Faculty of German Engineering and Industrial Management Education - FDIBA

Introduction to Computer Graphics

OpenGL Basics

Following the SIGGRAPH Lecture
of Prof. Ed Angel
+ Materials from the red book

Why study OpenGL

- OpenGL is the dominant graphics API
  - Supported on almost all architectures
  - Makes it easy to teach a programming-oriented class
- Another possibility:
  - DirectX: Microsoft platform dependent

What is OpenGL?

- Graphics rendering API
  - high-quality color images composed of geometric and image primitives
  - window system independent
  - operating system independent
  - close to hardware
    - leads to discussions of algorithms and implementation

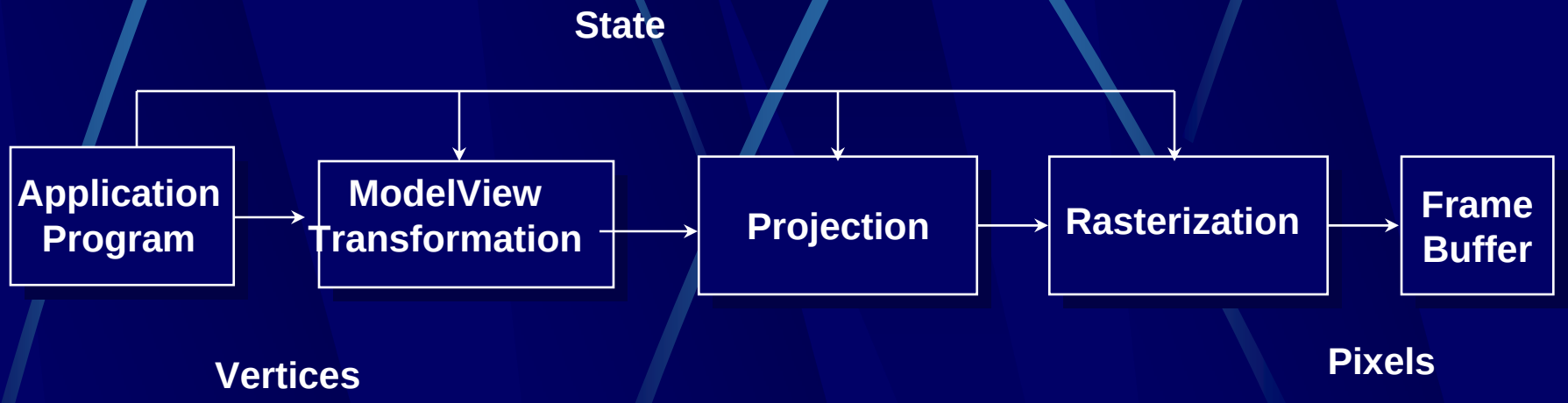
Physical Approaches to Rendering

- Ray tracing
  - Close to physics for simple models
  - Easy to teach
  - Easy to start with
  - Limited
- Radiosity
  - Closer to rendering equation
  - Not suitable for real time graphics
  - Not suitable for teaching first class

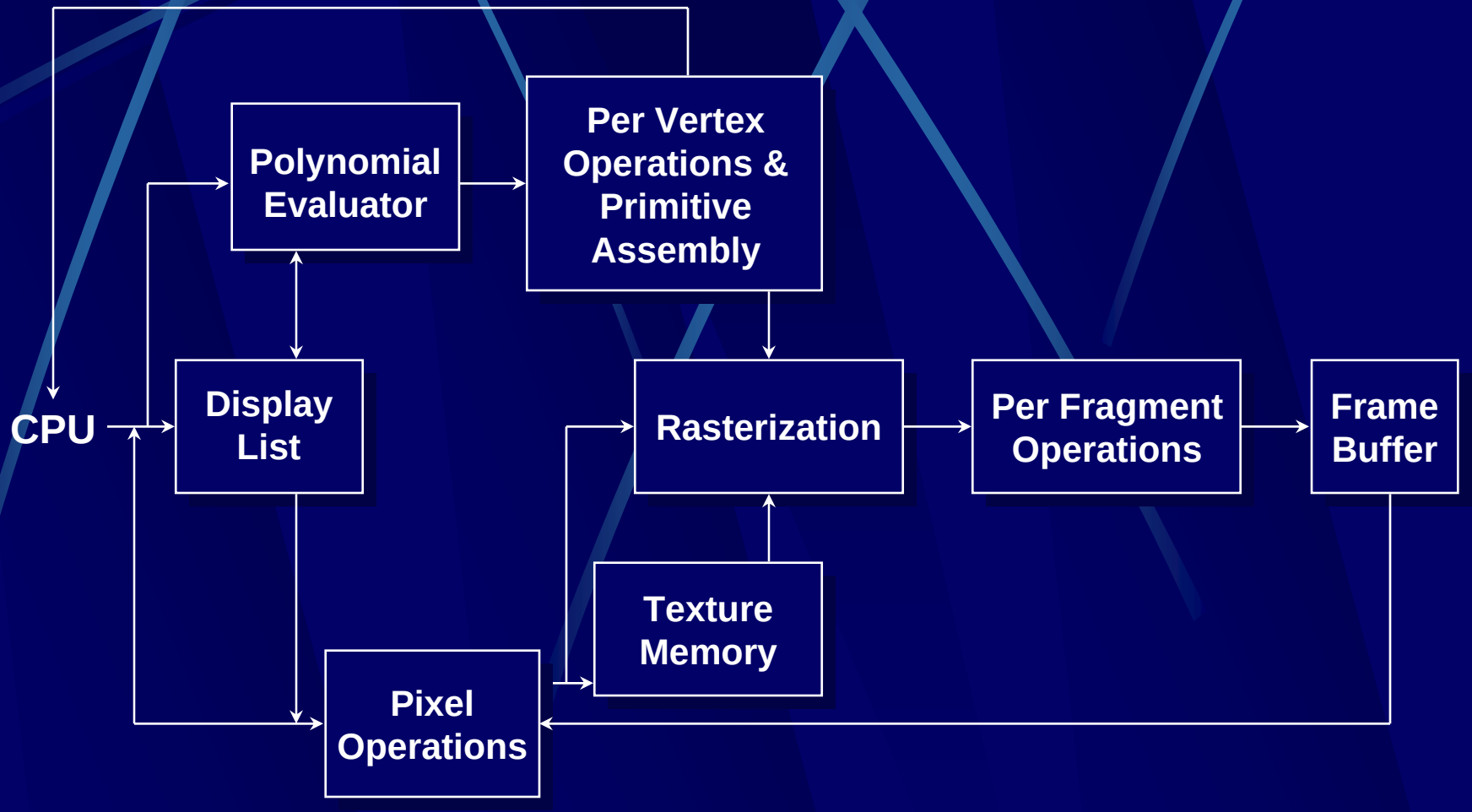
Pipeline Model

- Close to hardware
- Easy to program
- Direct implementation of synthetic camera
  - Image formation by computer is analogous to image formation by camera (or human)
  - Specify viewer, objects, lights
  - Let hardware/software form image (via projection)

OpenGL Geometric Pipeline



OpenGL Architecture



OpenGL as a Renderer

- Geometric primitives
  - points, lines and polygons
- Image Primitives
  - images and bitmaps
  - separate pipeline for images and geometry
    - linked through texture mapping
- Rendering depends on state
  - colors, materials, light sources, etc.

