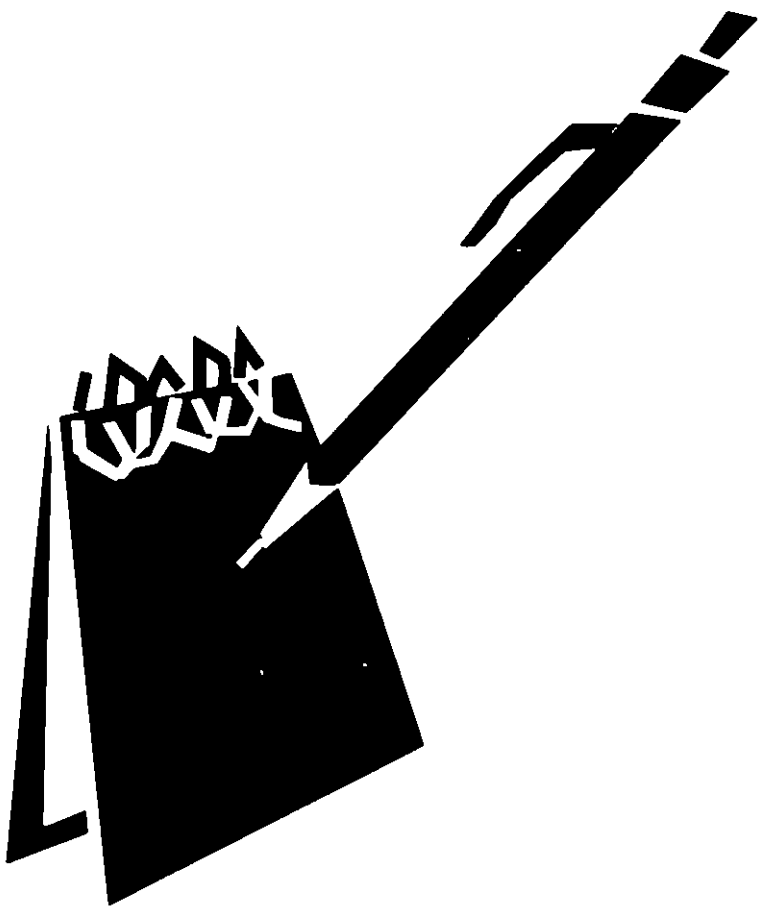
SIGGRAPH 98

7

A Visual Introduction to OpenGL Programming



25th International Conference on Computer Graphics and Interactive Techniques
Exhibition **21 23 July** 1998 Conference **19 24 July** 1998
Orlando, Florida USA

course notes

SIGGRAPH 98

7

A Visual Introduction to OpenGL Programming

Organizer
Mason Woo
World Wide Woo

Lecturers
Dave Shreiner
Silicon Graphics Inc

Mason Woo
World Wide Woo

25th International Conference on Computer Graphics and Interactive Techniques
Exhibition **21 23 July** 1998 Conference **19 24 July** 1998
Orlando, Florida USA

course notes

A Visual Introduction to OpenGL Programming

Siggraph 1998 - Course 7

Mason Woo
Dave Shreiner

Abstract

This course is an introduction to writing interactive graphics programs using the OpenGL API As opposed to seeing snippets of code and static captured images this course features interactive tools to visualize and experiment with computer graphics concepts such as transformations lighting and texture mapping Detailed explanations focus upon controlling the position and movement of the camera the light sources and objects in a scene The effects of changing the order of modeling transformations (and their associated matrices) are discussed and visually demonstrated Manipulating changes to parameters of the texture mapping API are shown with real-time graphics

Speaker Biographies

Mason Woo

After 10 years of training and marketing graphics libraries at Silicon Graphics Mason Woo became an independent consultant in 1996 He is co author of the OpenGL Programming Guide (Addison Wesley 2nd edition 1997) and former secretary of the OpenGL Architecture Review Board Mason has previously taught courses at SIGGRAPH the X Technical Conference and Exhibition and has been a speaker or panelist at JavaOne the Japan Personal Computer Software Association NCGA VESA Microsoft Win32 Professional Developer's Conference Defense & Government Computer Graphics Conference and SIGCHI & GI

Dave Shreiner

Dave is a member of the OpenGL development team at Silicon Graphics Computer Systems He has 10 years of experience with visual simulation and scientific visualization including 7 years at Silicon Graphics He was the original author of Introductory OpenGL Programming for Silicon Graphics Technical Education department Dave has a Bachelors of Mathematics from University of Delaware and has done graduate work at the Johns Hopkins University

Syllabus

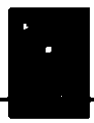
1 30pm	Woo	Welcome & OpenGL Introduction (pages 1-14)
1 50pm	Shreiner	Elementary Rendering (pages 15-36)
2 20pm	Woo	Matrix Transformations (pages 37-56)
3 00pm	Break	
3 15pm	Shreiner	Lighting Models (pages 57-68)
3 45pm	Shreiner	Texturing (pages 69-84)
4 25pm	Woo	Overview of Other Topics (pages 85 106)
5 15pm	All	Summay Q & A (pages 107-109)

Table of Contents

A Visual Introduction to OpenGL Programming

Mason Woo
Dave Shreiner

Course Flow



Introduction

- What is OpenGL?
- OpenGL Architecture
- OpenGL and the window system
- Typical program structure

Elementary Rendering

- Geometry Primitives
- States
- Animation
- Buffers

2

Course Flow

Viewing and Transformations

Lighting

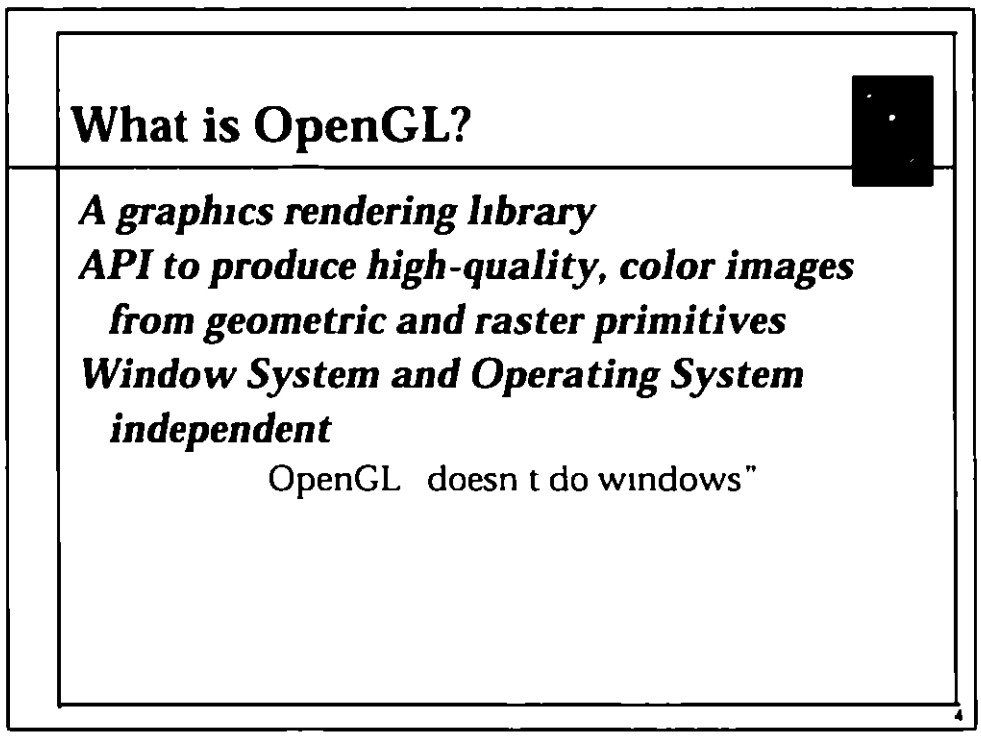
Texture mapping

Overview of Other Operations

- Display Lists
- Feedback
- Image Primitives
- Fog Antialiasing Fragment operations

Summary and References, Q&A

3



A graphics rendering library is a layer of abstraction between graphics hardware and an application program

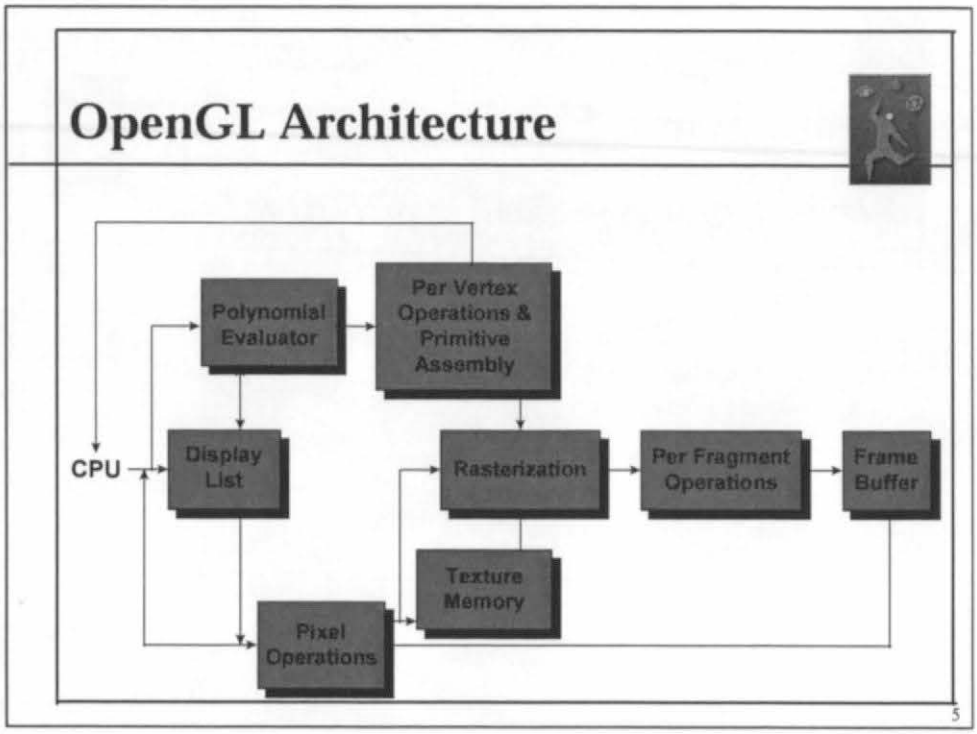
API = Application Programming (or Procedural) Interface

Geometric primitives are vertex-based and are either 2D or 3D

Raster primitives are pixel-based (either bitmaps or pixmaps) and generally 2D Texture mapping combines both raster and geometric primitives to create an image

OpenGL libraries are supported for use with X Window System® and UNIX®, Microsoft Windows®, Microsoft Windows NT®, and IBM OS/2®

OpenGL does not perform operations which are redundant with the window system window management, event (mouse & keyboard) handling, and loading color maps



This is the most important diagram you will see, representing the flow of graphical information, as it is processed from CPU to the frame buffer.

There are two pipelines of data flow. The upper pipeline is for geometric, vertex-based primitives. The lower pipeline is for pixel-based, image primitives. Texturing combines the two types of graphics together.

There is a pull-out poster in the back of the OpenGL Reference Manual (blue book), which shows this diagram in more detail.

OpenGL as a Renderer

Renders simple geometric primitives

- points lines polygons

Renders images and bitmaps

- separate pipelines for geometry and pixels linked through texture mapping

Rendering depends on state

- colors light sources materials
- surface normals texture coordinates

6

OpenGL and Related APIs

GLU (OpenGL Utility Library)

- guaranteed to be available
- tesselators quadrics NURBs etc
- some surprisingly common operations such as projection transformations (such as `gluPerspective`)

GLX or WGL

- bridge between window system and OpenGL

GLUT

- portable bridge between window system and OpenGL
- not "standard" but informal popularity

7

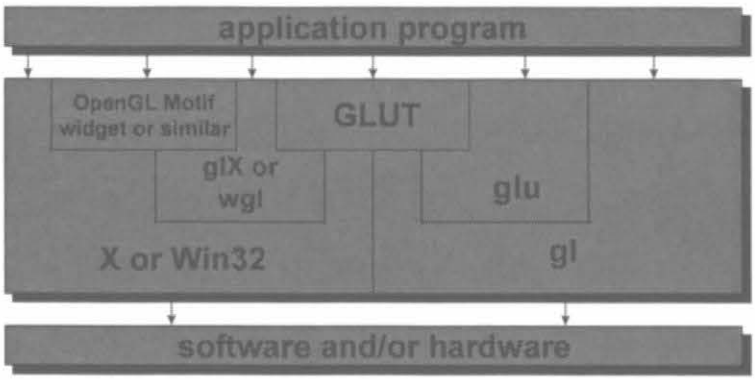
The GLU (OpenGL Utility library) is a set of commonly used graphics routines mandatory for *all* implementations of OpenGL All routines in the GLU have the prefix, *glu* The GLU contains more complicated commands, such as tesselators, quadric objects, and NURBS

The GLX and WGL libraries are extensions of the X Window System and Microsoft Windows, respectively They support operations to create an OpenGL context, visuals (or corresponding pixel format), frame buffer configuration (including double buffering and depth buffer size), and synchronization

The GLUT (OpenGL Utility Toolkit) is a set of portable, convenience routines to deal with window management, event handling, and modeling some basic 3D objects Implementations of GLUT have been ported to different window systems, including both X and Microsoft Windows, so programs written with GLUT port very easily GLUT is not an official, governed API, it was originally written by Mark Kilgard and has gained informal acceptance in the OpenGL community

