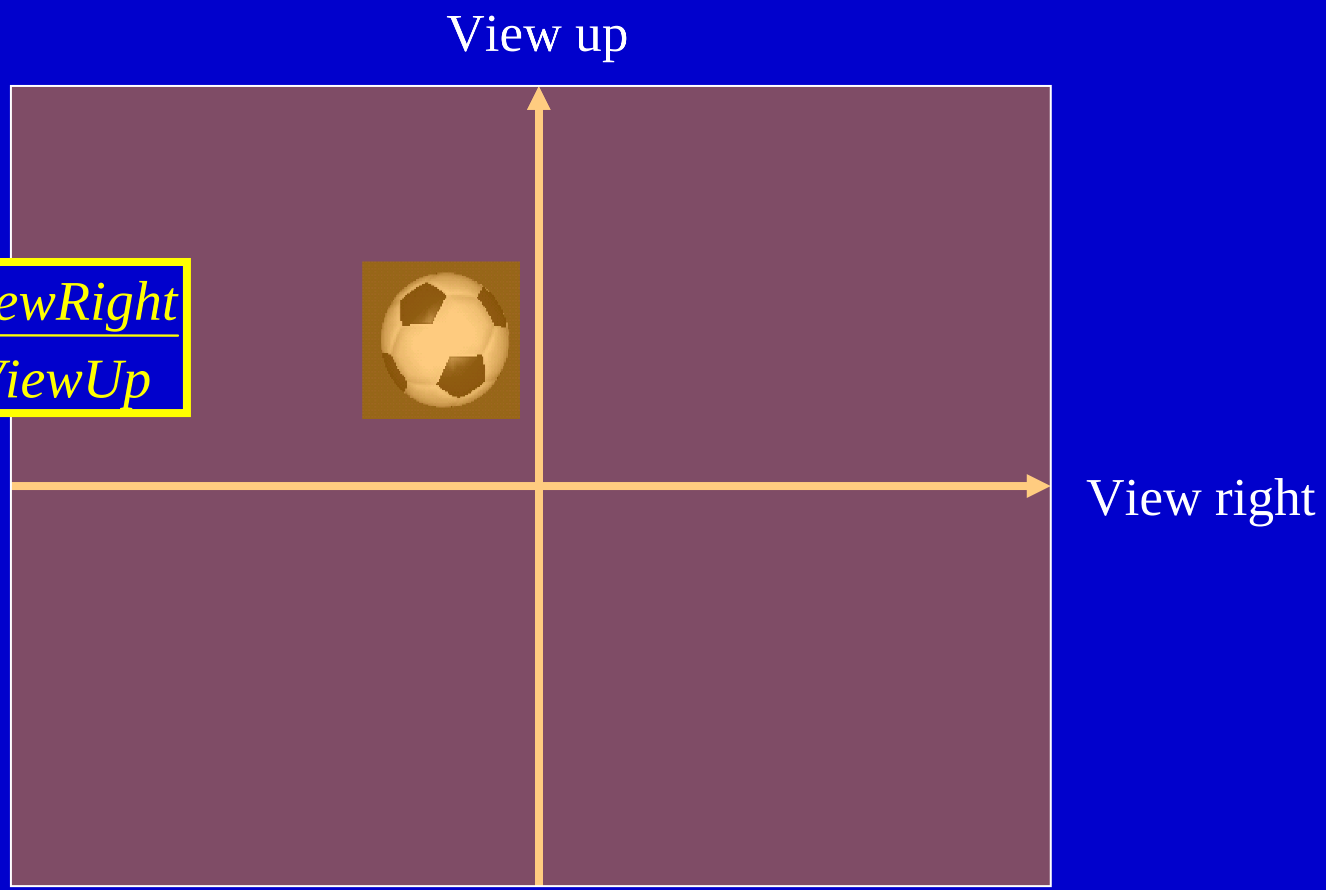
glutInitDisplayMode(GLUT_DOUBLE|GLUT_RBG|GLUT_DEPTH);

You can also clear the depth buffer (as we did for color buffer)

glClear(GL_DEPTH_BUFFER_BIT)