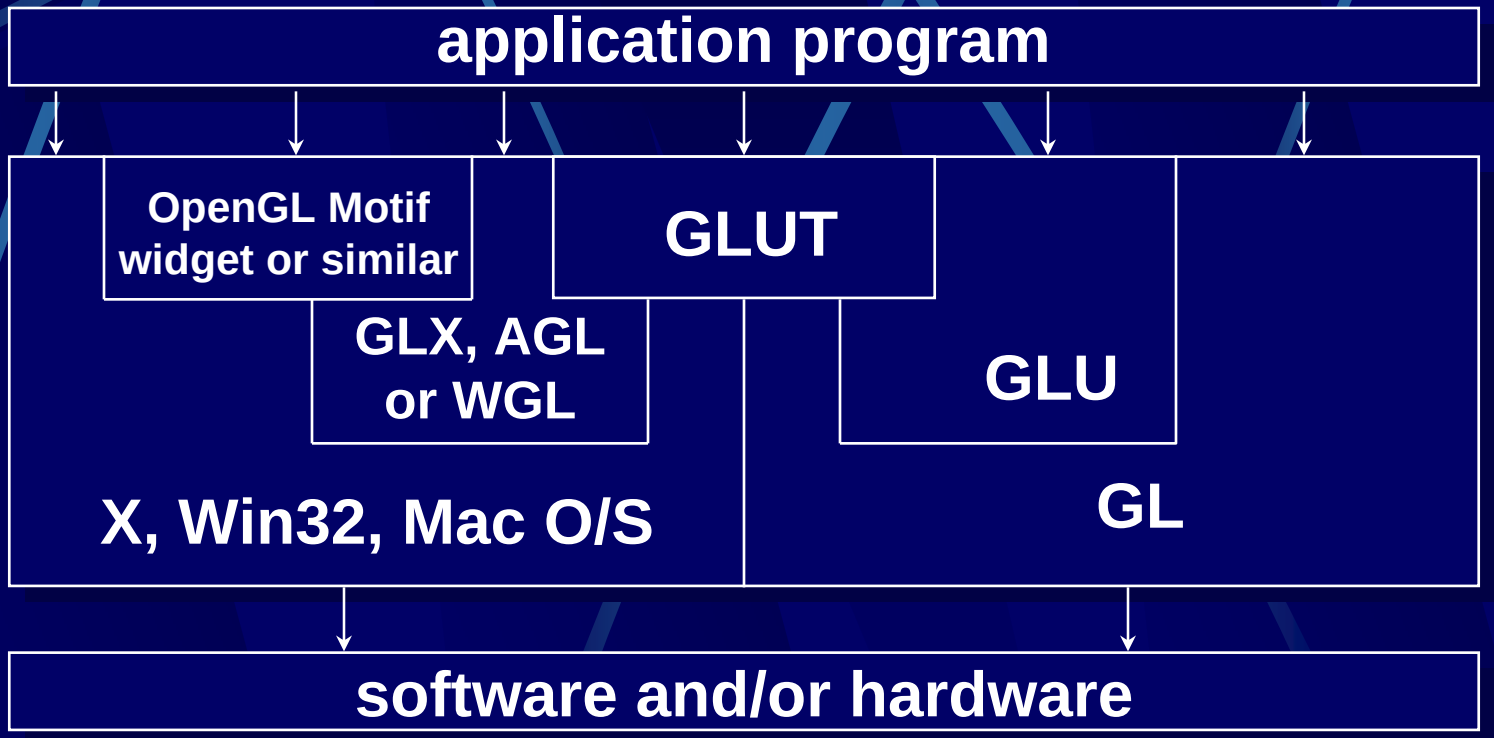
OpenGL and the Windowing System

- OpenGL is concerned only with rendering
  - Window system independent
  - No input functions
- OpenGL must interact with underlying OS and windowing system
  - Need minimal interface which may be system dependent
    - Done through additional libraries: AGL, GLX, WGL

GLU and GLUT

- GLU (OpenGL Utility Library)
  - part of OpenGL
  - NURBS, tessellators, quadric shapes, etc.
- GLUT (OpenGL Utility Toolkit)
  - portable windowing API
  - not officially part of OpenGL

OpenGL and Related APIs



Preliminaries

- Headers Files
  - #include <GL/gl.h>
  - #include <GL/glu.h>
  - #include <GL/glut.h>
- Libraries
- Enumerated Types
  - OpenGL defines numerous types for compatibility
    - GLfloat, GLint, GLenum, etc.

GLUT Basics

- Application Structure
  - Configure and open window
  - Initialize OpenGL state
  - Register input callback functions
    - render
    - resize
    - input: keyboard, mouse, etc.
  - Enter event processing loop

Sample Program

```
void main( int argc, char** argv )
{
   int mode = GLUT_RGB|GLUT_DOUBLE;
   glutInitDisplayMode( mode );
   glutCreateWindow( argv[0] );
   init();
   glutDisplayFunc( display );
   glutReshapeFunc( resize );
   glutKeyboardFunc( key );
   glutIdleFunc( idle );
   glutMainLoop();
}
```

GLUT Callback Functions

- Routine to call when something happens
  - window resize or redraw
  - user input
  - animation
- “Register” callbacks with GLUT

```
glutDisplayFunc( display );
glutIdleFunc( idle );
glutKeyboardFunc( keyboard );
```

Rendering Callback

- Do all of your drawing here

```
glutDisplayFunc( display );

void display( void )
{
  glClear( GL_COLOR_BUFFER_BIT );
  glBegin( GL_TRIANGLE_STRIP );
    glVertex3fv( v[0] );
    glVertex3fv( v[1] );
    glVertex3fv( v[2] );
    glVertex3fv( v[3] );
  glEnd();
  glutSwapBuffers();
}
```

Idle Callbacks

- Use for animation and continuous update

```
glutIdleFunc( idle );

void idle( void )
{
  t += dt;
  glutPostRedisplay();
}
```

User Input Callbacks

- Process user input

```
glutKeyboardFunc( keyboard );
```

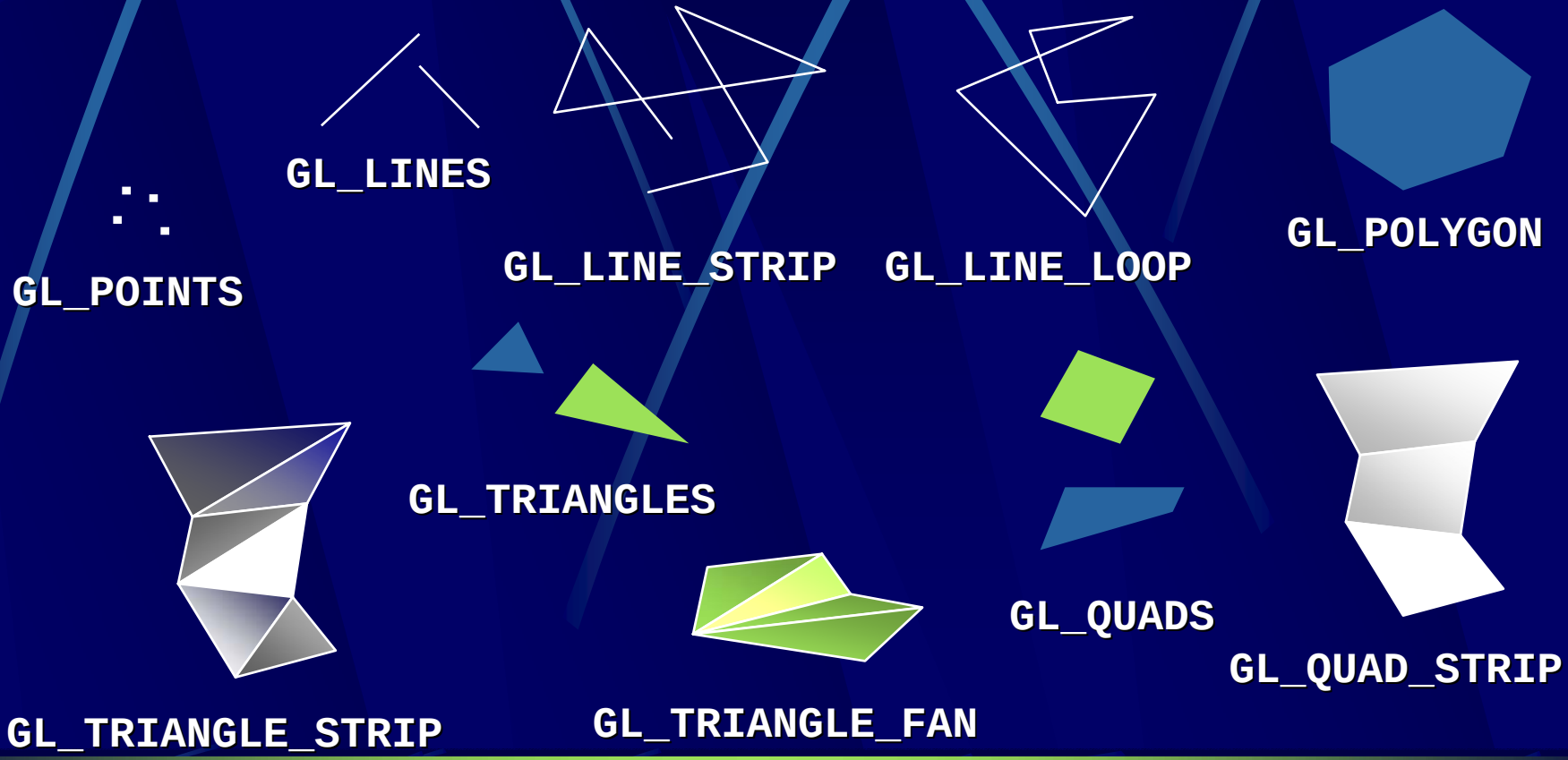
```
void keyboard( char key, int x, int
  y )
{
   switch( key ) {
      case ‘q’ : case ‘Q’ :
         exit( EXIT_SUCCESS );
         break;
      case ‘r’ : case ‘R’ :
         rotate = GL_TRUE;
         break;
   }
}
```