Mark Kilgard's book, *OpenGL Programming for the X Window System*, is published by Addison-Wesley (ISBN 0-201-48359-9) His book includes extensive description of the GLUT toolkit

OpenGL and Related APIs



8

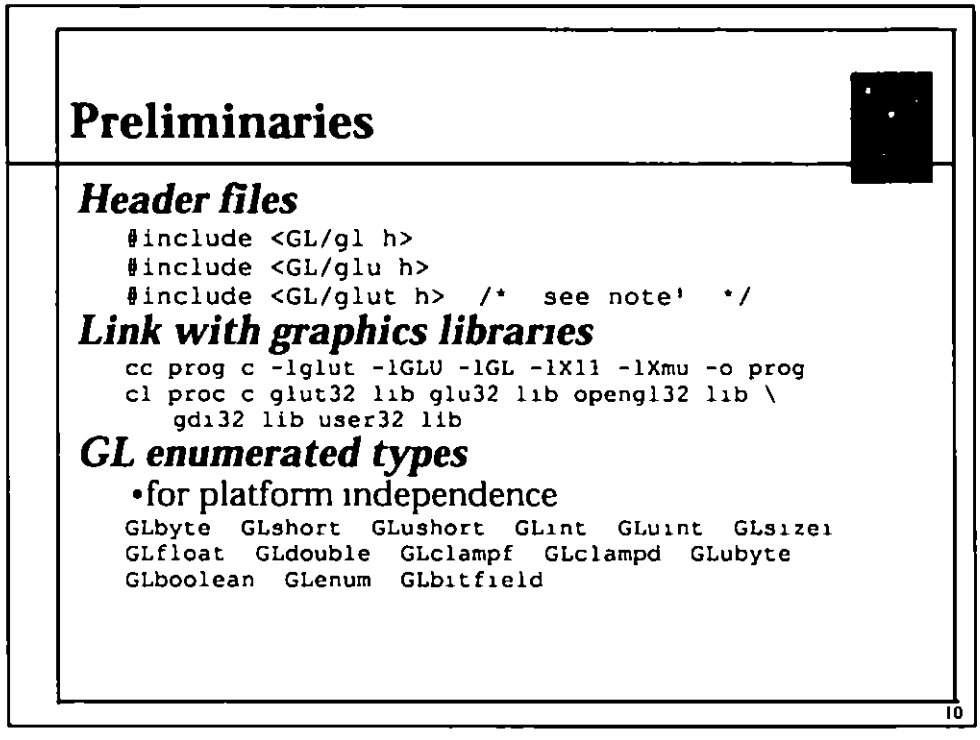
Program Structure

initialize visual & open window
initialize states & display lists
register display callback function
- clear screen change states render graphics swap buffers

register reshape callback function
- modify clipping viewing

register input device (mouse, keybd) callback functions
register idle callback function (the *keep busy* operation)
enter main loop
- if contents need to be redrawn display callback called
- if window resized reshape callback called
- if input event appropriate input callback function called
- if nothing happening idle callback function called

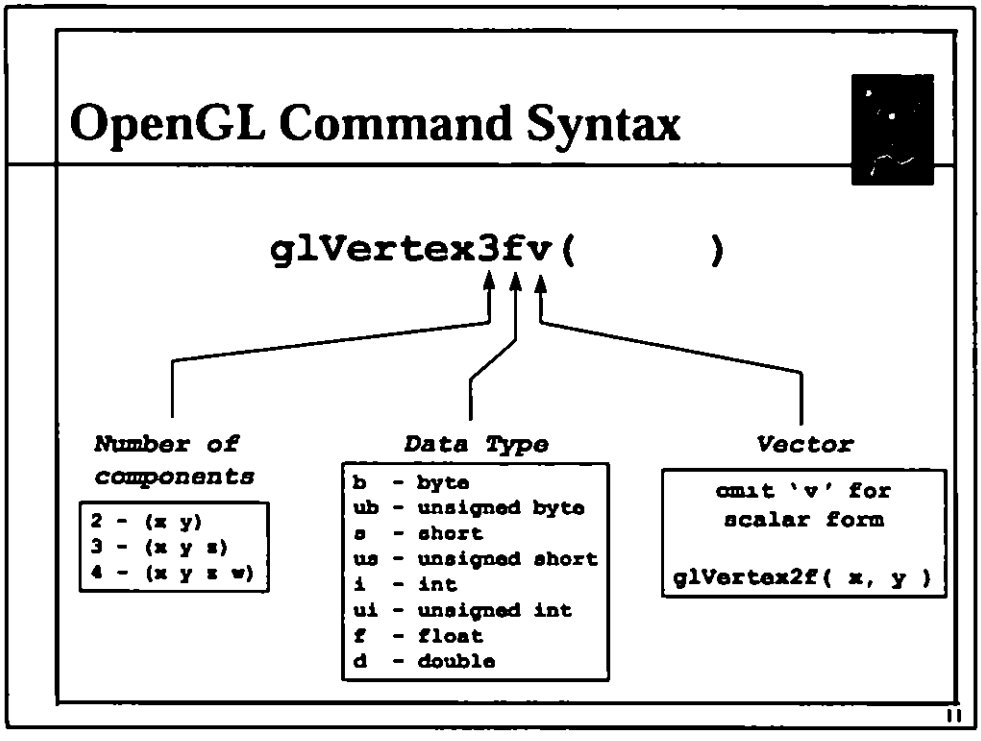
9



Note that including glut h automatically includes both gl h and glu h Also, for Microsoft Windows, glut h includes windows h along with gl h and glu h, so that no compiler errors or warnings result Therefore, if you use glut h, it is recommended that you *don t* redundantly include gl h and glu h again

For Microsoft Windows, be sure the INCLUDE and LIB environment variables are pointing to correct path One common setting is

```
set INCLUDE=c \msdev\include
set LIB=c \msdev\lib
```



The OpenGL API calls are designed to accept almost any basic data type, which is reflected in the call's name Knowing how the calls are structured makes it easy to determine which call should be used for a particular data format and size

For instance, vertices from most commercial models are stored as three component, single-precision, floating point vectors As such, the appropriate OpenGL command to use is `glVertex3fv( coords )`

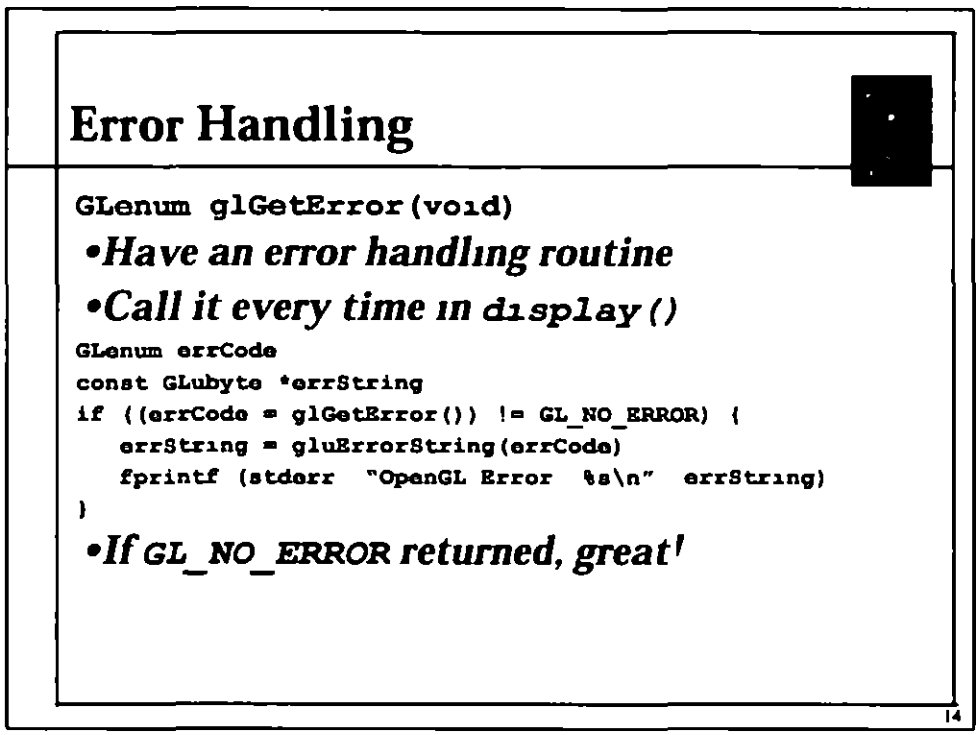
OpenGL uses homogenous coordinates to specify vertices For `glVertex*()` calls which don't specify all the coordinates (i e `glVertex2f()`), OpenGL defaults to z = 0 0, and w = 1 0

A Simple Example Program

```
#include <GL/glut h>
void display(void) {
  glClear(GL_COLOR_BUFFER_BIT)
  glColor3f(0 0, 1 0, 1 0)     /* cyan */
  glBegin(GL_QUADS)
  glVertex2i(100,100)
  glVertex2i(200, 100)
  glVertex2i(200 300)
  glVertex2i(100, 300)
  glEnd()
  glFlush()
}
void gfxinit(void) {
  glClearColor(0 0, 0 0, 0 0, 0 0)
}
```

Simple Program (continued)

```
void reshape(int width, int height) {
  glViewport(0, 0, width, height)
  glMatrixMode(GL_PROJECTION)
  glLoadIdentity()
  gluOrtho2D(0 0, (GLdouble) width, 0 0,
      (GLdouble) height)
}
void main(int argc, char **argv){
  glutInit(&argc, argv),
  glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB)
  win = glutCreateWindow("rect"),
  glutReshapeFunc(reshape),
  glutDisplayFunc(display)
  gfxinit()
  glutMainLoop(),
}
```



`glGetError()` can also return `GL_STACK_OVERFLOW`, `GL_STACK_UNDERFLOW`, `GL_INVALID_VALUE`, `GL_INVALID_OPERATION`, `GL_INVALID_ENUM`, or `GL_OUT_OF_MEMORY`

`gluErrorString()` converts the returned error into something printable

There is also `gluGetError()` for error conditions in the GLU

For a typical, single-threaded OpenGL implementation, only one error is recorded If multiple errors occur, only the first error is recorded For distributed implementations (such as multi-threaded), there may be several errors recorded

Elementary Rendering

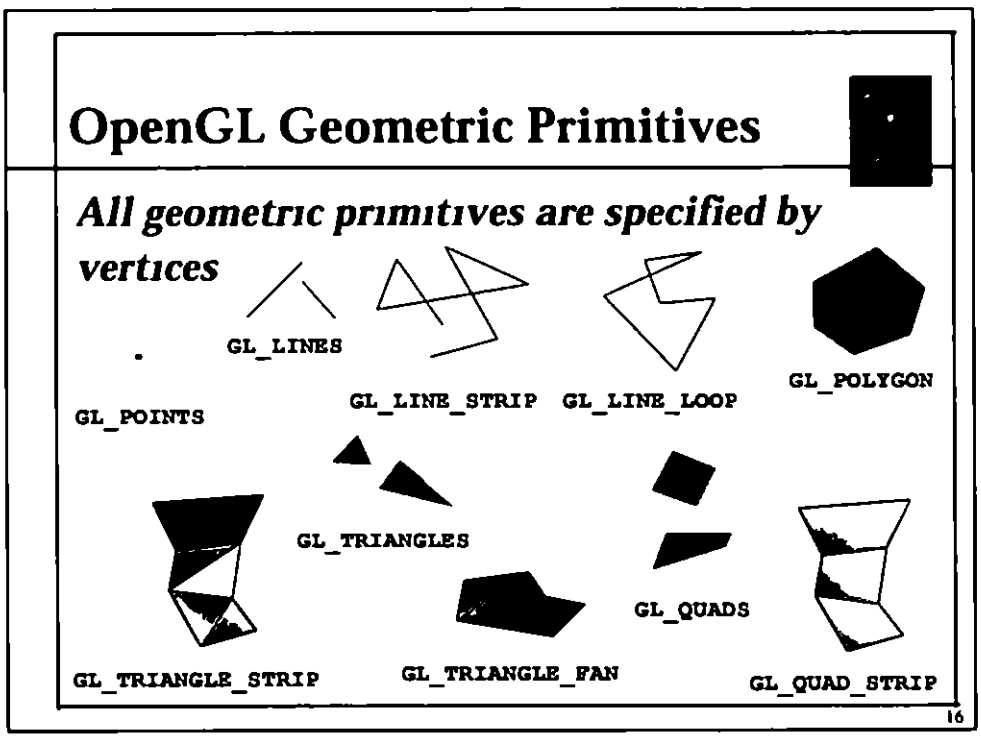
Geometric Primitives
Managing OpenGL State
OpenGL Buffers

15

In this section, we'll be discussing the basic geometric primitives that OpenGL uses for rendering, as well as how to manage the OpenGL state which controls the appearance of those primitives

OpenGL also supports the rendering of bitmaps and images, which is discussed in a later section

Additionally, we'll discuss the different types of OpenGL buffers, and what each can be used for



Every OpenGL geometric primitive is specified by its vertices, which are *homogenous coordinates* Homogenous coordinates are of the form (x, y, z, w) Depending on how vertices are organized, OpenGL can render any of the shown primitives