Clear z (depth) buffer

Viewing a 3D world

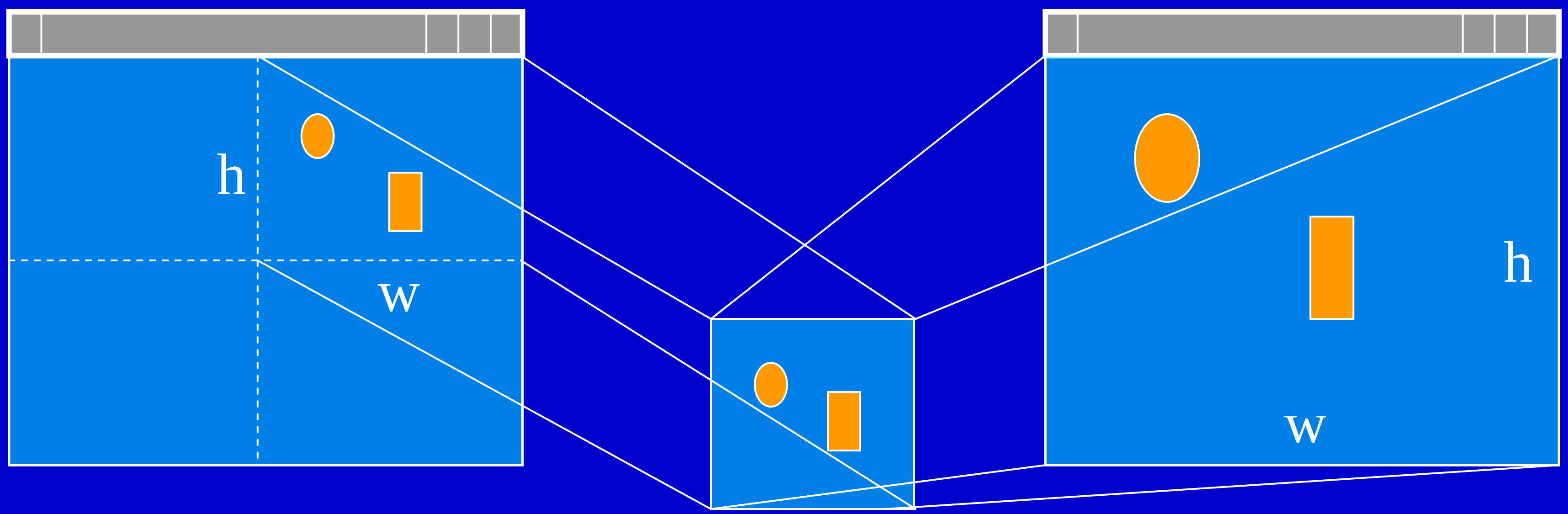


$$\boxed{\text{Aspect Ratio} = \frac{ViewRight}{ViewUp}}$$

Viewpoint

- **Viewpoint**
  - The region within the window that will be used for drawing the clipping area
  - By default, it is set to the entire rectangle of the window that is opened
  - Measured in the window coordinates, which reflect the position of pixels on the screen related to the lower-left corner of the window

Viewpoint Transformation



A viewpoint is defined as half the size of the window

A viewpoint is defined as the same size as the window

Aspect Ratio

- The Aspect Ratio of a rectangle is the ratio of the rectangle's width to its height:

  e.g. **Aspect Ratio = width/height**

- Viewport **aspect ratio** should be same as projection transformation, or resulting image may be distorted.

Viewpoint Commands

- **glViewport(x, y, width, height)**

  Defines a pixel rectangle in the window into which the final image is mapped

  (x, y) specifies the lower-left corner of the viewport

  (width, height) specifies the size of the viewport rectangle

Lighting

- **Point light source -** approximates the light source as a 3D point in space. Light rays emanate in all directions.
- **Distributed light source** - approximates the light source as a 3D object. Light rays usually emanate in specific directions.
- **Spotlights -** characterized by a narrow range of angles through which light is emitted.
- **Ambient light -** provide uniform illumination throughout the environment. It represents the approximate contribution of the light to the general scene, regardless of location of light and object. (Background Light)

Light Model

- **Ambient**

  The combination of light reflections from various surfaces to produce a uniform illumination. Background light.
- **Diffuse**

  Uniform light scattering of light rays on a surface. Proportional to the “amount of light” that hits the surface. Depends on the surface normal and light vector.
- **Sepecular**

  Light that gets reflected. Depends on the light ray, the viewing angle, and the surface normal.