Elementary Rendering

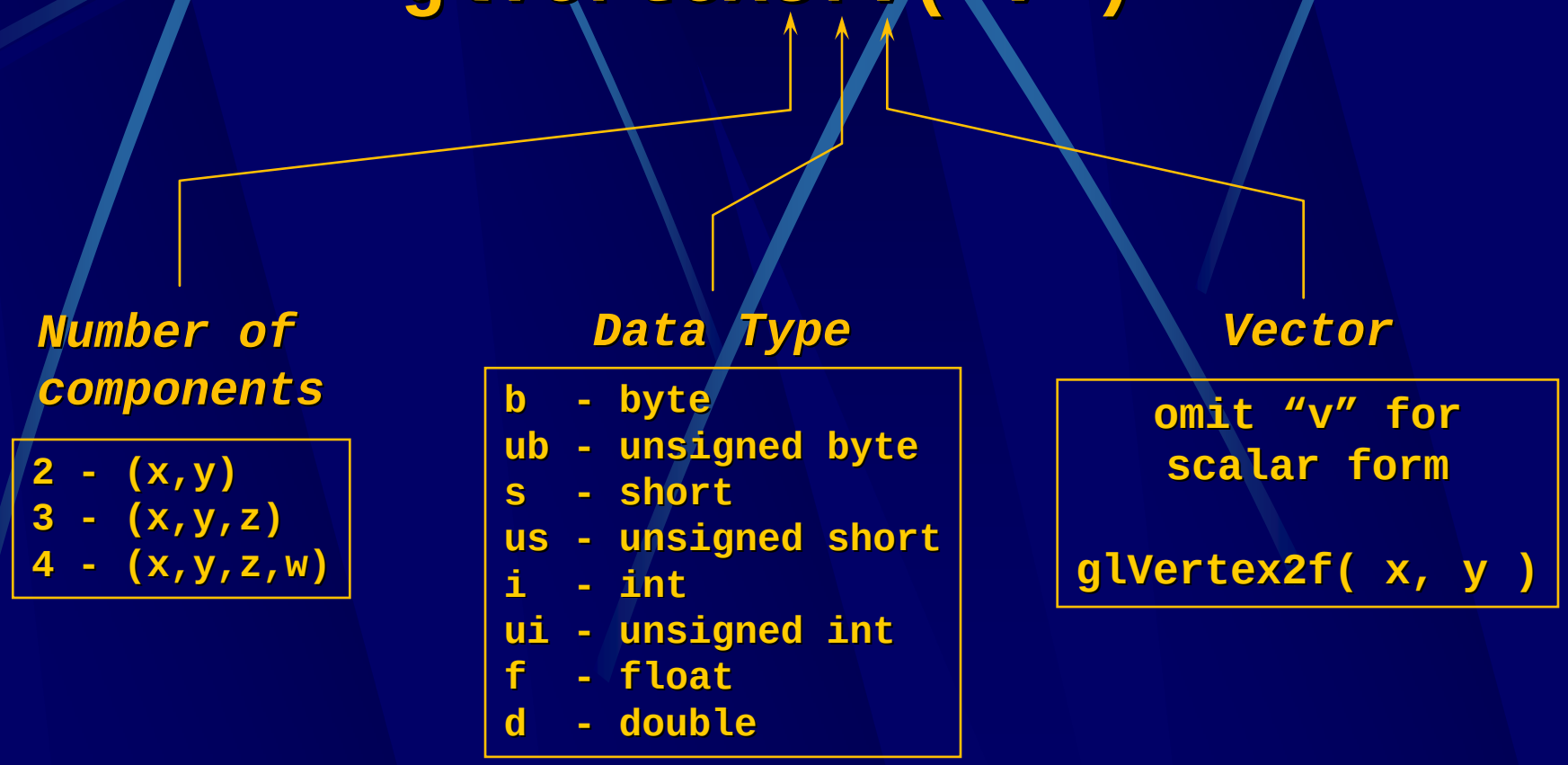
- Geometric Primitives
- Managing OpenGL State
- OpenGL Buffers

OpenGL Geometric Primitives

- All geometric primitives are specified by vertices



OpenGL Command Formats



Specifying Geometric Primitives

- Primitives are specified using

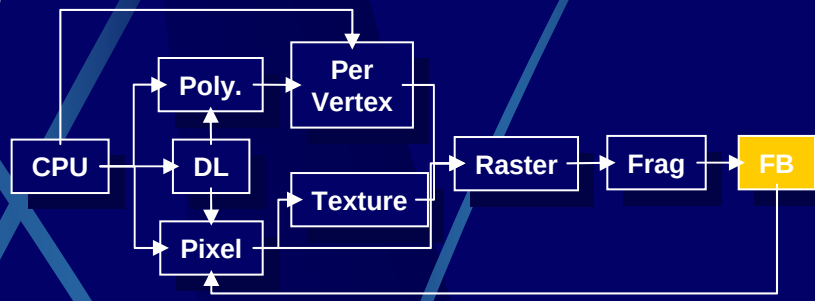
```
glBegin( primType );
glEnd();
```

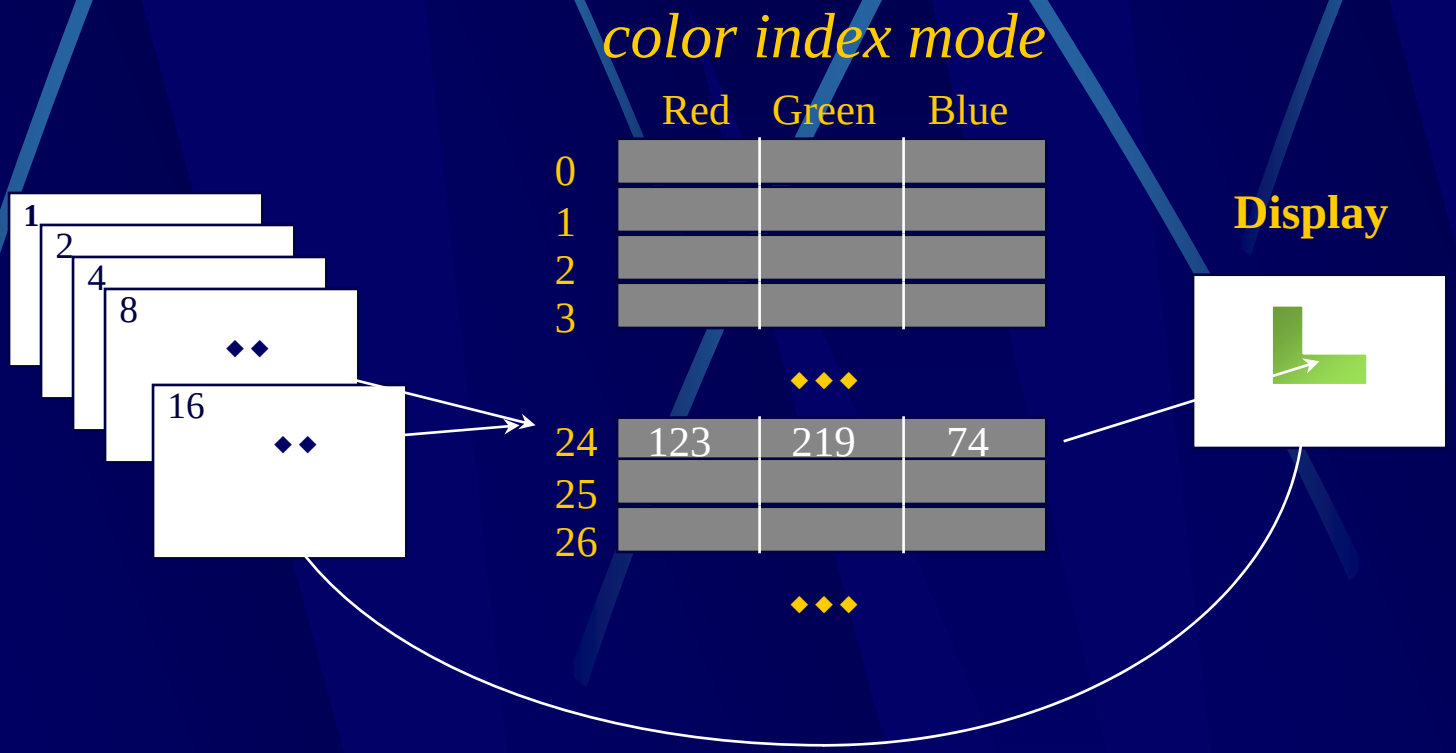
  - *primType* determines how vertices are combined

```
GLfloat red, greed, blue;
Glfloat coords[3];
glBegin( primType );
for ( i = 0; i < nVerts; ++i ) {
  glColor3f( red, green, blue );
  glVertex3fv( coords );
}
glEnd();
```