Specifying Geometric Primitives

Primitives are specified using

```
glBegin( primType ),
glEnd(),
```

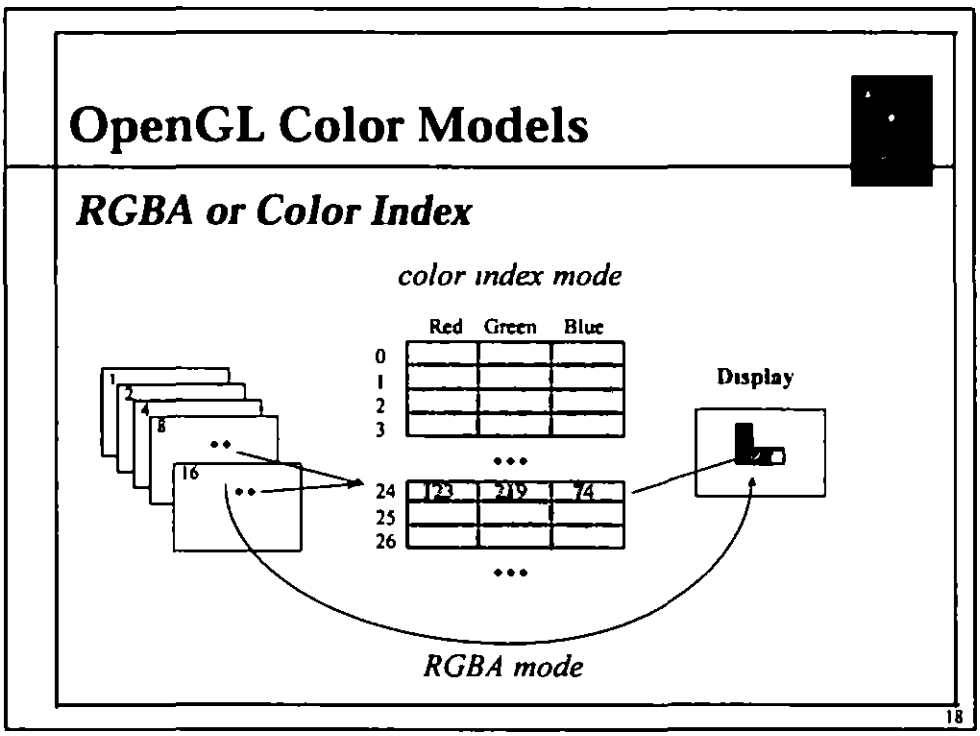
- *primType* determines how vertices are combined

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

17

OpenGL organizes vertices into primitives based upon which type is passed into `glBegin()` The possible types are

`GL_POINTS`	`GL_LINE_STRIP`
`GL_LINES`	`GL_LINE_LOOP`
`GL_POLYGON`	
`GL_TRIANGLE_STRIP`	
`GL_TRIANGLES`	`GL_TRIANGLE_FAN`
`GL_QUADS`	`GL_QUAD_STRIP`



Every OpenGL implementation must support rendering in both RGBA mode, (sometimes described as *TrueColor* mode) and color index (or *colormap*) mode

For RGBA rendering, vertex colors are specified using the `glColor*()` call

For color index rendering, the vertex's index is specified with `glIndex*()`

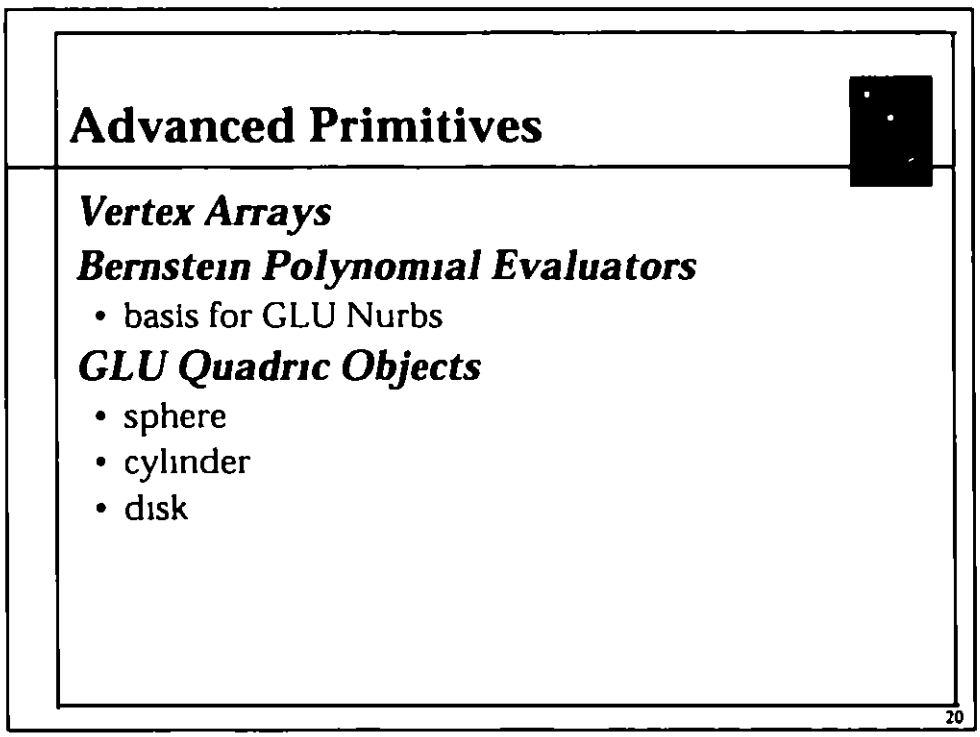
The type of window color model is requested from the windowing system Using GLUT, the `glutInitDisplayMode()` call is used to specify either an RGBA window (using `GLUT_RGBA`), or a color indexed window (using `GLUT_INDEX`)

Simple Example

```
void drawRhombus( GLfloat color[] )
{
  glColor3fv( color ),
  glBegin( GL_QUADS ),
  glVertex2f( 0 0, 0 0 ),
  glVertex2f( 1 0, 0 0 ),
  glVertex2f( 1 5, 1 118 ),
  glVertex2f( 0 5, 1 118 ),
  glEnd(),
}
```

19

The `drawRhombus()` routine causes OpenGL to render a single quadrilateral in a single color The rhombus is planar, since the z value is automatically set to 1 0 by `glVertex2f()`



In addition to specifying vertices one at a time using `glVertex*()`, OpenGL supports the use of arrays, which allows you to pass an array of vertices, lighting normals, colors, edge flags, or texture coordinates This is very useful for systems where functions calls are computationally expensive Additionally, the OpenGL implementation may be able to optimize the processing of arrays

OpenGL evaluators, which automate the evaluation of the Bernstein polynomials, allow curves and surfaces to be expressed algebraically They are the underlying implementation of the OpenGL Utility Library's NURBS implementation

Finally, the OpenGL Utility Library also has calls for generating polygonal representation of quadric objects The calls can also generate lighting normals and texture coordinates for the quadric objects

Vertex Arrays

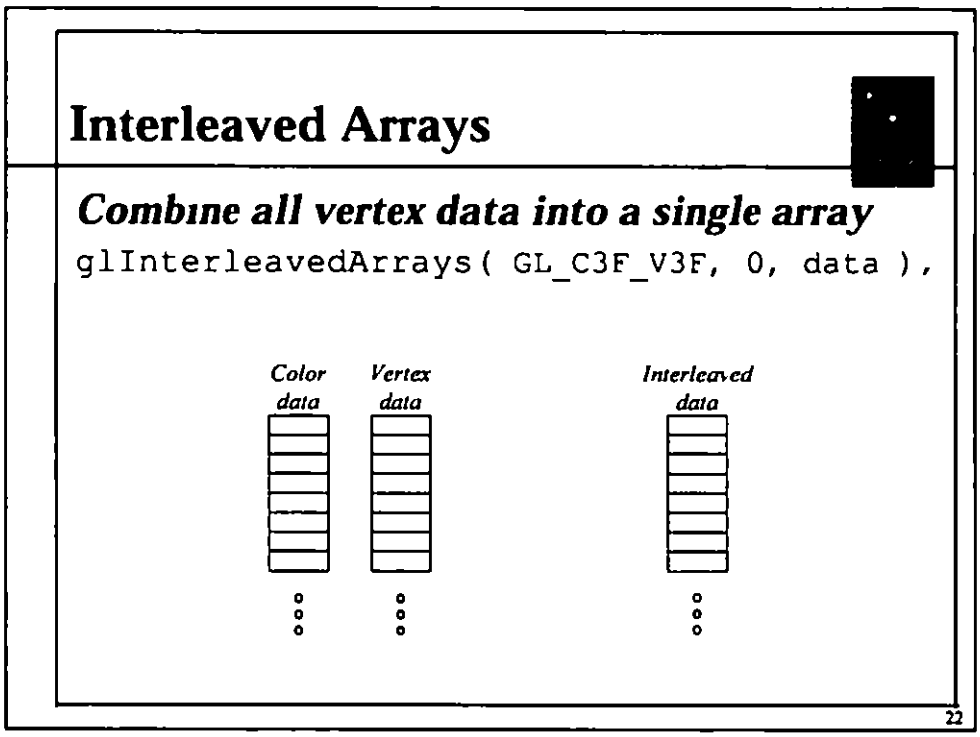
Pass arrays of vertices, colors, etc to OpenGL in a large chunk

```
glVertexPointer( 3, GL_FLOAT, 0, coords)
glColorPointer( 4, GL_FLOAT, 0, colors )
glEnableClientState( GL_VERTEX_ARRAY )
glEnableClientState( GL_COLOR_ARRAY )
```

All active arrays are used in rendering

21

Vertex Arrays allow vertices, and their attributes to be specified in chunks, which reduces the need for sending single vertices and their attributes one call at a time This is a useful optimization technique, as well as usually simplifying storage of polygonal models



`glInterleavedArrays()` combines all the information for a vertex together locally in memory When the array is traversed, the *format* (`GL_C3F_V3F` in the above example) tells OpenGL what data, and how it should be processed, are stored in the array

The supported formats for `glInterleavedArrays()` are

`GL_V2F`	`GL_C4UB_V2F`	`GL_C3F_V3F`
`GL_V3F`	`GL_C4UB_V3F`	`GL_N3F_V3F`
`GL_C4F_N3F_V3F`	`GL_T2F_C4UB_V3F`	`GL_T2F_V3F`
`GL_T2F_C3F_V3F`	`GL_T2F_N3F_V3F`	`GL_T4F_V4F`

`GL_T2F_C4F_N3F_V3F` `GL_T4F_C4F_N3F_V4F`

where the letters below represent the following vertex data

V - Vertex coordinates

C - Color information

N - Normal vectors

T - Texture coordinates

Rendering with Vertex Arrays

Render arrays sequentially

```
glDrawArrays( GL_TRIANGLE_STRIP, 0, numVerts )
```

Automatically index into arrays

```
GLuint indices[] = { 0, 2, 1, 3, 2, 3, 6, 5 },
glDrawElements( GL_QUADS, 2, GL_UNSIGNED_INT,
  indices )
```

Manually index into arrays

```
glBegin( GL_LINES ),
for ( i = 0, i < numLines  ++i ) {
   glArrayElement( leftEnd[i] )
   glArrayElement( rightEnd[i] ),
}
glEnd()
```

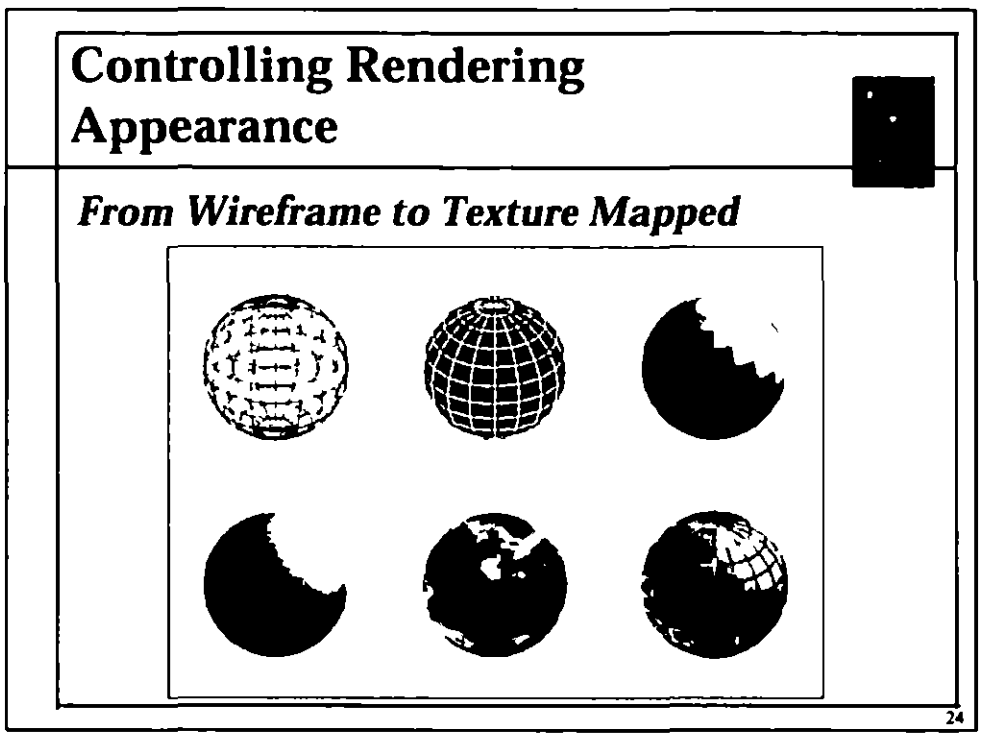
23

When OpenGL processes the arrays, any enabled array is used for rendering There are three methods for rendering using vertex arrays