Light Model

- Light at a pixel from a light = Ambient +
  Diffuse +
  Specular

$$\boldsymbol{I}_{\text{light}} = \boldsymbol{I}_{\text{ambient}} + \boldsymbol{I}_{\text{diffuse}} + \boldsymbol{I}_{\text{specular}}$$

$$I_{light} = k_a L_a + \sum_{l=0}^{lights-1} f(d_l)\,[\,k_d L_{l_d} (L \cdot N) + k_s L_{l_s} (R \cdot V)^{ns}]$$

Shading

- **Flat Shading**
  - Calculate one lighting calculation (pick a vertex) per triangle
  - Color the entire triangle the same color
- **Gouraud Shading**
  - Calculate three lighting calculations (the vertices) per triangle
  - Linearly interpolate the colors as you scan convert
- **Phong Shading**
  - While do scan convert, linearly interpolate the normals.
  - With the interpolated normal at each pixel, calculate the lighting at each pixel

Lighting in OpenGL

- OpenGL supports the four types of light sources.
- OpenGL allows at least eight light sources set in a program.
- We must specify and enable individually (as exactly required by the Phong model)

Steps of Specifying Lighting

- Define normal vectors for each vertex of every object.
- Create, position, and enable one or more light sources.
- Select a lighting model.
- Define material properties for the objects in the scene.
- Don’t forget to *Enable/disable lighting.*

Creating Light Sources

- Define light properties
  - color, position, and direction

glLight*(Glenum *light*, Glenum *pname*, TYPE *param*)

Create the light specified by *light*, which can be GL_light0, GL_light1,…GL_light7.

pname indicates the properties of light that will be specified with *param*.

Creating Light Sources

- **Color**

```
GLfloat light_ambient[] = {0.0, 0.0, 1.0, 1.0);
GLfloat lght_diffuse[] = {1.0, 1.0, 1.0, 1.0);

glLightfv (GL_LIGHT0, GL_AMBIENT, lgiht_ambient);
glLightfv (GL_LIGHT1, GL_DIFFUSE, lgiht_diffuse);

glEnable(GL_LIGHT0);
glEnable(GL_LIGHT1);
```

Creating Light Sources

- **Position**

```
GLfloat light_position[] = {1.0, 1.0, 1.0, 0.0);
glLightfv (GL_LIGHT0, GL_POSITION, lgiht_position);

GLfloat spot_dir[] = {-1.0, -1.0, 0.0);
glLightfv (GL_LIGHT1, GL_SPOT_DIRECTION, spot_dir);

glEnable(GL_LIGHT0);
glEnable(GL_LIGHT1);
```

Creating Light Sources

- **Controlling a Light's Position and Direction**

OpenGL treats the position and direction of a light source just as it treats the position of a geometric primitives. In other word, a light source is subject to the same matrix transformations as a primitives.

- – A light position that remains fixed
- – A light that moves around a stationary object
- – A light that moves along with the viewpoint

Selecting Lighting Model

- OpenGL notion of a lighting model has three components:
  - The global ambient light intensity.
  - Whether the viewpoint position is local to the scene or whether it should be considered to be infinite distance away.
  - Whether lighting calculations should be performed differently for both the front and back faces of objects.

glLightModel*(Glenum *pname*, TYPE *param*)

Sets properties of the lighting model.

Selecting Lighting Model

- **Global ambient light**

  GLfloat lmodel_ambient[] = {0.2, 0.3, 1.0, 1.0);

  glLightModelfv (GL_LIGHT_MODEL_AMBIENT, lmodel_ambient);

- **Local or Infinite Viewpoint**

  glLightModelfi (GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE);

- **Two-sided Lighting**

  glLightModelfi (GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE);