OpenGL Color Models

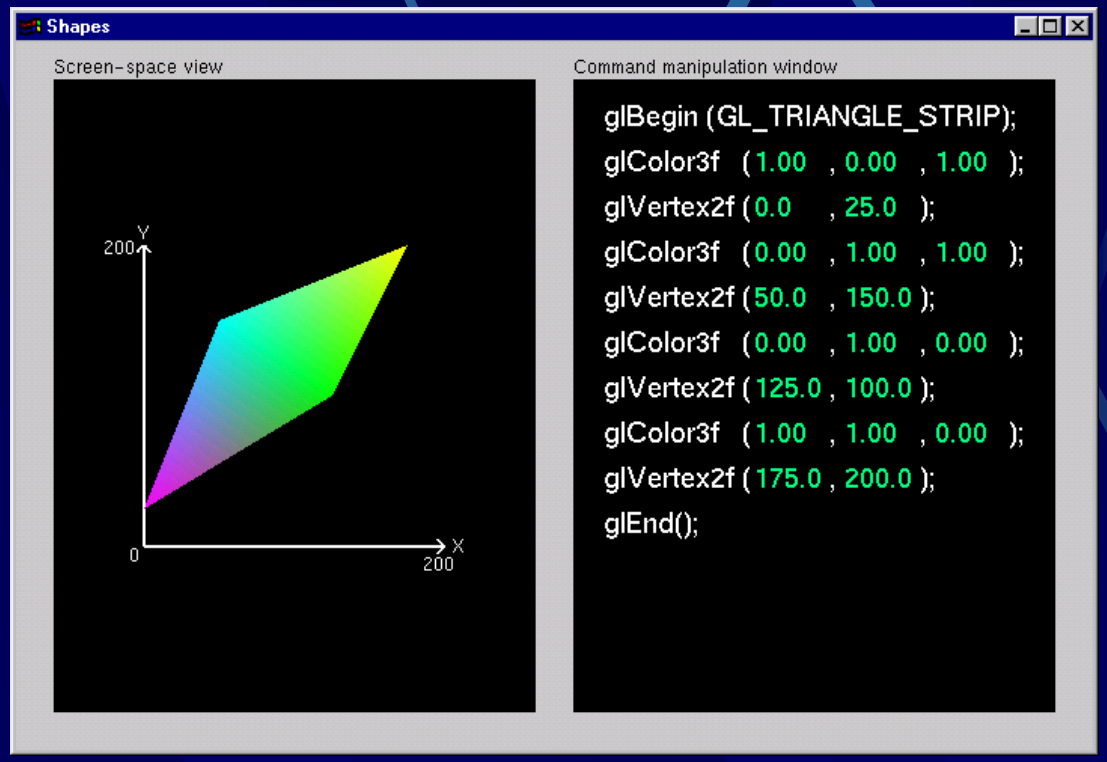
- RGBA or Color Index





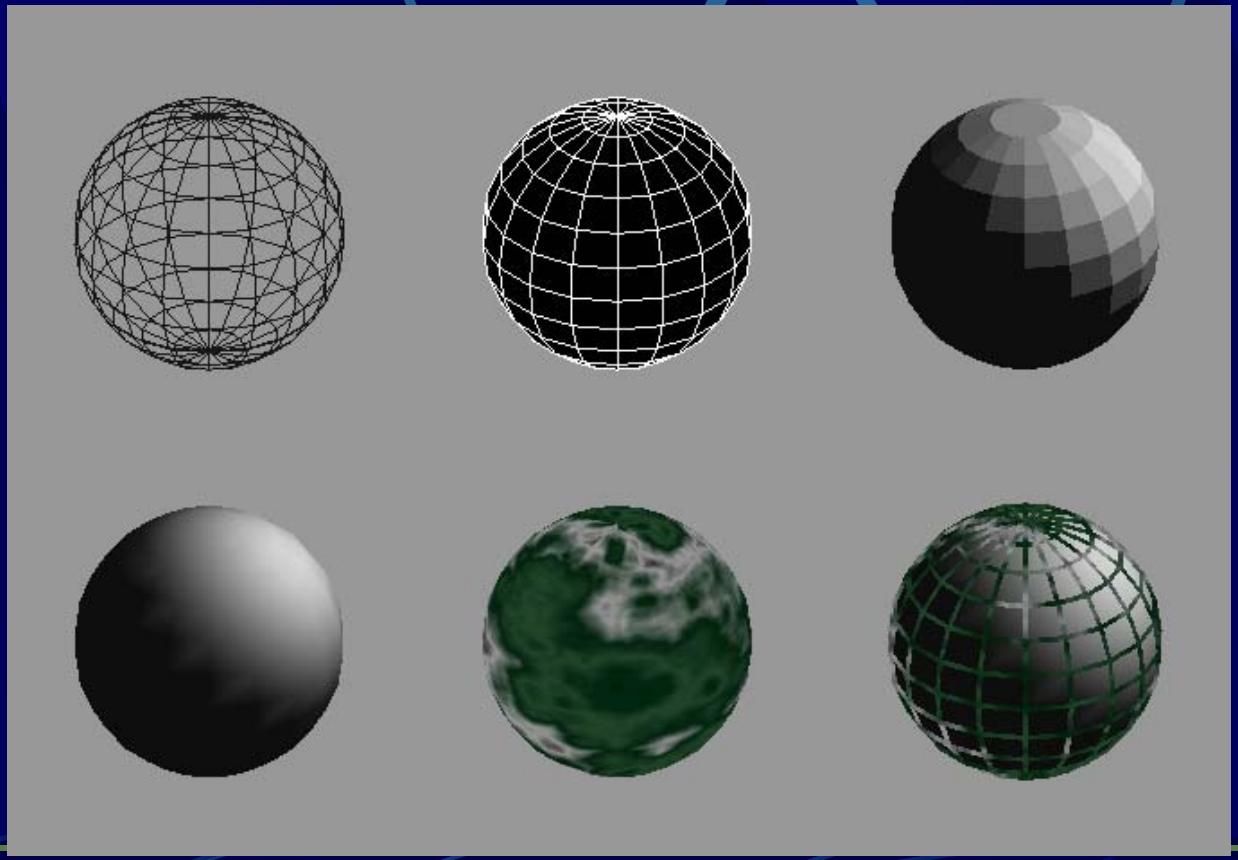
Shapes Tutorial

- developed by Nate Robbins
- Available on web (see references)