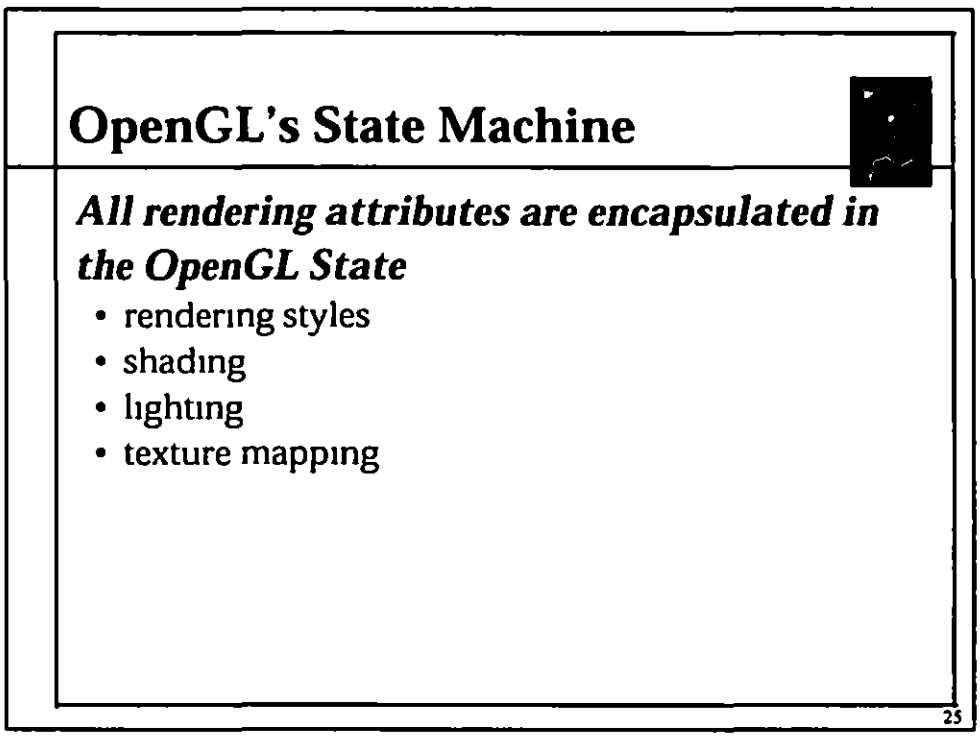
First, the `glDrawArrays()` routine will render the specified primitive type by processing *numVerts* consecutive data elements from the enabled arrays

Second, `glDrawElements()` allows indirect indexing of data elements in the enabled arrays This allows shared data elements to be specified only once in the arrays, but be accessed numerous times

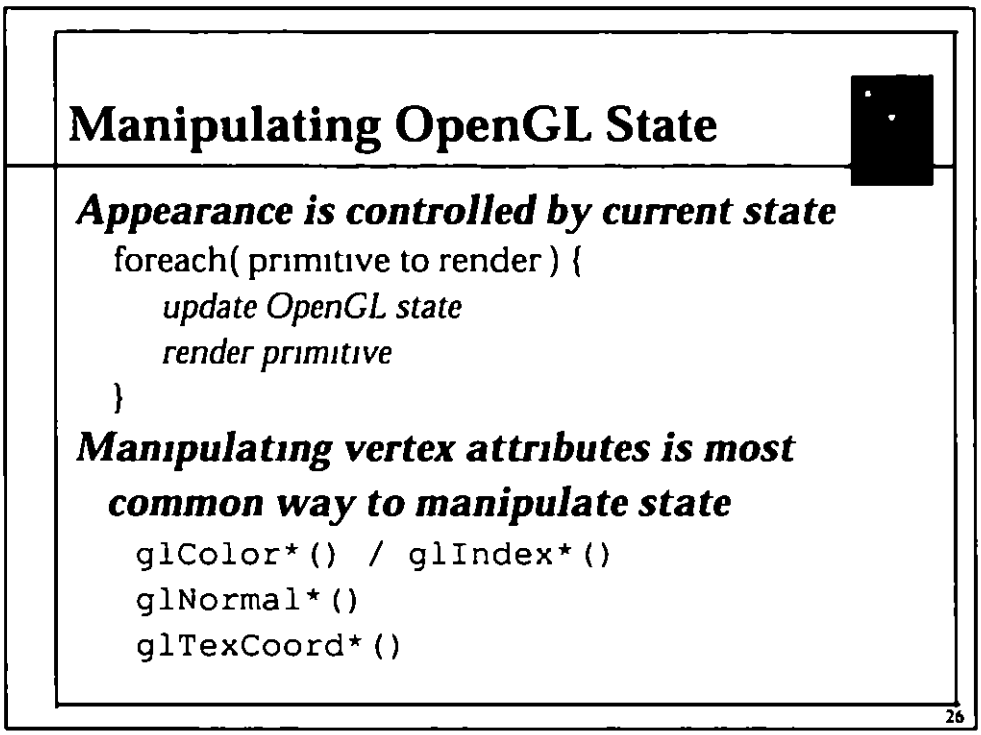
Finally, `glArrayElement()` processes a single set of data elements from all activated arrays As compared to the previous two commands above, `glArrayElement()` must appear between a `glBegin()` / `glEnd()` pair



OpenGL can render from a simple line-based wireframe to complex multi-pass texturing algorithms to simulate bump mapping or Phong lighting



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



The general flow of any OpenGL rendering is to set up the required state, then pass the primitive to be rendered, and repeat for the next primitive

In general, the most common way to manipulate OpenGL state is by setting vertex attributes, which include color, lighting normals, and texturing coordinates

Controlling current state

Setting State

```
glPointSize( size ),
glLineStipple( repeat, pattern ),
glMaterialfv( GL_FRONT, GL_DIFFUSE,
  color ),
```

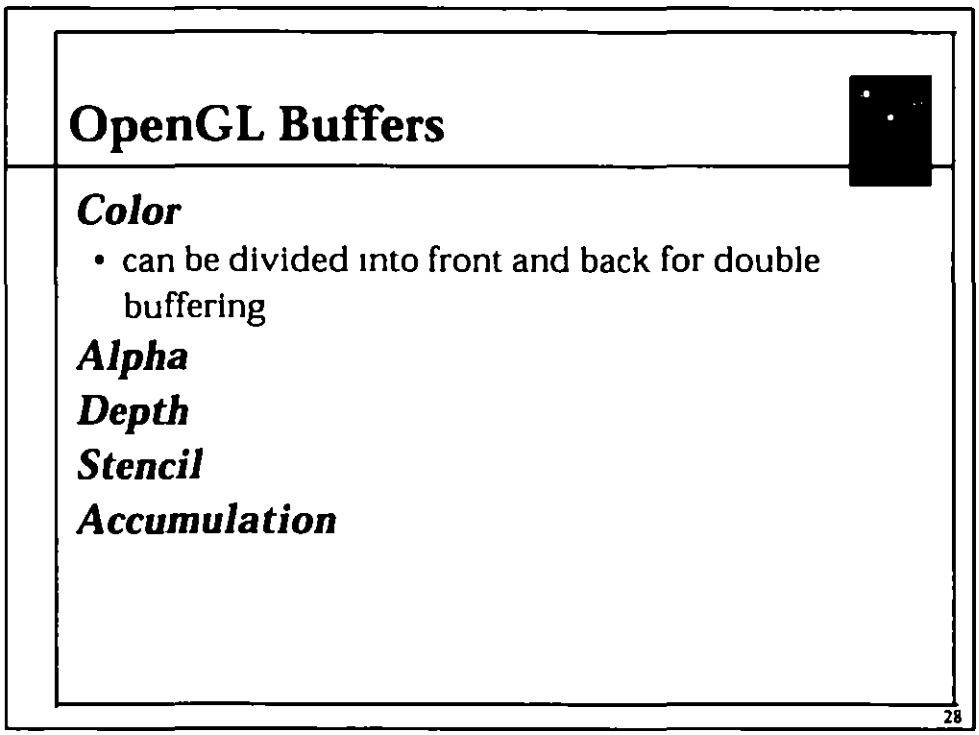
Enabling Features

```
glEnable( GL_LIGHTING ),
glDisable( GL_TEXTURE_2D ),
```

27

Setting OpenGL state usually includes modifying the rendering attribute, such as loading a texture map, or setting the line width Also for some state changes, setting the OpenGL state also enables that feature (like setting the point size or line width)

Other features need to be turned on This is done using `glEnable()`, and passing the token for the feature, like `GL_LIGHT0` or `GL_POLYGON_STIPPLE`



OpenGL supports a variety of different buffers, some of which hold color data, and others which control the updating of that color data

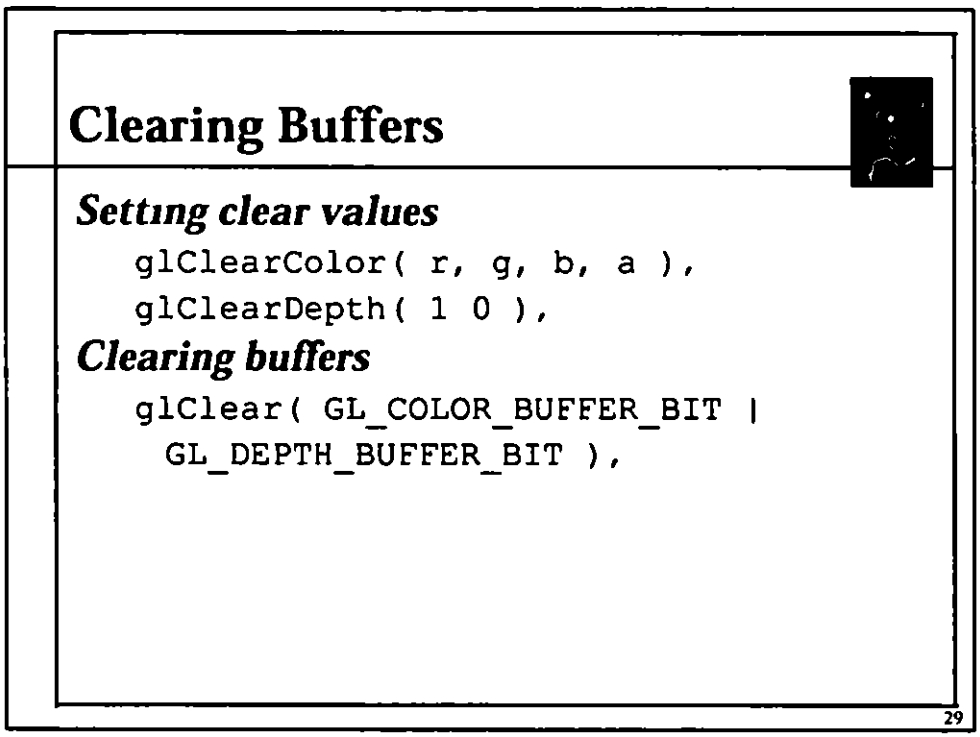
The principle OpenGL buffer is the *color buffer*, which is generally the hardware framebuffer, or an offscreen rendering pixmap

The *depth buffer*, which is also called the *z*-buffer, is used for visible surface determination

The *stencil buffer* provides a per-pixel mask test to determine if a pixel in the color buffer should be updated

Finally, the *accumulation buffer* is used for the compositing of several renderings from the color buffer, which can be scaled and combined together to produce a final image which is transferred back to the color buffer for display

The stencil and accumulation buffers are somewhat advanced topics, and are outside the scope of this course



The `glClear*()` commands are used to set the default values used to clear each of a windows active buffers to When the `glClear()` call is executed, it clears each of the buffers requested with their clear values

The commands for setting the default clear values are

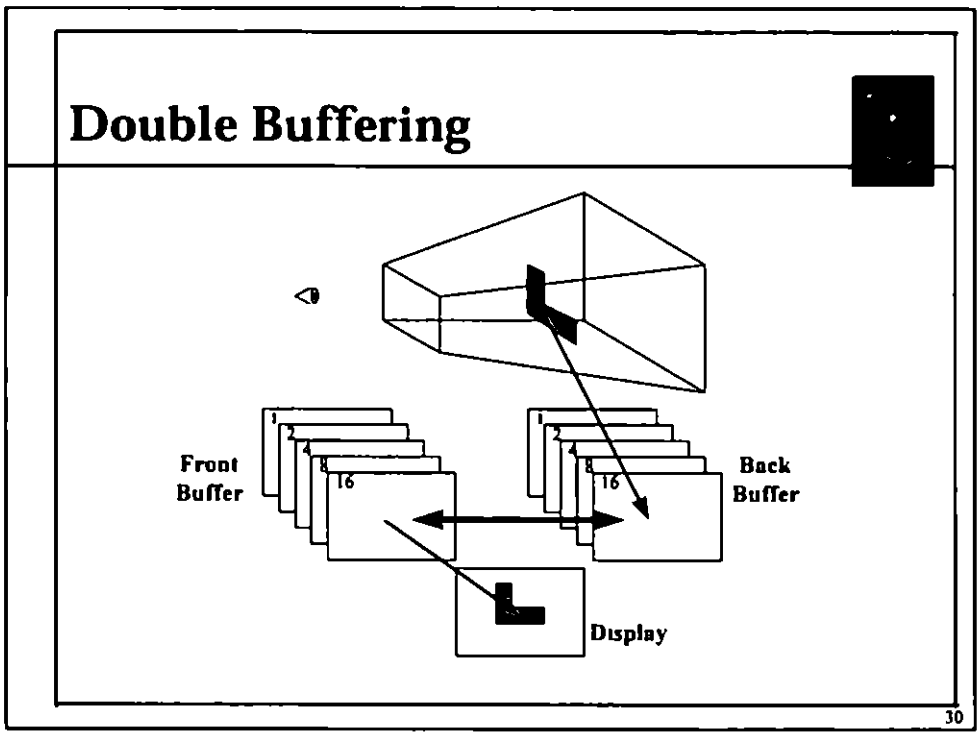
`glClearColor()` - set default RGBA value for color buffer (RGBA mode)