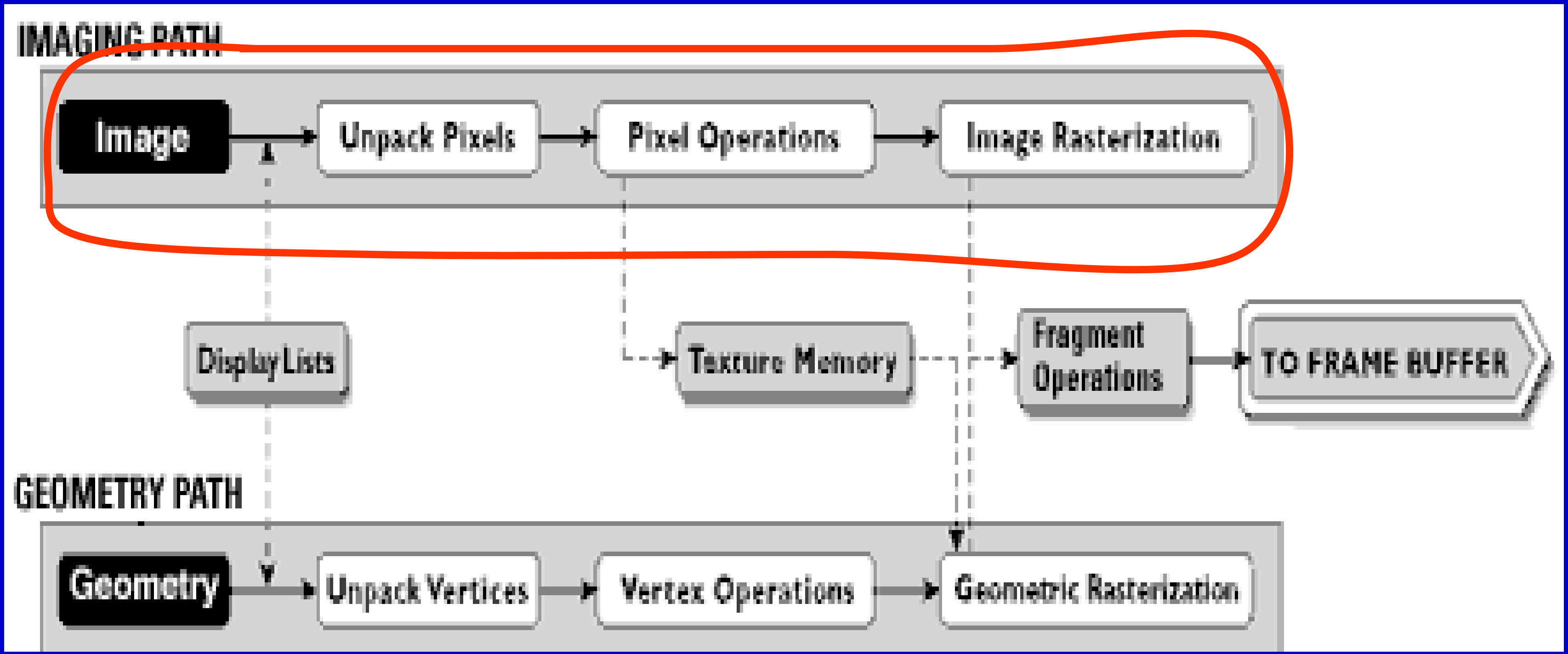
Defining Material Properties

- OpenGL supports the setting of material properties of objects in the scene
  - Ambient
  - Diffuse
  - Specular
  - Shininess

glMaterial*(Glenum face, Glenum *pname*, TYPE *param*)

Specifies a current material property foe use in lighting calculations.

OpenGL Image Path



Texture Mapping in OpenGL

- **Steps in Texture Mapping**
  - Create a texture object and specify a texture for that object.
  - Indicate how the texture is to be applied to each pixel.
  - Enable texture mapping.
  - Draw the scene, supplying both texture and geometric coordinates.

- Keep in mind that texture mapping works *only* in RGBA mode. Texture mapping results in color-index mode are undefined.

Specifying the Texture

GLubyte image[rows][cols]

void glTexImage2D (GLenum *target*, GLint *level*, GLint *internalFormat*, GLsizei *width*, GLsizei *height*, GLint *border*, GLenum *format*, GLenum *type*, const GLvoid **pixels*)

Note: both *width* and *height* must have the form 2m+2b, where m is nonnegative integer, and b is the value of *board*.

Texture Object

- Texture objects are an important new feature since OpenGL 1.1. A texture object stores data and makes it readily available.
- To use texture objects for your texture data, take these steps:
  - Generate texture names.
  - Initially bind (create) texture objects to texture data, including the image arrays and texture properties.
  - Bind and rebind texture objects, making their data currently available for rendering texture models.

Naming a Texture Object

- Any nonzero unsigned integer may be used as a texture name. To avoid accidentally resulting names, consistently use glGenTexture() to provide unused texture names.

void glGenTextures (Glsizei *n*, GLuint **textureNames*)

Creating and Using Texture Object

- The same routine, glBindTexture(), both *creates* and *uses* texture objects.

void glBindTexture (GLenum *target*, GLuint *textureName*)

When using it first time, a new texture object is *created*. When binding to previously created texture object, that texture object becomes *active*.

Ex:

glBindTexture(GL_TEXTURE_2D, name);

Cleaning Up Texture Objects

void glDeleteTexture (GLsizei *n*, const GLuint *textureNames*)

Delete n texture object, named by elements in the array textureNames. The freed texture names may now be reused.