Controlling Rendering Appearance

- From Wireframe to Texture Mapped



OpenGL's State Machine

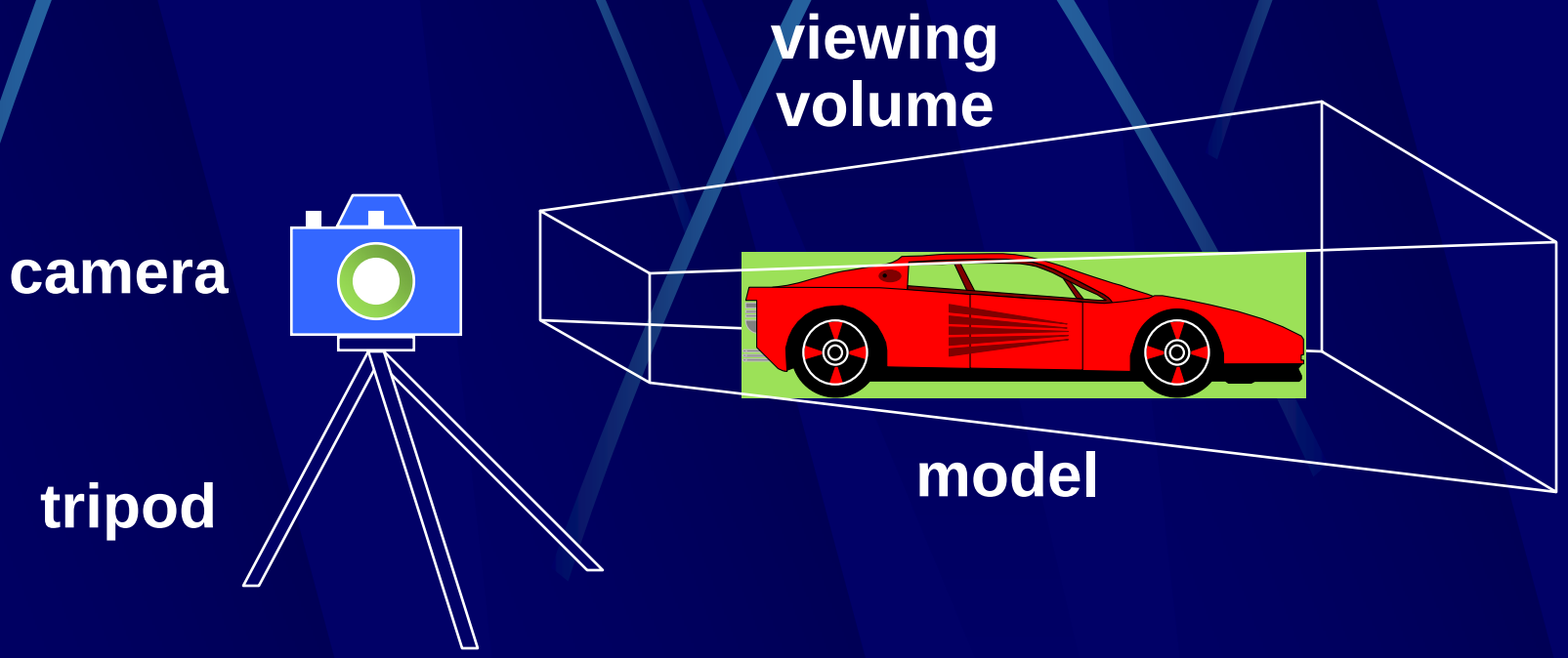
- All rendering attributes are encapsulated in the OpenGL State
  - rendering styles
  - shading
  - lighting
  - texture mapping

Transformations in OpenGL

- Modeling
- Viewing
  - orient camera
  - projection
- Animation
- Map to screen

Camera Analogy

- 3D is just like taking a photograph (lots of photographs!)



Camera Analogy and Transformations

- Projection transformations
  - adjust the lens of the camera
- Viewing transformations
  - tripod–define position and orientation of the viewing volume in the world
- Modeling transformations
  - moving the model
- Viewport transformations
  - enlarge or reduce the physical photograph

Coordinate Systems and Transformations

- Steps in Forming an Image
  - specify geometry (world coordinates)
  - specify camera (camera coordinates)
  - project (window coordinates)
  - map to viewport (screen coordinates)
- Each step uses transformations
- Every transformation is equivalent to a change in coordinate systems (frames)

Affine Transformations

- Want transformations which preserve geometry
  - lines, polygons, quadrics
- Affine = line preserving
  - Rotation, translation, scaling
  - Projection
  - Concatenation (composition)

Homogeneous Coordinates

- each vertex is a column vector

$$\vec{v} = \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

- w is usually 1.0
- all operations are matrix multiplications
- directions (directed line segments) can be represented with w = 0.0

3D Transformations

- A vertex is transformed by 4 x 4 matrices
  - all affine operations are matrix multiplications
  - all matrices are stored column-major in OpenGL
  - matrices are always post-multiplied
  - product of matrix and vector is

$$\mathbf{M}\vec{v}$$

$$\mathbf{M} = \begin{bmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{bmatrix}$$

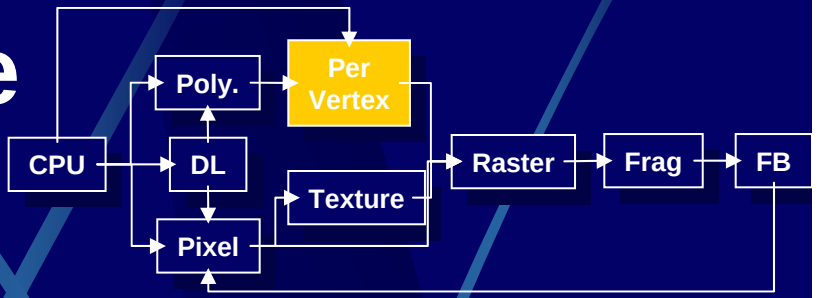
Specifying Transformations

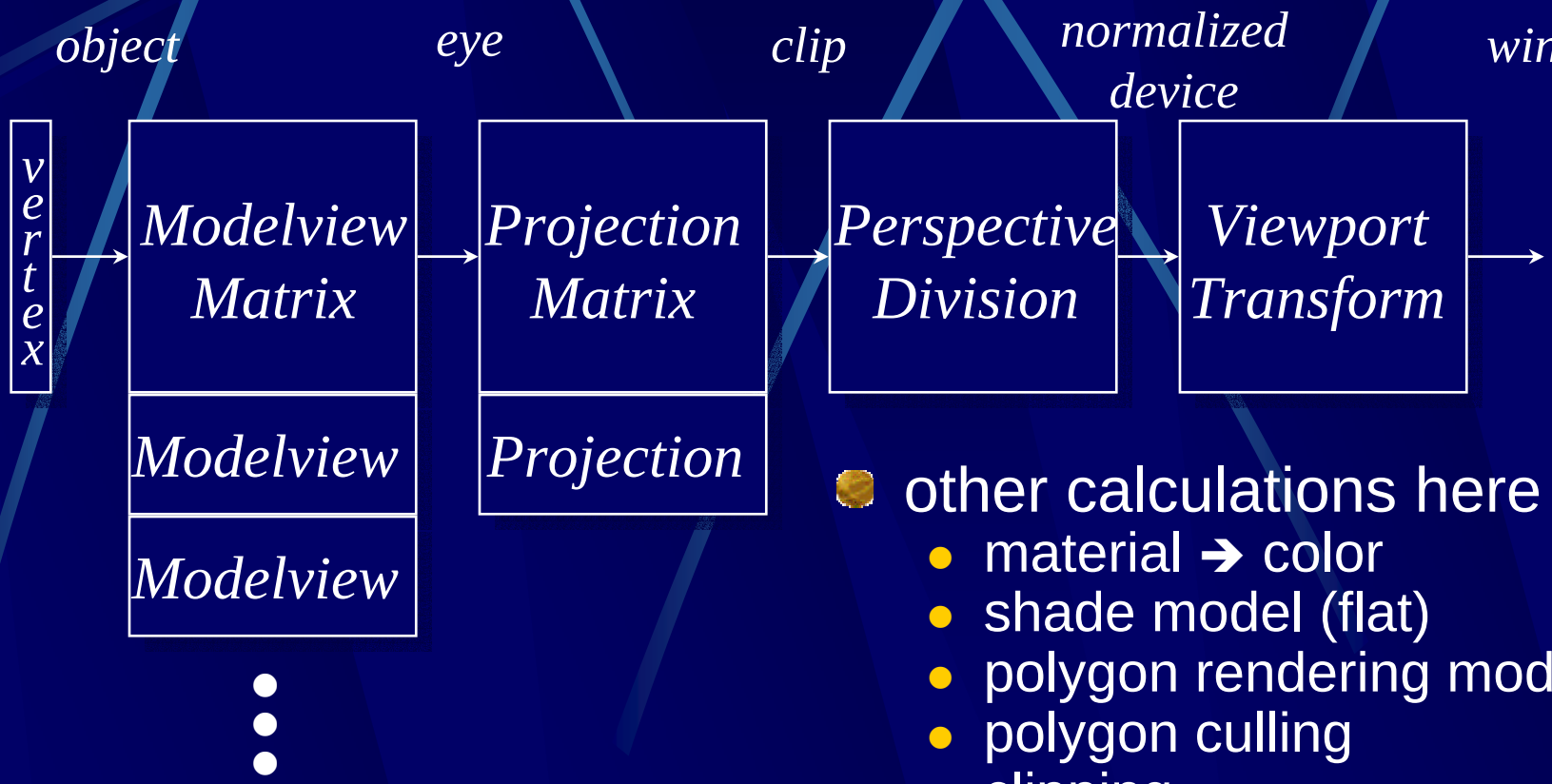
- Programmer has two styles of specifying transformations
  - specify matrices (`glLoadMatrix, glMultMatrix`)
  - specify operation (`glRotate, glOrtho`)
- Programmer does not have to remember the exact matrices
  - check appendix of Red Book (Programming Guide)