`glClearIndex()` - set default color index for color buffer (Color index mode)

`glClearDepth()` - set default depth value for depth buffer

`glClearAccum()` - set default color for accumulation buffer

`glClearStencil()` - set default value for stencil buffer



Double buffer is a technique for tricking the eye into seeing smooth animation of rendered scenes The color buffer is usually divided into two equal halves, called the *front buffer* and the *back buffer*

The front buffer is displayed while the application renders into the back buffer When the application completes rendering to the back buffer, it requests the graphics display hardware to swap the roles of the buffers, causing the back buffer to now be displayed, and the previous front buffer to become the new back buffer

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

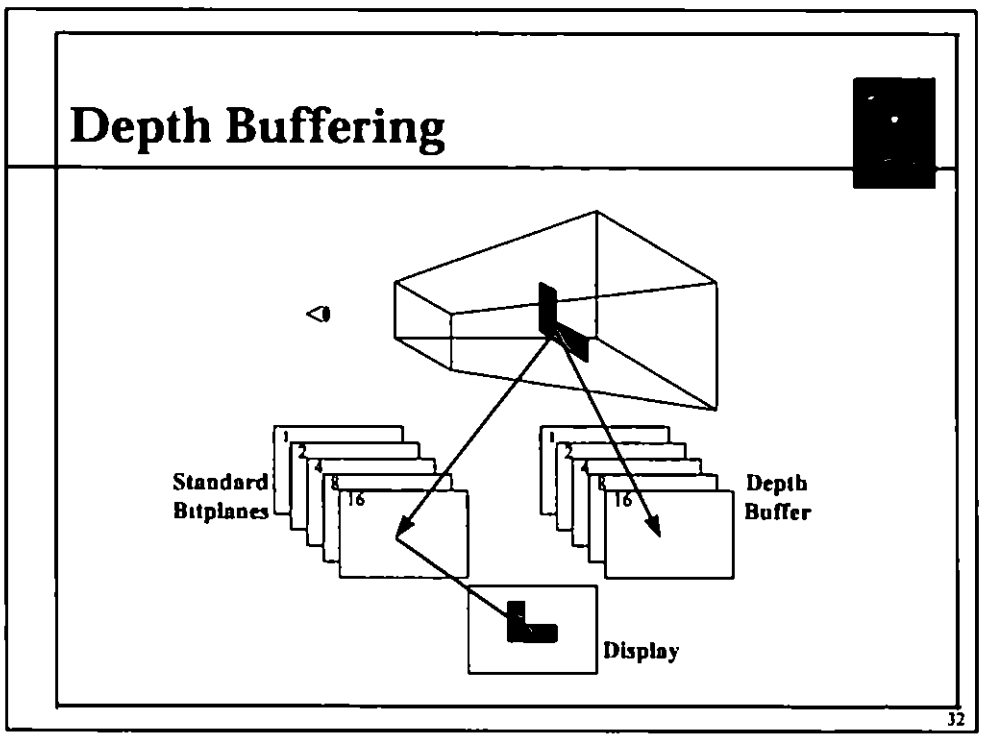
```
glutSwapBuffers(),
```

Repeat steps 2-4 for animation

31

Requesting double buffering in GLUT is simple In addition to specifying what type of color model you would like to use, adding `GLUT_DOUBLE` to your `glutInitDisplayMode()` call will cause your window to be double buffered

When your application is finished rendering its current frame, and wants to swap the front and back buffers, the `glutSwapBuffers()` call will request the windowing system to update the window's color buffers



Depth buffering is a technique for determine which primitives in your scene are occluded by other primitives As each pixel in a primitive is rasterized, its distance from the eyepoint (depth value), is compared with the values stored in the depth buffer If the pixel's depth value is less than the stored value, the pixel's depth value is written to the depth buffer, and its color is written to the color buffer

The depth buffer algorithm is generally

```
if ( pixel->z < depthBuffer(x,y)->z ) {
     depthBuffer(x,y)->z = pixel->z,
     colorBuffer(x,y)->color = pixel->color,
  }
```

OpenGL depth values range from [0, 1], with one being essientially infinitely from the eyepoint Generally, the depth buffer is cleared to one at the start of a frame

Depth Buffering Using OpenGL

1) Request a depth buffer

```
glutInitDisplayMode( GLUT_RGB |
  GLUT_DOUBLE | GLUT_DEPTH ),
```

2) Enable depth buffering

```
glEnable( GL_DEPTH_TEST ),
```

3) Clear color and depth buffers

```
glClear( GL_COLOR_BUFFER_BIT |
  GL_DEPTH_BUFFER_BIT ),
```

4) Render scene

5) Swap Buffers

33

Enabling depth testing in OpenGL is very straightforward

A depth buffer must be requested for your window, once again using the `glutInitDispayMode()`, and the `GLUT_DEPTH` bit

Once the window is created, the depth test is enabled using `glEnable( GL_DEPTH_TEST )`

A Complete Example

```
void main( int argc, char** argv )
{
   glutInit( &argc, argv ),
   glutInitDisplayMode( GLUT_RGB |
      GLUT_DOUBLE | GLUT_DEPTH ),
   glutCreateWindow( "Tetrahedron" ),
   init(),
   glutIdleFunc( drawScene ),
   glutMainLoop(),
}
```

34

In `main()`,

1) GLUT initializes and creates a window named "Tetrahedron"
2) set OpenGL state which is enabled through the entire life of the program in

   `init()`

3) set GLUT's idle function, which is used executed when there are no user user events to process
4) enter the main event processing loop of the program

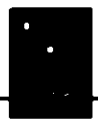
A Complete Example (cont)

```
void init( void )
{
   GLfloat verts[][3] = {    },
   GLfloat colors[][3] = {    },
   glClearColor( 1 0, 0 0, 1 0, 1 0 ),
   glVertexPointer( 3, GL_FLOAT, 0, verts ),
   glColorPointer( 3, GL_FLOAT, 0, colors ),
   glEnableClientState( GL_VERTEX_ARRAY ),
   glEnableClientState( GL_COLOR_ARRAY ),
}
```

35

In `init()` the basic OpenGL state to be used throughout the program is initialized Since vertex arrays are used, they are set and enabled in `init()`, and referenced when rendering occurs Additionally, we set the background (clear color) for the color buffer

A Complete Example (cont.)



```
void drawScene( void )
{
   GLuint indices[] = { 0,    },
   glClear( GL_COLOR_BUFFER_BIT |
     GL_DEPTH_BUFFER_BIT ),
   glDrawElements( GL_TRIANGLE_STRIP, 6,
     GL_UNSIGNED_INT, indices ),
   glutSwapBuffers(),
}
```

36

In `drawScene()`,

1) the color buffer is cleared to the background color
2) a triangle strip is rendered to create a tetrahedron using `glDrawElements()` and the vertex arrays that were set up in `init()`
3) the front and back buffers are swapped

Transformations

Prior to rendering, view, locate, and orient

- eye/camera position
- 3D geometry

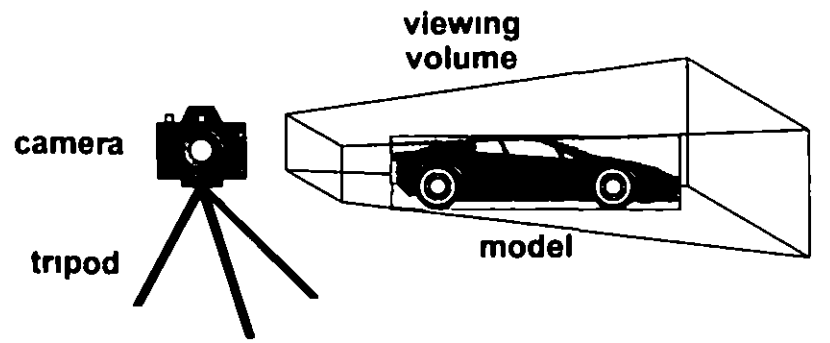
Manage the matrices

- including matrix stack

Combine (composite) transformations

Camera Analogy

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



3D Mathematics

A vertex is transformed by matrices

- each vertex is a column vector $\mathbf{v}$ $(x\ y\ z\ w)^T$ where w is usually 1 0
- all operations are matrix multiplications
- all matrices are column major
- matrices are always post multiplied
- product of matrix and vector is **Mv**

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

Programmer does not have to remember the exact matrices

- check appendix of Red Book (Programming Guide)

39

A 3D vertex is represented by a 4-tuple vector (homogeneous coordinate system)

Why is a 4-tuple vector (and a 4 by 4 matrix) used for a 3D (x, y, z) vertex? To ensure that all matrix operations are multiplications Perspective projection and translation require 4th row and column, or operations would require addition, as well as multiplication

Camera Analogy

Projection transformations

adjust the lens of the camera

Viewing transformations

tripod–define position and orientation of the viewing volume in the world

Modeling transformations

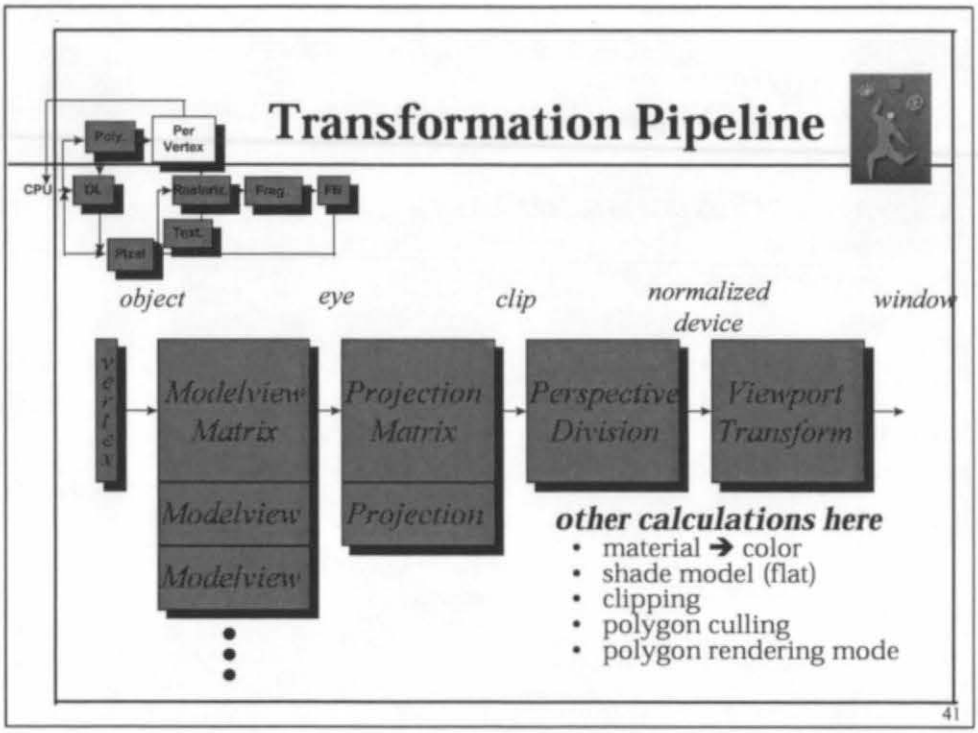
- moving the model

Viewport transformations

- enlarge or reduce the physical photograph

40

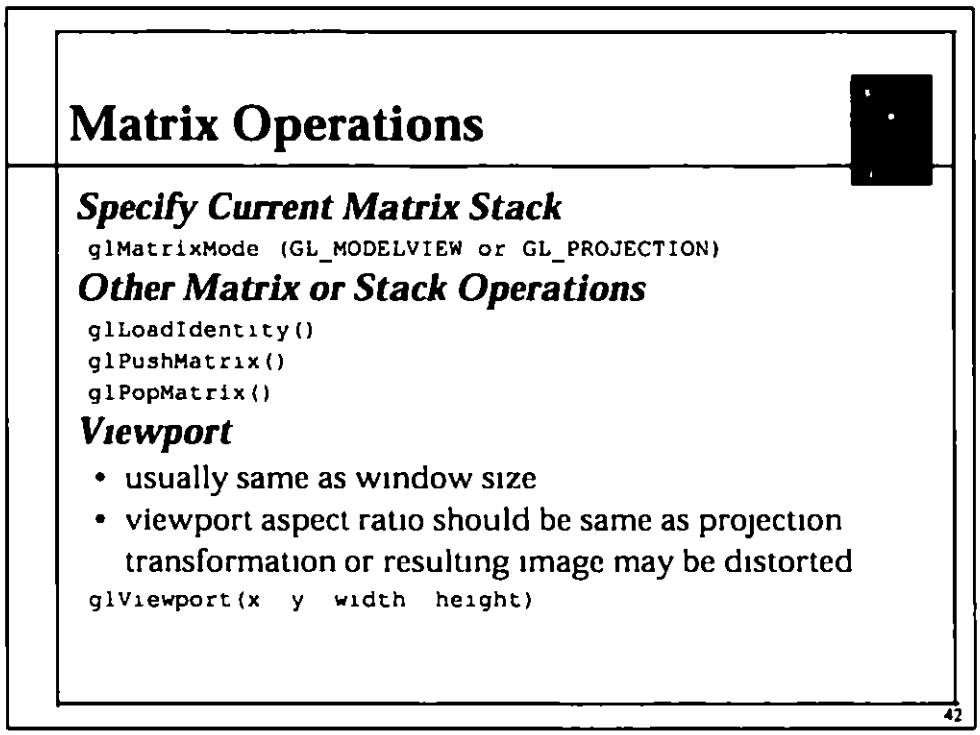
Note that human vision and a camera lens have cone-shaped viewing volumes OpenGL (and almost all computer graphics APIs) describe a pyramid-shaped viewing volume Therefore, the computer will "see" differently from the natural viewpoints, especially along the edges of viewing volumes This is particularly pronounced for wide-angle "fish-eye" camera lenses



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

The material-to-color, flat-shading, and clipping calculations take place after the Modelview matrix calculations, but before the Projection matrix. The polygon culling and rendering mode operations take place after the Viewport operations.