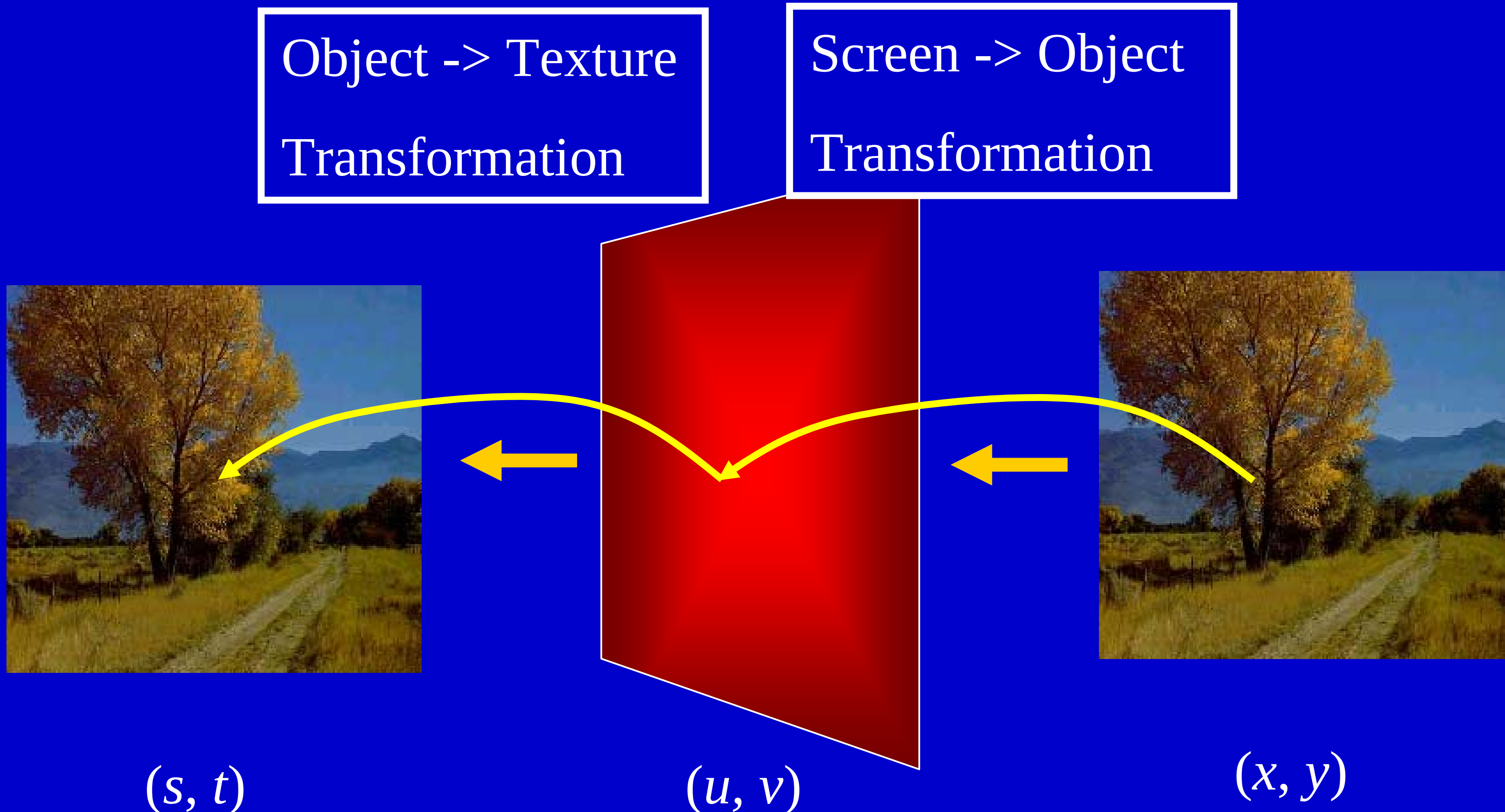
Set Texture Parameters

void glTexParameter* (GLenum *target*, GLenum *pname*, TYPE *param*)

Set various parameters that control how a texture is treated as it's applied or stored in a texture object.