Programming Transformations

- Prior to rendering, view, locate, and orient:
  - eye/camera position
  - 3D geometry
- Manage the matrices
  - including matrix stack
- Combine (composite) transformations

Transformation Pipeline





- other calculations here
  - material ➔ color
  - shade model (flat)
  - polygon rendering mode
  - polygon culling
  - clipping

Matrix Operations

- Specify Current Matrix Stack

  `glMatrixMode( GL_MODELVIEW or GL_PROJECTION )`

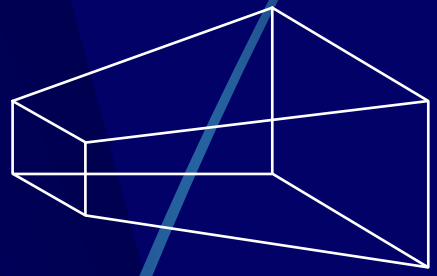
- Other Matrix or Stack Operations

  `glLoadIdentity()` `glPushMatrix()` `glPopMatrix()`

- Viewport
  - usually same as window size
  - viewport aspect ratio should be same as projection transformation or resulting image may be distorted

  `glViewport( x, y, width, height )`

Projection Transformation



- Shape of viewing frustum
- Perspective projection

```
gluPerspective( fovy, aspect,
  zNear, zFar )
glFrustum( left, right, bottom, top,
  zNear, zFar )
```

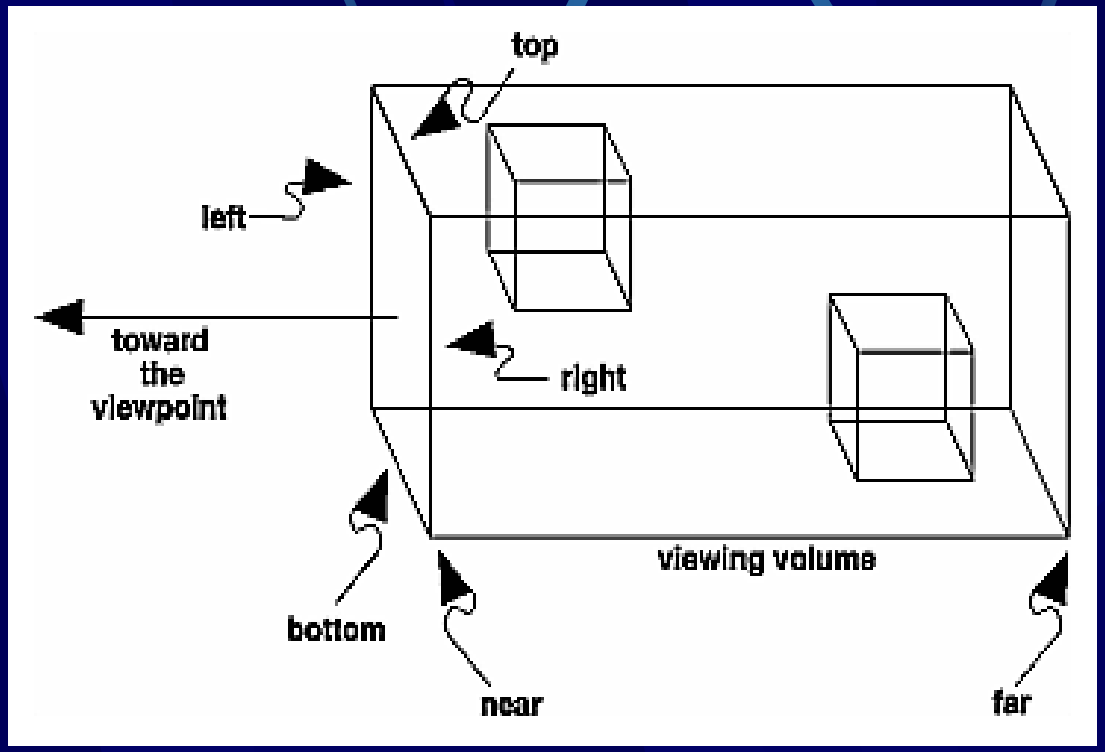
- Orthographic parallel projection

```
glOrtho( left, right, bottom, top,
  zNear, zFar )
gluOrtho2D( left, right, bottom,
  top )
```

  - calls **`glOrtho`** with z values near zero

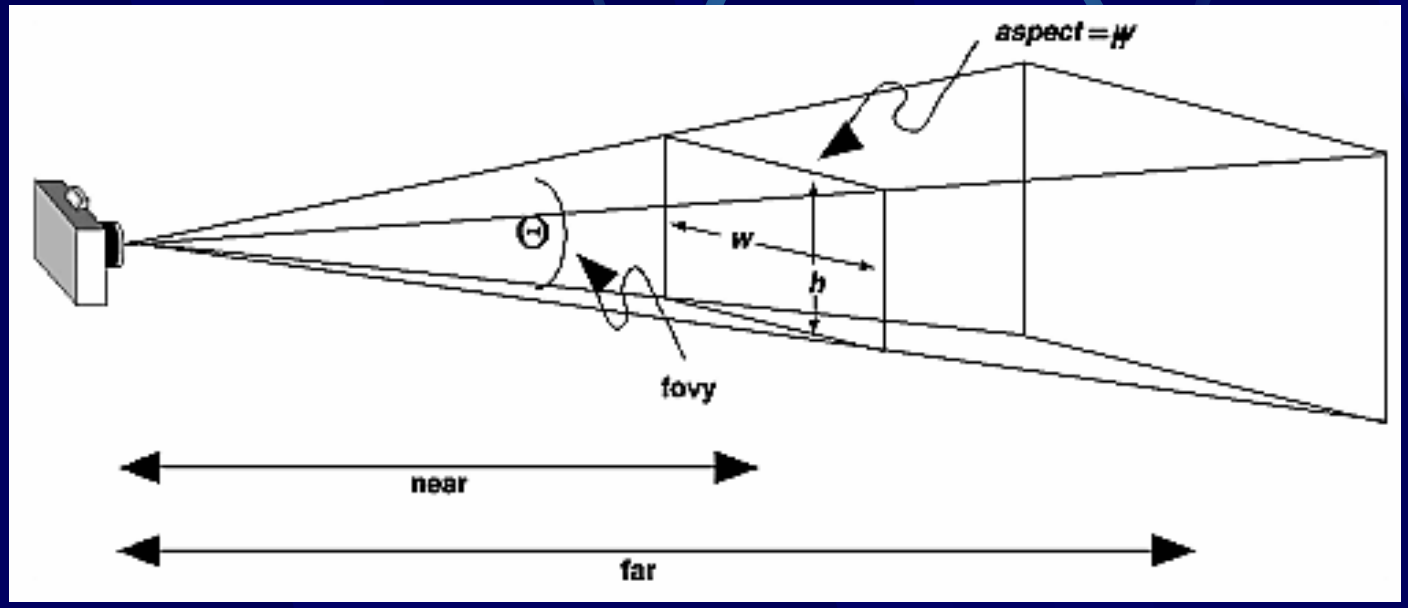
Program Structure

- void **glOrtho** (GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near, GLdouble far);
- Creates a matrix for an **orthographic parallel viewing volume** and multiplies the current matrix by it.