There is also a texture matrix stack, which is not mentioned here. It is an advanced texture mapping topic.



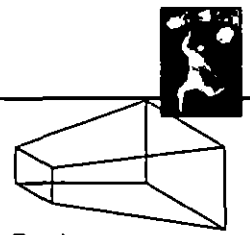
Also `glLoadMatrix{df}(matrix)` and `glMultMatrix{df}(matrix)`, where matrix is a column-major 4 x 4 double or single precision floating point matrix The matrix is either loaded or post-multiplied onto the top of the current matrix stack

The stacks are used, because it is more efficient to save and restore matrices than to calculate and multiply new matrices Popping a matrix stack can be said to "jump back" to a previous location or orientation

`glViewport` clips the vertex or raster position For geometric primitives, a new vertex may be created For raster primitives, the raster position is completely clipped

There is a per-fragment operation, the scissor test, which works in situations where viewport clipping doesn't The scissor operation is particularly good for fine clipping raster (bitmap or image) primitives

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*

43

For perspective projections, the viewing volume is shaped like truncated pyramid (frustum) There is a distinct camera (eye) position, and vertexes of objects are "projected" to camera Objects which are further from the camera appear smaller The default camera position at (0, 0, 0), looking down z-axis, although the camera can be moved by other transformations

For `gluPerspective`, `fovy` is the angle of field of view (in degrees) in the y direction `fovy` must be between 0 0 and 180 0, exclusive `aspect` is x/y and should be same as the viewport to avoid distortion `zNear` and `zFar` define the distance to the near and far clipping planes

`glFrustum` is rarely used Warning for `gluPerspective` or `glFrustum`, don't use zero for `zNear`!

For `glOrtho()`, the viewing volume is shaped like a rectangular parallelepiped (a box) Vertexes of an object are "projected" towards infinity Distance does not change the apparent size of an object Orthographic projection is used for drafting and design (such as blueprints)

Viewing Transformations

Position the camera/eye in the scene

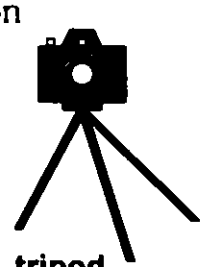
- place the tripod down aim camera

To "fly through" a scene

- change viewing transformation and redraw scene

```
gluLookAt(eyex eyey eyez, aimx, aimy aimz upx upy upz)
```

- up vector determines unique orientation
  careful of degenerate positions



tripod

44

`gluLookAt()` multiplies itself onto the current matrix, so it usually comes after `glMatrixMode(GL_MODELVIEW)` and `glLoadIdentity()`

Because of degenerate positions, `gluLookAt()` is not recommended for most animated fly-over applications

Modeling Transformations

Move object

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

Rotate object around arbitrary axis (x, y, z)

```
glRotate{fd}(angle, x, y, z)
```

angle is in degrees

Dilate (stretch or shrink) or mirror object

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

45

`glTranslate()`, `glRotate()`, and `glScale()` multiplies itself onto the current matrix, so it usually comes after `glMatrixMode(GL_MODELVIEW)` There are many situations where the modeling transformation is multiplied onto a non-identity matrix

A vertex's distance from the origin changes the effect of `glRotate()` or `glScale()` Generally, the further from the origin, the more pronounced the effect

Connection: Viewing and Modeling

- ***Moving camera is equivalent to moving every object in the world towards a stationary camera***
- ***Viewing transformations are equivalent to several modeling transformations***
  - `gluLookAt` has its own command
  - can make your own *polar view* or *pilot view*

46

Instead of `gluLookAt`, one can use the following combinations of `glTranslate` and `glRotate` to achieve a viewing transformation Like `gluLookAt`, these transformations should be multiplied onto the ModelView matrix, which should have an initial identity matrix

To create a viewing transformation in which the viewer orbits an object, use this sequence (which is known as "polar view")

```
glTranslated (0, 0, -distance)
glRotated (-twist, 0, 0, 1)
glRotated (-incidence, 1, 0, 0)
glRotated (azimuth, 0, 0, 1)
```

To create a viewing transformation which orients the viewer (roll, pitch, and heading) at position (x, y, z), use this sequence (known as "pilot view")

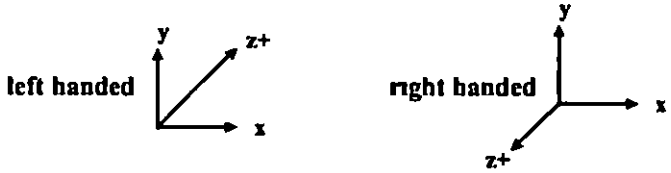
```
glRotated (roll, 0, 0, 1)
glRotated (pitch, 0, 1, 0)
glRotated (heading, 1, 0, 0)
glTranslated (-x, -y, -z)
```

Projection is left handed

Projection transformations (`gluPerspective`, `glOrtho`) are left handed

- think of `zNear` and `zFar` as distance from view point

Everything else is right handed, including the vertexes to be rendered



Common Transformation Usage

3 examples of `resize()` routine

- restate projection & viewing transformations

Usually called when window resized

Registered as callback for `glutReshapeFunc()`

resize(): Perspective & LookAt

```
void resize (int w, int h) {
   glViewport (0, 0, (GLsizei) w, (GLsizei) h)
   glMatrixMode (GL_PROJECTION)
   glLoadIdentity ()
   gluPerspective (65 0, (GLfloat) w/(GLfloat) h,
                   1 0, 100 0),
   glMatrixMode (GL_MODELVIEW),
   glLoadIdentity (),
   gluLookAt (0 0, 0 0, 5 0, 0 0, 0 0, 0 0,
              0 0, 1 0, 0 0)
}
```

49

Using the viewport width and height as the aspect ratio for `gluPerspective` eliminates distortion

resize() Perspective & Translate

Same effect as previous LookAt

```
void resize (int w  int h) {
   glViewport (0  0, (GLsizei) w, (GLsizei) h)
   glMatrixMode (GL_PROJECTION)
   glLoadIdentity ()
   gluPerspective (65 0, (GLfloat) w/(GLfloat) h,
                   1 0, 100 0)
   glMatrixMode (GL_MODELVIEW)
   glLoadIdentity ()
   glTranslatef (0 0, 0 0, -5 0)
}
```

50

Moving all objects in the world five units away from you is mathematically the same as "backing up" the camera five units

resize(): Ortho

```
void resize (int w, int h) {
   glViewport (0  0, (GLsizei) w, (GLsizei) h)
   glMatrixMode (GL_PROJECTION)
   glLoadIdentity ()
   if (w <= h)
      glOrtho (-2 5, 2 5, -2 5*(GLfloat)h/(GLfloat)w,
        2 5*(GLfloat)h/(GLfloat)w, -10 0, 10 0)
   else
      glOrtho (-2 5*(GLfloat)w/(GLfloat)h,
        2 5*(GLfloat)w/(GLfloat)h  -2 5, 2 5  -10 0, 10 0)
   glMatrixMode (GL_MODELVIEW)
   glLoadIdentity ()
}
```

51

The two `glOrtho` calls are needed to accommodate for different aspect ratios, maintaining a minimum viewable region

Compositing Modeling Transformations

Problem 1 hierarchical objects

- one position depends upon a previous position
- robot arm or hand sub-assemblies

Solution 1 moving local coordinate system

- modeling transformations move coordinate system post multiply column-major matrices
- OpenGL post-multiplies matrices

52

The order in which modeling transformations are performed is important because each modeling transformation is represented by a matrix, and matrix multiplication is not commutative So a rotate followed by a translate is different from a translate followed by a rotate

Compositing Modeling Transformations

Problem 2 objects move relative to absolute world origin

my object rotates around the wrong origin

- *make it spin around its center or something else*

Solution 2 fixed coordinate system

modeling transformations move objects around fixed coordinate system

- pre-multiply column-major matrices
  OpenGL post multiplies matrices
  must reverse order of operations to achieve desired effect

53

You'll adjust to reading a lot of code backwards!

Additional Clipping Planes

- *At least 6 more clipping planes available*
- *Good for cross-sections*
- *Modelview matrix moves clipping plane*
- *Ax+By+Cz+D < 0 clipped*

```
glEnable (GL_CLIP_PLANEi)
glClipPlane (GL_CLIP_PLANEi, GLdouble* coeff)
```

Reversing Coordinate Projection

Screen space back to world space

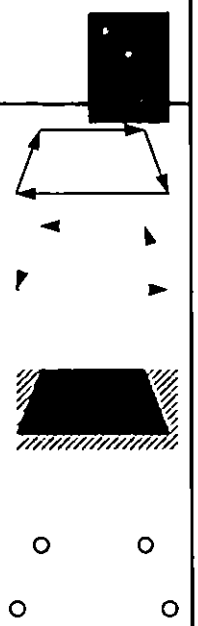
```
glGetIntegerv(GL_VIEWPORT, GLint viewport[4])
glGetDoublev(GL_MODELVIEW_MATRIX, GLdouble mvmatrix[16])
glGetDoublev(GL_PROJECTION_MATRIX, GLdouble projmatrix[16])
gluUnProject(GLdouble winx, winy, winz
        mvmatrix[16], projmatrix[16], GLint viewport[4],
        GLdouble *objx, *objy  *objz)
```

`gluProject` goes from world to screen space

55

Generally, OpenGL projects 3D data onto a 2D screen Sometimes, you need to use a 2D screen position (such as a mouse location) and figure out where in 3D it came from If you use `gluUnProject` with `winz` = 0 and `winz` = 1, you can find the 3D point at the near and far clipping planes Then you can draw a line between those points, and you know that some point on that line was projected to your screen position

Culling and Polygon Mode



Culling

- turn on mode to eliminate rendering of front or back facing polygons
- front and back facing determined by orientation (winding) of polygons

Polygon Mode

- controls how polygons are rendered (filled wire frame or points)
- polygon means `GL_POLYGON`, `GL_QUADS`, `GL_TRIANGLES`, `GL_TRIANGLE_STRIP`, etc

56

`glFrontFace (windingMode)` determines which winding method is front- and back-facing for both culling and polygon mode `windingMode` is `GL_CCW` or `GL_CW` (counter clockwise or clockwise) `GL_CCW` is the default

OpenGL routines for culling include

```
glEnable (GL_CULL_FACE)
glCullFace (whichWay)
```

`whichWay` is `GL_FRONT` or `GL_BACK` (the default)

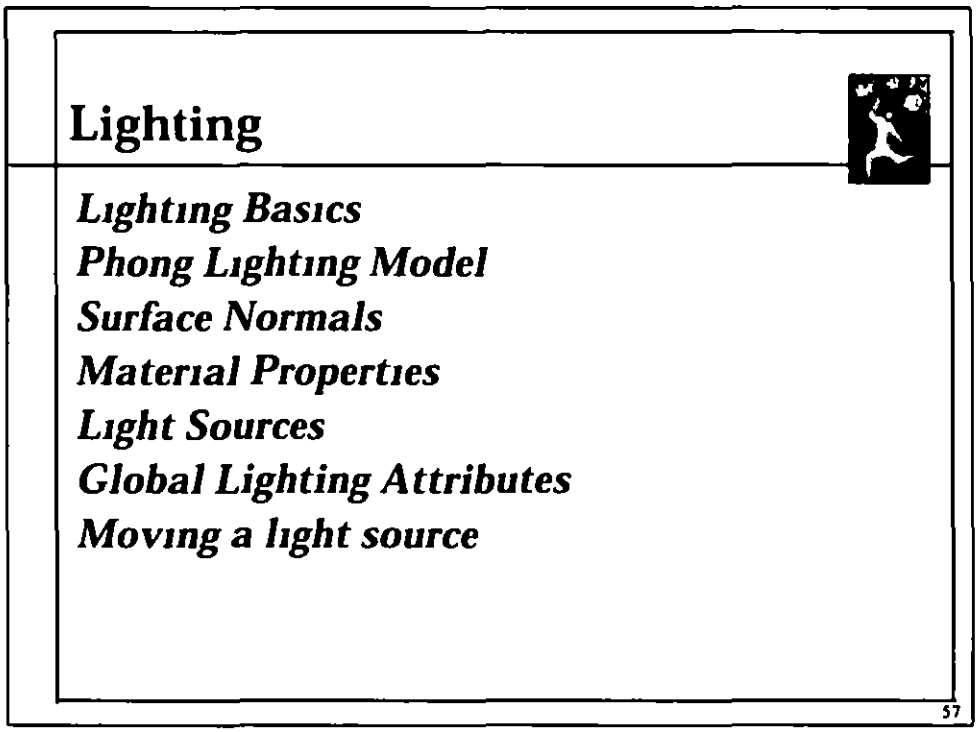
The application (and associated data sets) are responsible for creating the proper winding of the polygons Culling is disabled by default