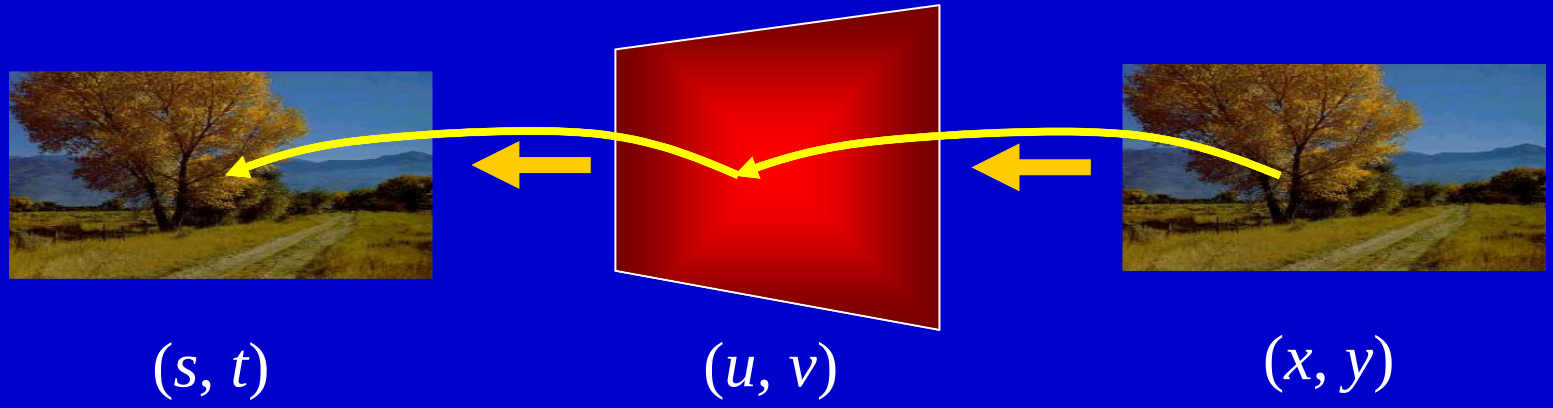
How to control texture mapping and rendering?

Texture Mapping Process



Rendering the Texture

- Rendering texture is similar to shading: It proceeds across the surface *pixel-by pixel*. For each pixel, it must determine the corresponding texture coordinates (*s, t*), access the texture, and set the pixel to the proper texture color.



Combining Lighting and Texturing

- There is no lighting involved with texture mapping
- They are *independent* operations, which may be combined
- It all depends on how to “apply” the texture to the underlying triangle

Set Texturing Function

void glTexEnv (GLenum *target*, GLenum *pname*, TYPE *param*)

Set the current texturing function.

We can use directly the texture colors to *paint* the object, or use the texture values to *modulate* or *blend* the color in the texture map with the original color of object.

Ex:

glTexEnv(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_BLEND)

Assigning Texture Coordinates

void glTexCoord* (TYPE *coords*)

Sets the current texture coordinates. Subsequent calls to glVertex*() result in those vertices being assigning the current texture coordinates.

```
glBegin(GL_QUAN) {
glTexCoord2f (0, 0);   glVertex2f (0, 0, 5);
glTexCoord2f (1, 0);   glVertex2f (10, 0, 5);
glTexCoord2f (1, 1);   glVertex2f (10, 10, 5);
glTexCoord2f (0, 1);   glVertex2f (0, 10, 5);
}
```

Remember to Enable Texture

glEnable (GL_TEXTURE_2D)

glDisable (GL_TEXTURE_2D)

Enable/disable texture mapping

Automatic Texture-Coordinate Generation

- OpenGL can *automatically* generate texture coordinate for you.

void glTexGen* (GLenum *coord*, GLenum *pname*, TYPE *param*

Specifies the function for automatically generating texture coordinates.