Program Structure

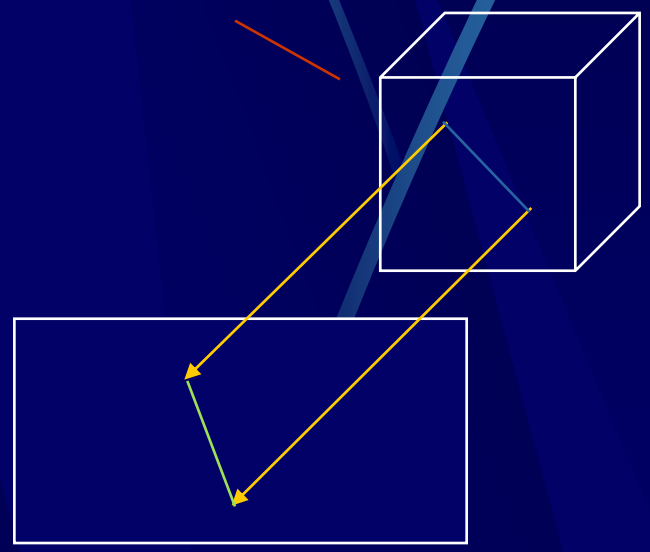
- void **gluPerspective** (GLdouble fovy, GLdouble aspect, GLdouble near, GLdouble far);
- Creates a matrix for an **symmetric perspective-view frustum** and multiplies the current matrix by it.
  - **fovy** is the angle of the field of view in the x-z plane;



Applying Projection Transformations

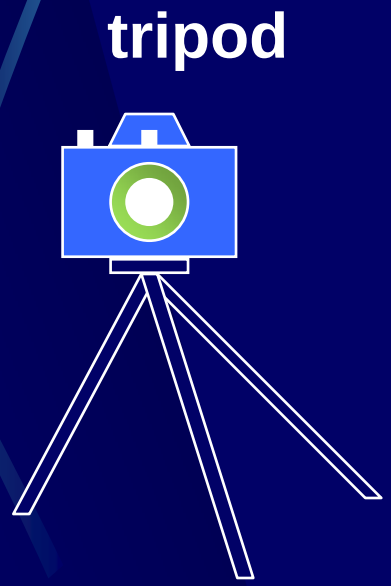
- Typical use (orthographic projection)

```
glMatrixMode( GL_PROJECTION );
glLoadIdentity();
glOrtho( left, right, bottom, top, zNear,
  zFar );
```



Viewing Transformations

- Position the camera/eye in the scene
  - place the tripod down; aim camera
- To “fly through” a scene
  - change viewing transformation and redraw scene
- `gluLookAt( eye`$_x$`, eye`$_y$`, eye`$_z$`,`
  `aim`$_x$`, aim`$_y$`, aim`$_z$`,`
  `up`$_x$`, up`$_y$`, up`$_z$ `)`
  - up vector determines unique orientation
  - careful of degenerate positions



Modeling Transformations

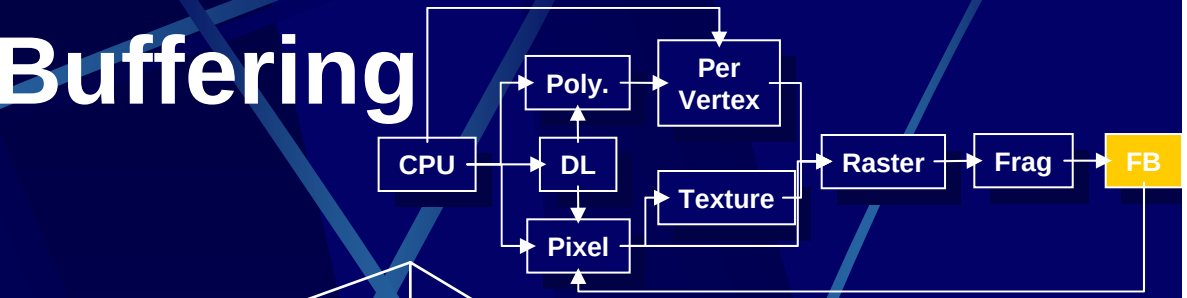
- Move object

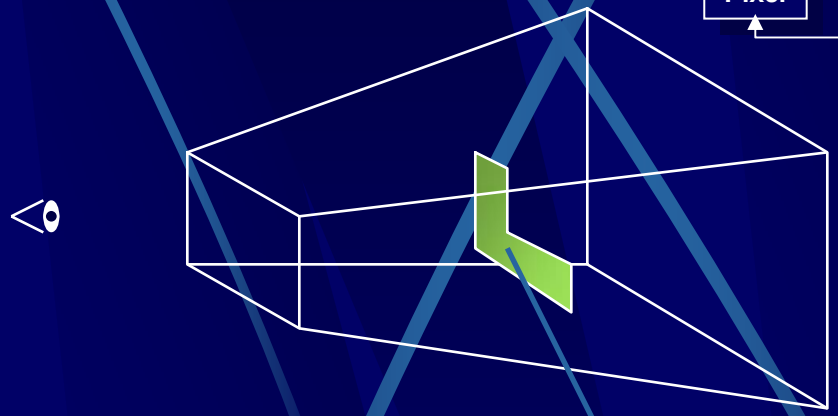
  `glTranslate{fd}( x, y, z )`
- Rotate object around arbitrary axis $(x \quad y \quad z)$

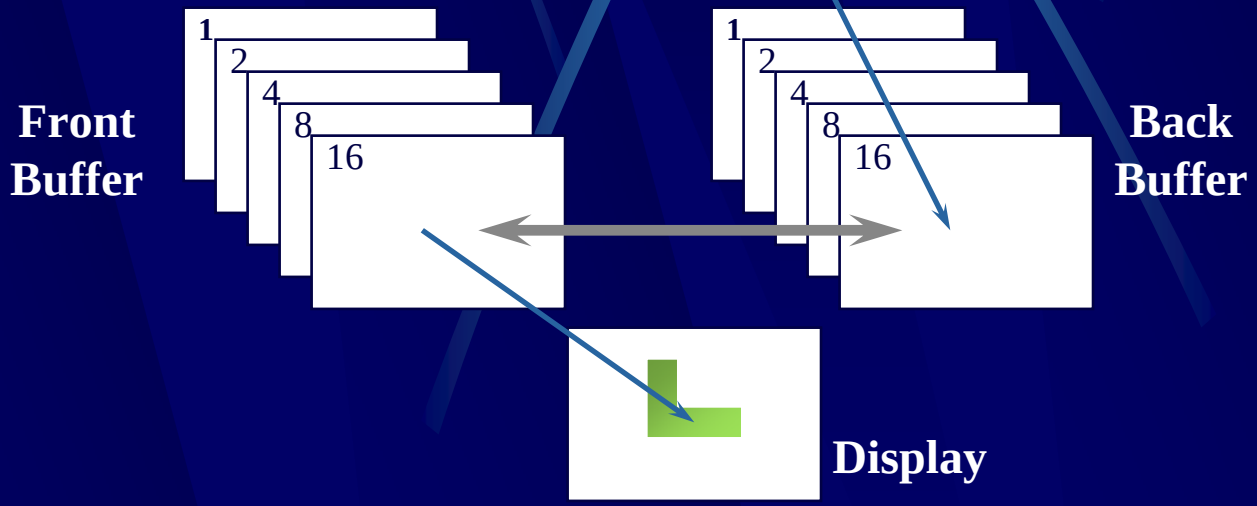
  `glRotate{fd}( angle, x, y, z )`
  - angle is in degrees
- Dilate (stretch or shrink) or mirror object

  `glScale{fd}( x, y, z )`

Double Buffering







Animation Using Double Buffering

① Request a double buffered color buffer

```
glutInitDisplayMode( GLUT_RGB | GLUT_DOUBLE
  );
```

② Clear color buffer

```
glClear( GL_COLOR_BUFFER_BIT );
```

③ Render scene

④ Request swap of front and back buffers

```
glutSwapBuffers();
```

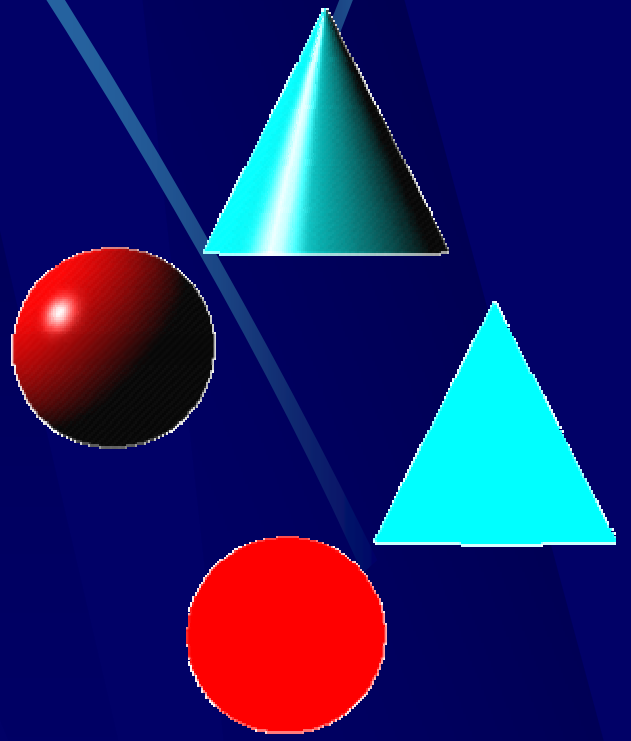
- Repeat steps 2 - 4 for animation

An Updated Program Template

```
void main( int argc, char** argv )
{
    glutInit( &argc, argv );
    glutInitDisplayMode( GLUT_RGB |
       GLUT_DOUBLE);
    glutCreateWindow( “Tetrahedron”
   );
    init();
    glutIdleFunc( idle );
    glutDisplayFunc( display );
    glutMainLoop();
}
```