The OpenGL routine for selecting polygon mode is

```
glPolygonMode (GLenum face, GLenum mode)
```

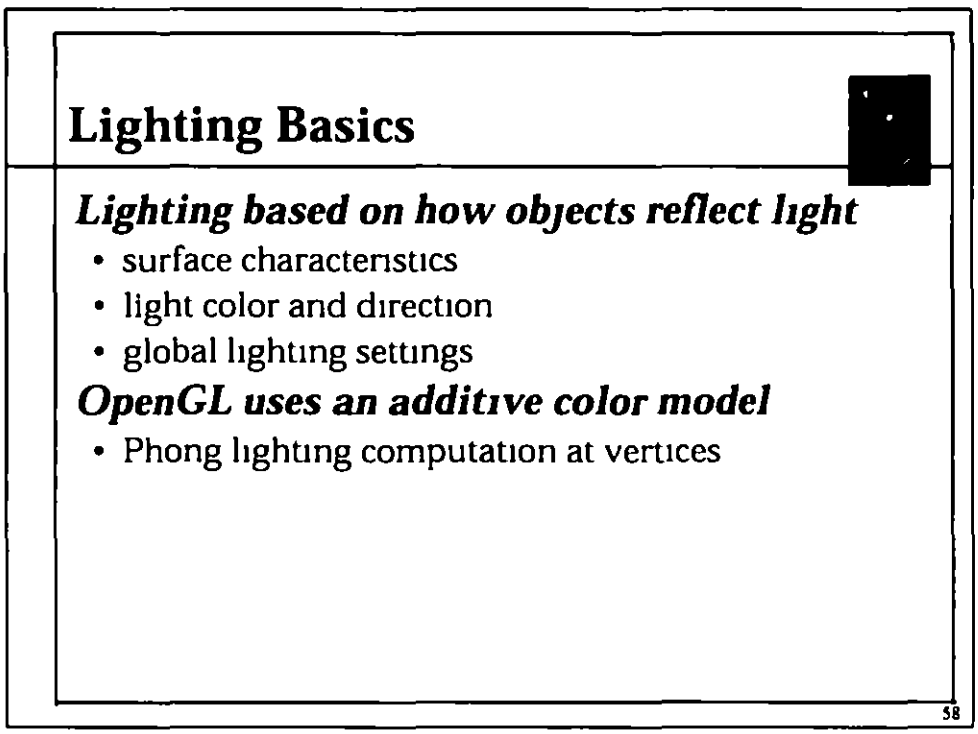
where face is GL_FRONT, GL_BACK, or GL_FRONT_AND_BACK

mode is GL_FILL (default), GL_LINE, or GL_POINT



In this section we discuss OpenGL Lighting, which is based on the Phong lighting model

The lighting model combines an object's material properties and position, the light's color and position, and global lighting parameters to compute a vertex's color



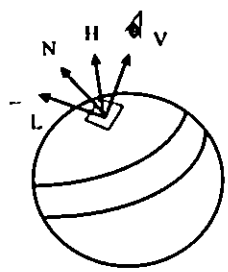
OpenGL uses a Phong based lighting model to compute colors for vertices The color generated is a combination of the object's material properties, the light's color and position, and the global lighting characteristics

Since lighting is used to compute a vertex's color, the tessellation of the object is important The better the tessellation and lighting normals, the better the lighting effects generated

Phong Lighting Model

Color determined by several factors

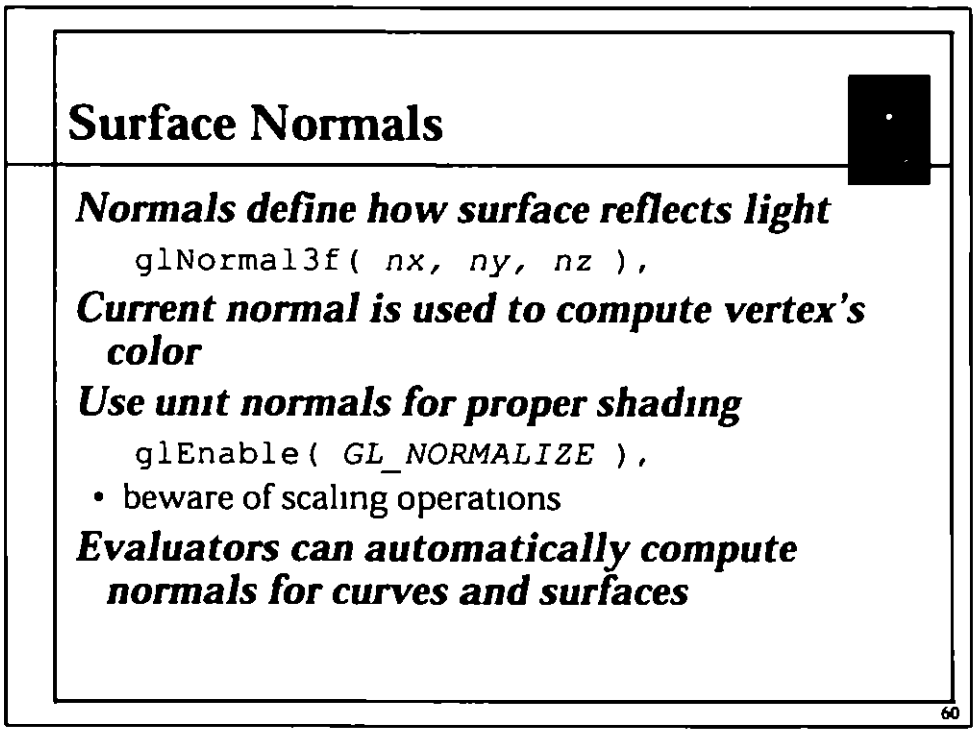
- surface normal and color
- light position
- eye position



59

OpenGL uses an approximation to the Phong lighting model to compute the color of a lit vertex In the diagram above, the following vectors are shown

- V - vector from eyepoint to vertex
- N - vertex lighting normal
- L - vector from vertex to light
- H - 1/2 (V + L)



The lighting normal tells OpenGL how the object reflects light around a vertex If you imagine that there is a small mirror at the vertex, the lighting normal describes how the mirror is oriented, and consequently how light is reflected

`glNormal()` sets the current normal, which is used in the lighting computation for all vertices until a new normal is provided

Lighting normals should be normalized to unit length for correct lighting results `glScale()` affects normals as well as vertices, which can change the normal's length, and cause it to no longer be normalized OpenGL can automatically normalize normals, by enabling `glEnable(GL_NORMALIZE)` Since normalization requires the computation of a square root, it can potentially lower performance

OpenGL evaluators and NURBS can provide lighting normals for generated vertices automatically

Specifying Material Properties

```
glMaterialfv( face, property, value );
```

Material properties

- GL_EMISSION
- GL_AMBIENT
- GL_DIFFUSE
- GL_SPECULAR
- GL_SHININESS

Primitive have material properties for front and back sides

61

Material properties describe the color and surface properties of a material (dull, shiny, etc) OpenGL supports material properties for both the front and back of objects, as described by their vertex winding

The OpenGL material properties are

- GL_EMISSION - color emitted from the object (think of a firefly)
- GL_AMBIENT - color of object when not directly illuminated
- GL_DIFFUSE - base color of object
- GL_SPECULAR - color of highlights on object
- GL_SHININESS - concentration of highlights on objects Values range from 0 (very rough surface - no highlight) to 128 (very shiny)

Material properties can be set for each face separately by specifying either GL_FRONT or GL_BACK, or for both faces simultaneously using GL_FRONT_AND_BACK

Material Example

```
GLfloat green[] = { 0 0, 1 0, 0 0, 0 5 },
GLfloat red[]   = { 1 0, 0 0, 0 0, 0 75 },
GLfloat white[] = { 1 0, 1 0, 1 0, 1 0 },

glMaterialfv( GL_FRONT, GL_AMBIENT_AND_DIFFUSE,
   green ),
glMaterialfv( GL_BACK, GL_AMBIENT_AND_DIFFUSE,
   red ),
glMaterialfv( GL_FRONT, GL_SPECULAR, white ),
glMaterialf( GL_FRONT_AND_BACK, GL_SHININESS,
   100 0 ),
```

62

In the above example, we set the following properties for the material

- ambient and diffuse colors for the front side (as determined by the vertex winding) to a 50% opaque green
- ambient and diffuse color for the back side to a 75% opaque red color
- specular color for the front side to be full intensity white
- shininess to 100 (shiny) for both the front and back sides

The transparency of an object is controlled only by the diffuse material's alpha value

Light Sources

`glLightfv( light, property, value ),`

light ***specifies which light***

- multiple lights starting with `GL_LIGHT0`
- `glGetIntegerv( GL_MAX_LIGHTS, &n ),`

Infinite and local lights

- `GL_POSITION`
  - *w coordinate determines type*

Local lights can also be spot lights

63

The `glLight()` call is used to set the parameters for a light OpenGL implementations must support at least eight lights, which are named `GL_LIGHT0` through `GL_LIGHT`*n*, where *n* is one less than the maximum number supported by an implementation

OpenGL supports two types of lights infinite (directional) and local (point) light sources The type of light is determined by the w coordinate of the light's position

- if (w == 0 0) then the light is an infinite light, with (x,y,z) specifying the light's direction
- if (w != 0 0) then the light is a local light, with (x/w,y/w,z/w) specifying the light's position

A local light can also be converted into a spot light By setting the `GL_SPOT_DIRECTION`, `GL_SPOT_CUTOFF`, and `GL_SPOT_EXPONENT`, the local light will shine in a direction and its light will be limited to a cone centered around that direction vector

Light Sources (cont.)

Light color properties

- GL_AMBIENT
- GL_DIFFUSE
- GL_SPECULAR

Light attenuation

- GL_CONSTANT_ATTENUATION
- GL_LINEAR_ATTENUATION
- GL_QUADRATIC_ATTENUATION

64

OpenGL light's can emit different colors for each of a materials properties For example, a light's GL_AMBIENT color is combined with a material's GL_AMBIENT color to produce the ambient contribution to the color - Likewise for the diffuse and specular colors

Each OpenGL light source supports attenuation, which describes how light diminishes with distance The OpenGL model supports quadratic attenuation, and utilized the following attenuation factor, f_i, where d is the distance from the eyepoint to the vertex being lit

$$f_i = \frac{1.0}{k_0 + k_1 * d + k_2 * d^2}$$

where

- k_0 is the GL_CONSTANT_ATTENUATION term
- k_1 is the GL_LINEAR_ATTENUATION term
- k_2 is the GL_QUADRATIC_ATTENUATION term

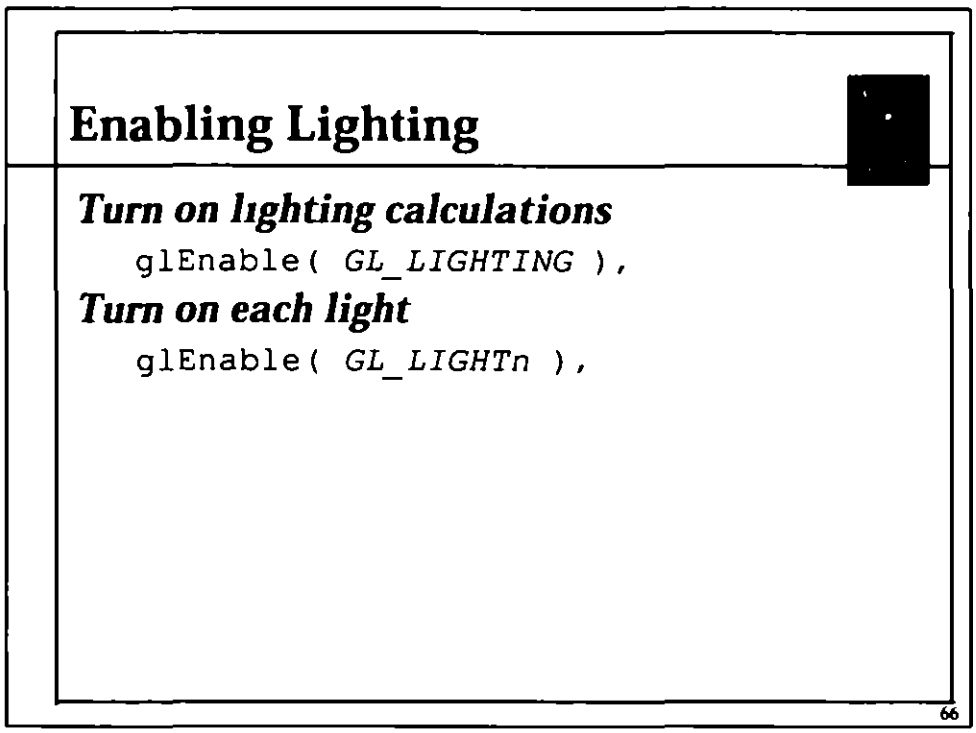
Lighting Example

```
GLfloat white[] = { 1, 1, 1, 0 },
GLfloat magenta[] = { 0 8, 0, 0 8, 0 },
GLfloat pos[] = { 2 5, 6, 3 5, 1 0 },

glLightfv( GL_LIGHT0,
   GL_AMBIENT_AND_DIFFUSE, magenta ),
glLightfv( GL_LIGHT0, GL_SPECULAR, white ),
glLightfv( GL_LIGHT0, GL_POSITION, pos ),

glEnable( GL_LIGHT0 ),
glEnable( GL_LIGHTING ),
```