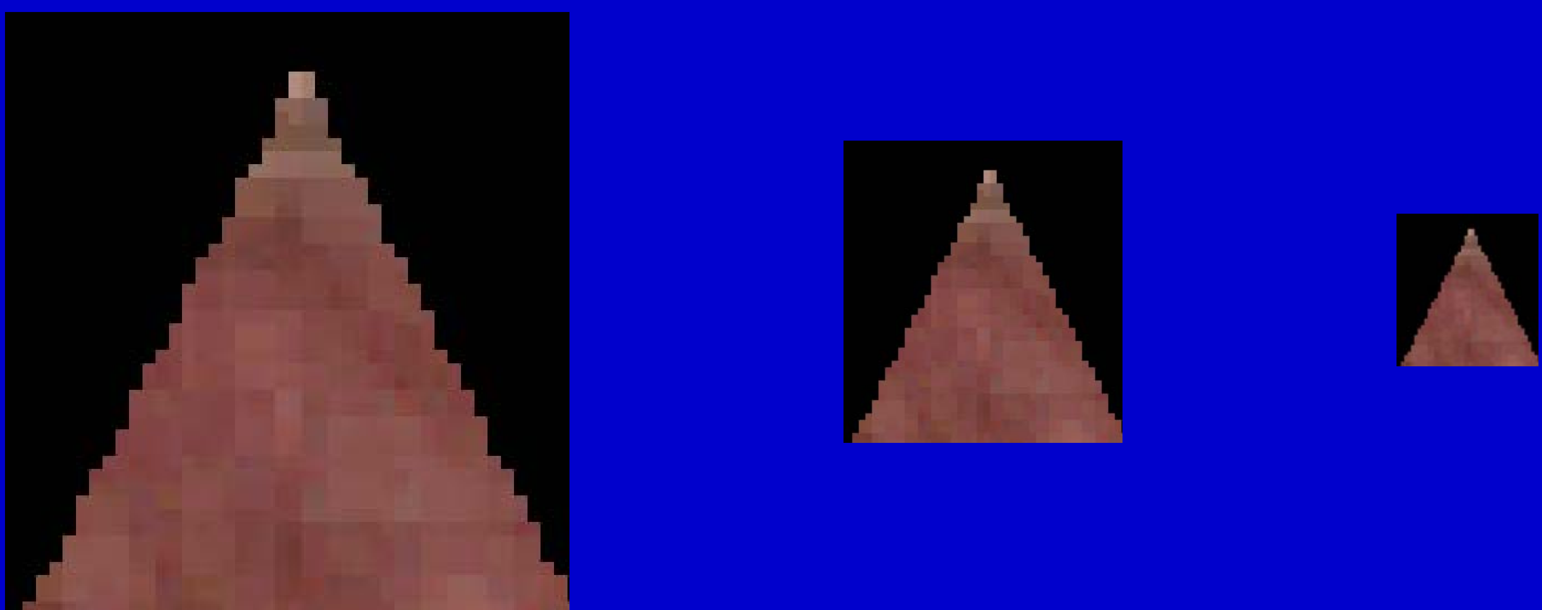
coord: GL_S, GL_T, GL_R, GL_Q

Recall Aliasing

- Aliasing manifests itself as “jaggies” in graphics. Thus we don’t have enough pixels to accurately represent the underlying function.
- Three reasons
  - Pixel numbers are fixed in the frame buffer
  - Pixel locations are fixed on a uniform
  - Pixel size/shape are fixed
- How do we fix it?
  - Increase resolution
  - Filtering

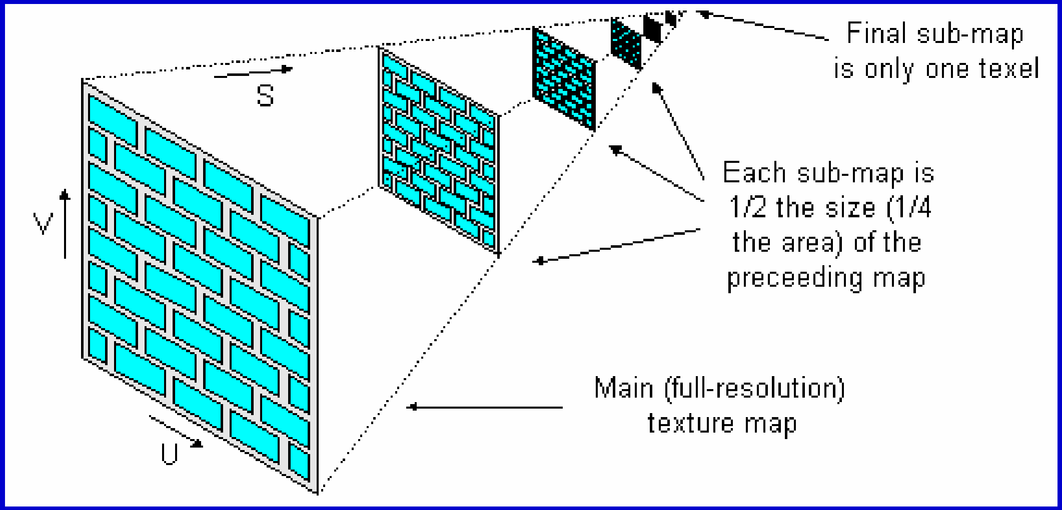
Increase Rendering Resolution

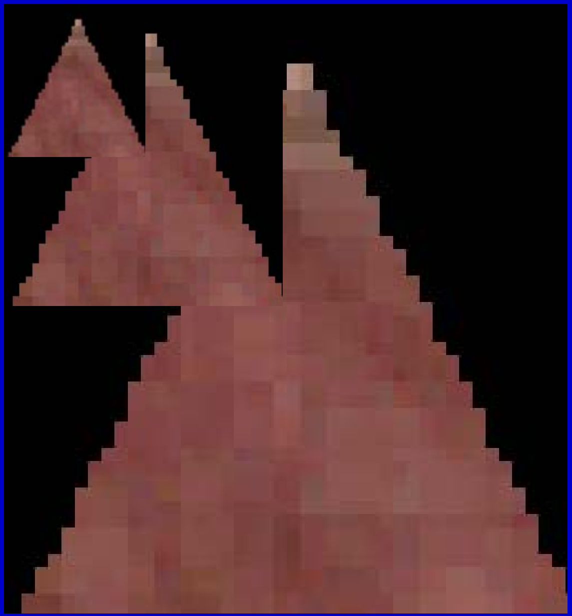
- Render the image at a higher resolution and downsample (like you are letting your eye do some filtering).