Lighting Principles

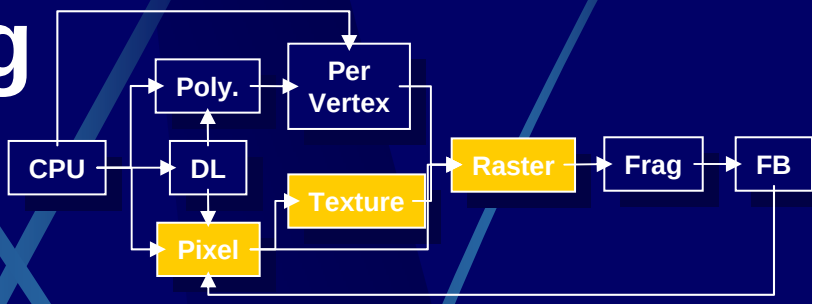
- Lighting simulates how objects reflect light
  - material composition of object
  - light's color and position
  - global lighting parameters
    - ambient light
    - two sided lighting
  - available in both color index and RGBA mode



How OpenGL Simulates Lights

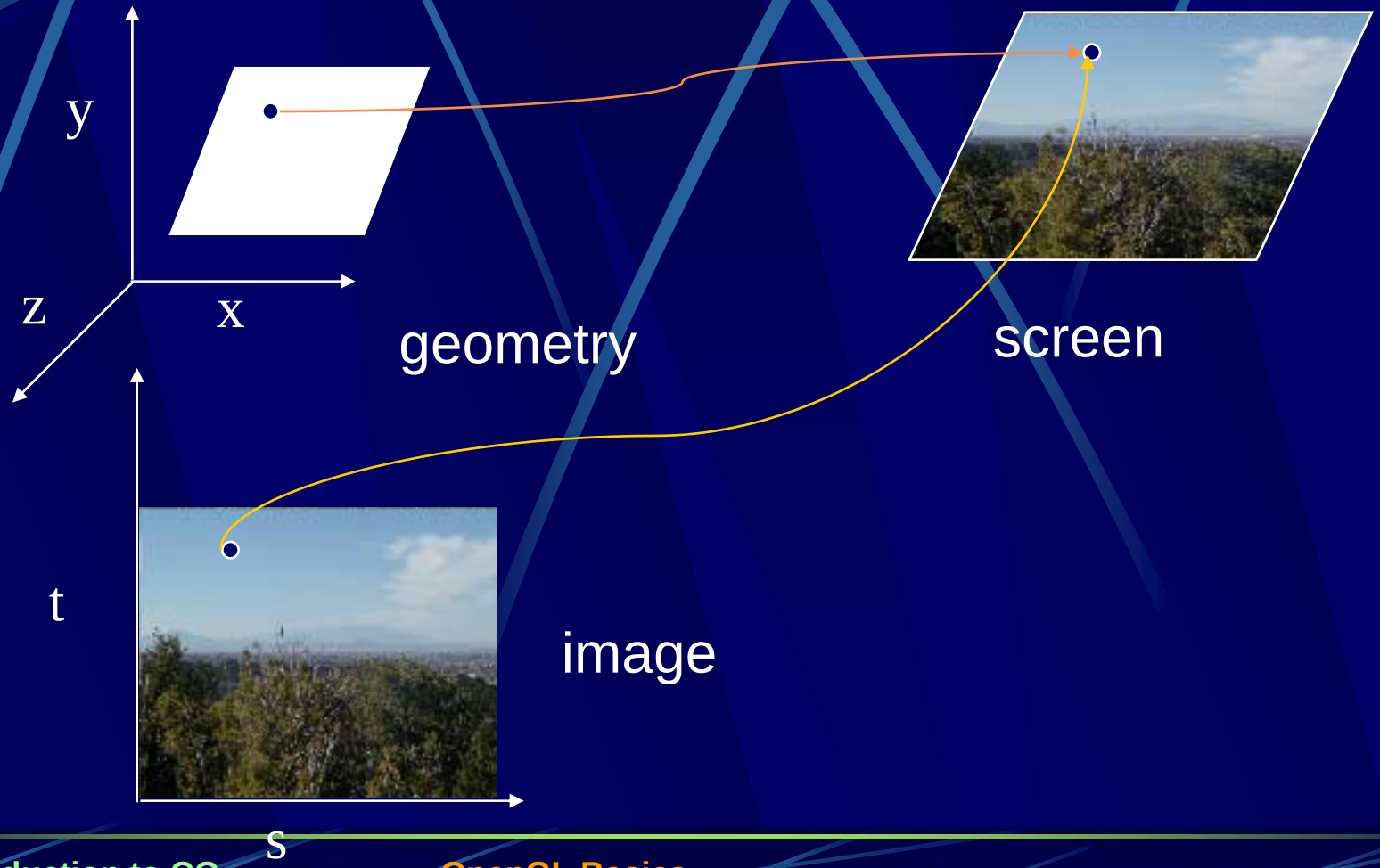
- Phong lighting model
  - Computed at vertices
- Lighting contributors
  - Surface material properties
  - Light properties
  - Lighting model properties

Texture Mapping



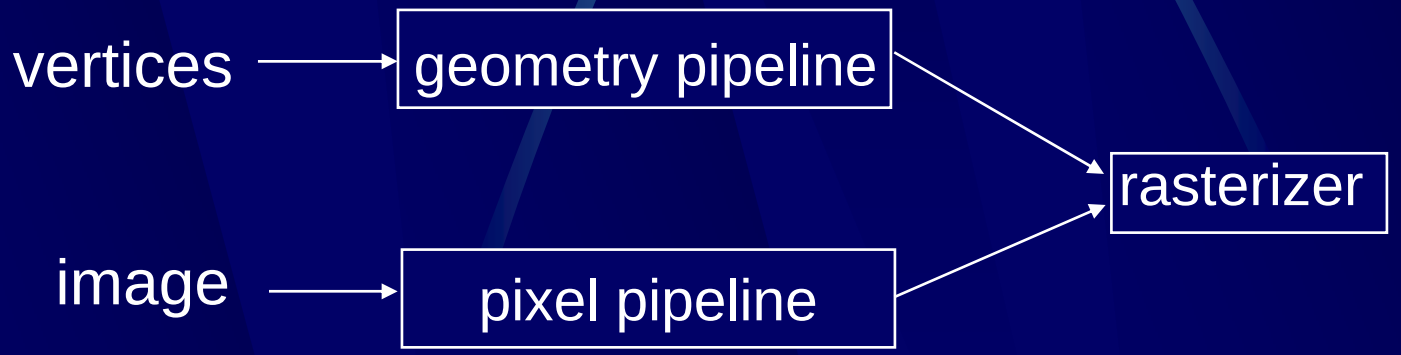
- Apply a 1D, 2D, or 3D image to geometric primitives
- Uses of Texturing
  - simulating materials
  - reducing geometric complexity
  - image warping
  - reflections

Texture Mapping



Texture Mapping and the OpenGL Pipeline

- Images and geometry flow through separate pipelines that join at the rasterizer
  - "complex" textures do not affect geometric complexity



Applying Textures I

- Three steps
  - ① specify texture
    - read or generate image
    - assign to texture
  - ② assign texture coordinates to vertices
  - ③ specify texture parameters
    - wrapping, filtering

Immediate Mode versus Display Listed Rendering

- Immediate Mode Graphics
  - Primitives are sent to pipeline and display right away
  - No memory of graphical entities
- Display Listed Graphics
  - Primitives placed in display lists
  - Display lists kept on graphics server
  - Can be redisplayed with different state
  - Can be shared among OpenGL graphics contexts

Ed Angel Class

- www.cs.unm.edu/~angel/CS433

On-Line Resources

- **http://www.opengl.org**
- **news:comp.graphics.api.opengl**
- **http://www.sgi.com/software/opengl**
- **http://www.mesa3d.org/ (Mesa)**
- **http://www.cs.utah.edu/~narobins/opengl.html (Tutorials)**
- **http://www.cs.unm.edu/~angel**
- **http://www.ecs.umass.edu/ece/hill**

- **Nate Robins**: GLUT ported to Win32 GLUT
  http://www.xmission.com/~nate/glut.html
  glut-3.7.6-bin.zip (117 KB)