Mip Mapping

- MIP - *multium in parvo* - many in a small place.
- Build a pyramid of images, each smaller and filtered from the original.





Mipmapping

- Thus as we render, we choose the texture that best “fits” what you are drawing.
- OpenGL supports mipmapping

```
gluBuild2DMipmaps(GL_TEXTURE_2D, GL_RGBA, width, height,
    GL_RGBA, GL_UNSIGNED_BYTE, image);

glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
    GL_LINEAR_MIPMAP_LINEAR);
```

Bump Mapping

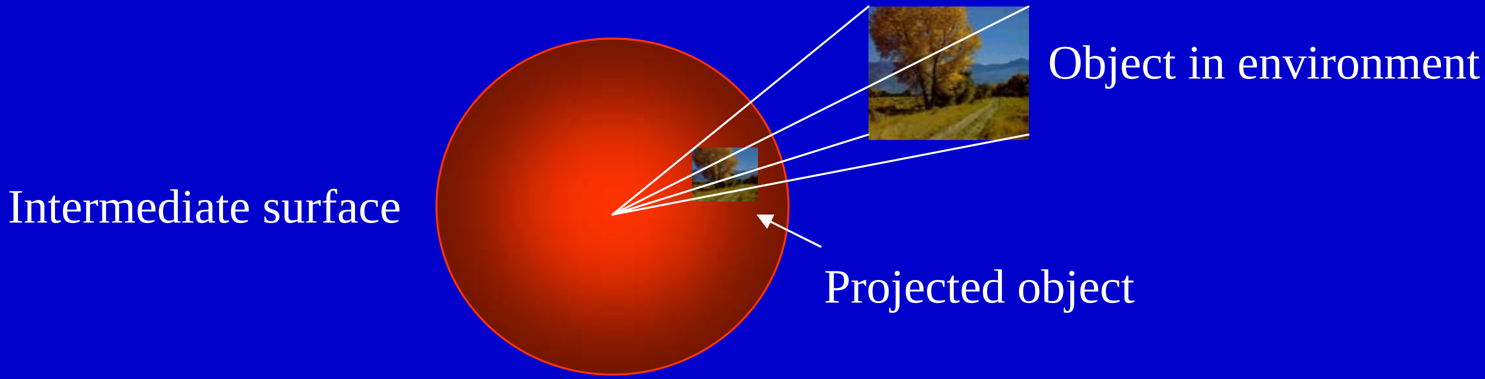
- We can also use textures for so much more than just images!
- We can use the textures as a "road map" on how to perturb normals across a surface.
- As we shade each pixel, perturb the normal by the partial derivatives of the corresponding *s*, *t* in the bump map.

Environmental Mapping

- Highly reflective surface are characterized by specular reflections that mirror the environment.
- We can extend our mapping techniques to obtain an image that approximates the desired reflection by extending texture maps to environmental (reflection) maps.

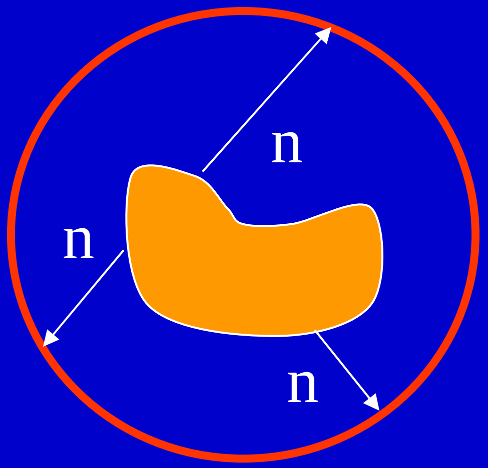
Environmental Mapping

- **Two-pass texture mapping**
  (1) Map the texture to a 3D *intermediate surface*.
  (2) Map the intermediate surface to the surface being rendered.



Environmental Mapping

- Then, we need to map the texture values on the intermediate surface to the desired surface.



Now It’s Your Turn

- Find a good reference/book
- Play with an example
- Make your own code

Computer graphics is best learned by doing!