65

In the above example, we set the following properties for the light

- ambient and diffuse colors to a shade of magenta
- specular color to pure white
- position to (2 5, 6, 3 5) and with w = 1 0, define the light to be a local light
- turn on both GL_LIGHT0 and enable GL_LIGHTING



To enable the lighting computations, you need to enable each light which you would like to use in the scene, using `glEnable( GL_LIGHTn )`, where *n* represents the light

Additionally, you need to globally enable lighting using `glEnable( GL_LIGHTING )`

Controlling a Light's position

Position is transformed by current modelview matrix

Different affects based on when position is specified

- eye coordinates
- world coordinates
- model coordinates

Push and pop matrices to uniquely control light's position

67

By specifying different ModelView transformations, you can achieve different types of lighting effects, since the light's position is transformed when the `glLight()` call is issued

If you specify the light's position before any modeling or viewing transformations (eye coordinate space), you'll achieve a headlamp effect, where the light's emanating from the eyepoint

If you specify after the viewing transformation, but before any modeling, the light's position will remain fixed relative to the world coordinate system of the scene This is like mounting the light on a steel rod at the origin Regardless of how you move the world coordinates, the light's position always remains constant relative to the world origin

Finally, if you issue both modeling and viewing transforms, you can animate the light independent of the eye or anything else in your scene

Specifying Lighting Model Properties

`glLightModelfv( property, value )`,

Control global ambient color

- `GL_LIGHT_MODEL_AMBIENT`

Two sided lighting

- `GL_LIGHT_MODEL_TWO_SIDE`

Local viewer mode

- `GL_LIGHT_MODEL_LOCAL_VIEWER`

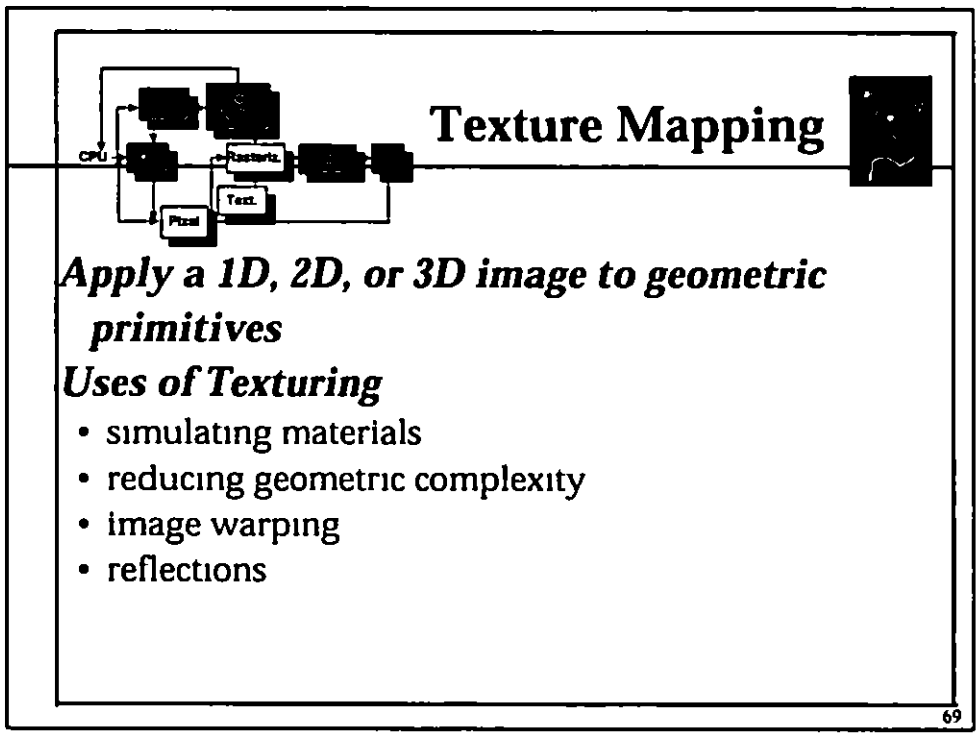
68

`glLightModel()` controls the global parameters of lighting such as the ambient light not contributed by a light and two sided lighting

`GL_LIGHT_MODEL_AMBIENT` sets the global ambient color for the scene This is used in combination with the material's ambient to produce ambient lighting, even if no lights are enabled

`GL_LIGHT_MODEL_TWO_SIDE` is used to enable primitives to have different material properties for each side Based on the winding of the primitive (set with `glCullFace()`), you can set up front and back materials for primitives

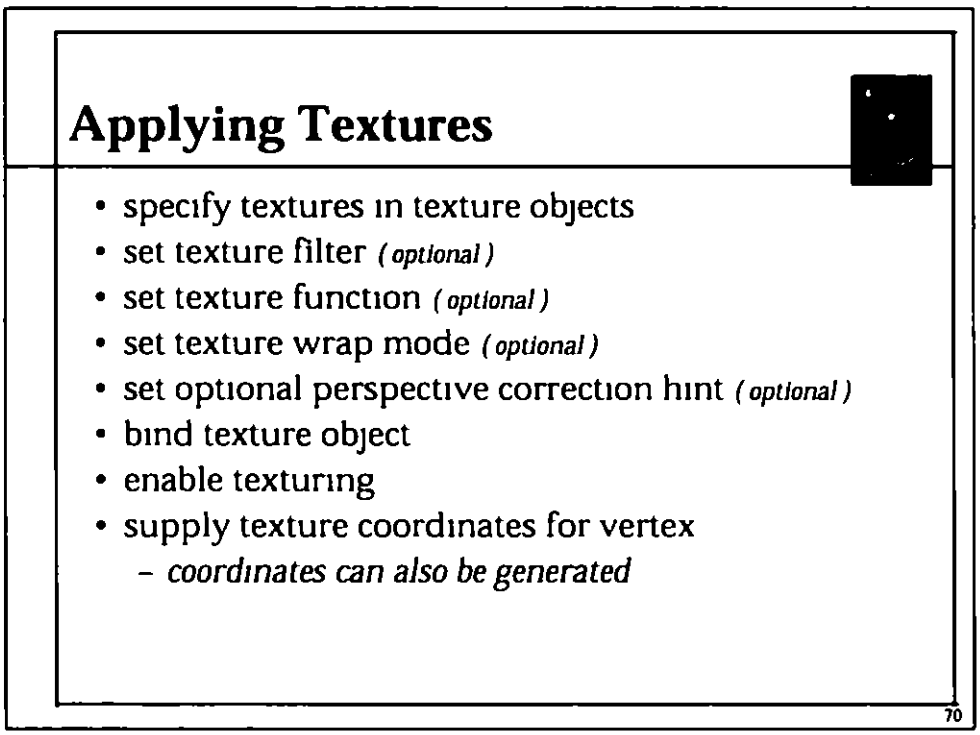
`GL_LIGHT_MODEL_LOCAL_VIEWER` is used to produce better lighting results, by eliminating some approximations made to make OpenGL lighting faster This setting will produce better lighting results, but at a possible performance penalty



In this section, we'll discuss *texture* (sometimes also called *image*) mapping Texture mapping augments the colors specified for a geometric primitive with the colors stored in an image An image can be a 1D, 2D, or 3D set of colors called *texels* 2D textures will be used throughout the section for demonstrations, however, the processes described are identical for 1D and 3D textures

Some of the many uses of texture mapping include

- simulating materials like wood, bricks or granite
- reducing the complexity (number of polygons) of a geometric object
- image processing techniques like image warping and rectification, rotation and scaling
- simulating reflective surfaces like mirrors or polished floors



The general steps to enable texturing are listed above Some steps are optional, and due to the number of combinations, complete coverage of the topic is outside the scope of this course

We will be using the *texture object* approach Using texture objects may enable your OpenGL implementation to make some optimizations behind the scenes

As with any other OpenGL state, texture mapping requires that `glEnable()` be called The tokens for texturing are

`GL_TEXTURE_1D` - one dimensional texturing

`GL_TEXTURE_2D` - two dimensional texturing

`GL_TEXTURE_3D` - three dimensional texturing

2D texturing is the most commonly used 1D texturing is useful for applying contours to objects (like altitude contours to mountains) 3D texturing is useful for volume rendering

Texture Objects

Like display lists for texture images

- one image per texture object
- may be shared by several graphics contexts

Generate texture names

```
glGenTextures( n, *texIds );
```

Create texture objects with texture data and state

```
glBindTexture( target, id );
```

Bind textures before using

```
glBindTexture( target, id );
```

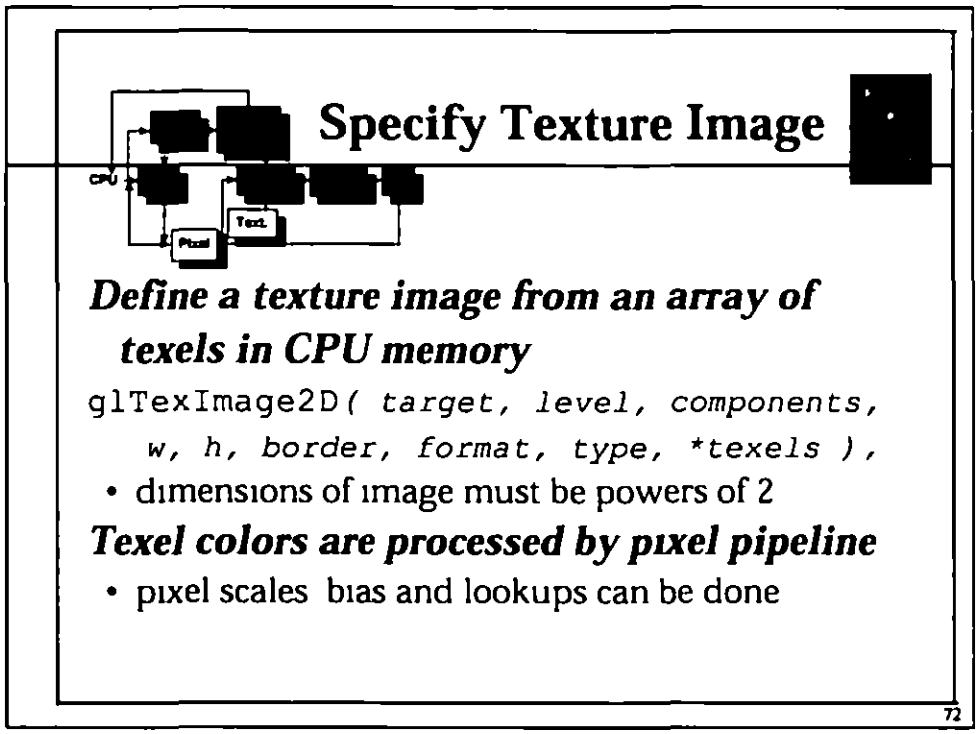
71

The first step in creating texture objects is to have OpenGL reserve some indices for your objects `glGenTextures()` will request *n* texture ids and return those values back to you in *texIds*

To begin defining a texture object, you call `glBindTexture()` with the *id* of the object you want to create The *target* is on of `GL_TEXTURE_{123}D` All texturing calls become part of the object until the next `glBindTexture()` is called

To have OpenGL use a particular texture object, call `glBindTexture()` with the *target* and *id* of the object you want to be active

To delete texture objects, use `glDeleteTextures( n, *texIds );` where *texIds* is an array of texture object identifiers to be deleted



Specifying the texels for a texture is done using the `glTexImage{123}D()` call This will transfer the texels in CPU memory to OpenGL, where they will be processed and converted into an internal format

The array of texels sent to OpenGL with `glTexImage*()` must be a power of two in both directions An optional one texel wide border may be added around the image This is useful for certain wrapping modes

The *level* parameter is used for defining how OpenGL should use this image when mapping texels to pixels Generally, you'll set the *level* to 0, unless you're using a texturing technique called *mipmapping*, which we'll discuss in a few slides

Converting A Texture Image

If dimensions of image are not power of 2

```
gluScaleImage( format, w_in, h_in,
type_in, *data_in, w_out, h_out, type_out,
*data_out ),
```

- *_in ***is for source image***
- *_out ***is for destination image***

Image interpolated and filtered during scaling

If your image does not meet the power of two requirement for a dimension, the `gluScaleImage()` call will resample an image to a particular size It uses a simple box filter to interpolate the new images pixels from the source image

Additionally, `gluScaleImage()` can be used to convert from one data type (i e `GL_FLOAT`) to another type, which may better match the internal format in which OpenGL stores your texture

Specifying a Texture Other Methods

Use frame buffer as source of texture image

uses current buffer as source image

`glCopyTexImage2D( )`

`glCopyTexImage1D( )`

Modify part of a defined texture

`glTexSubImage2D( )`

`glTexSubImage1D( )`

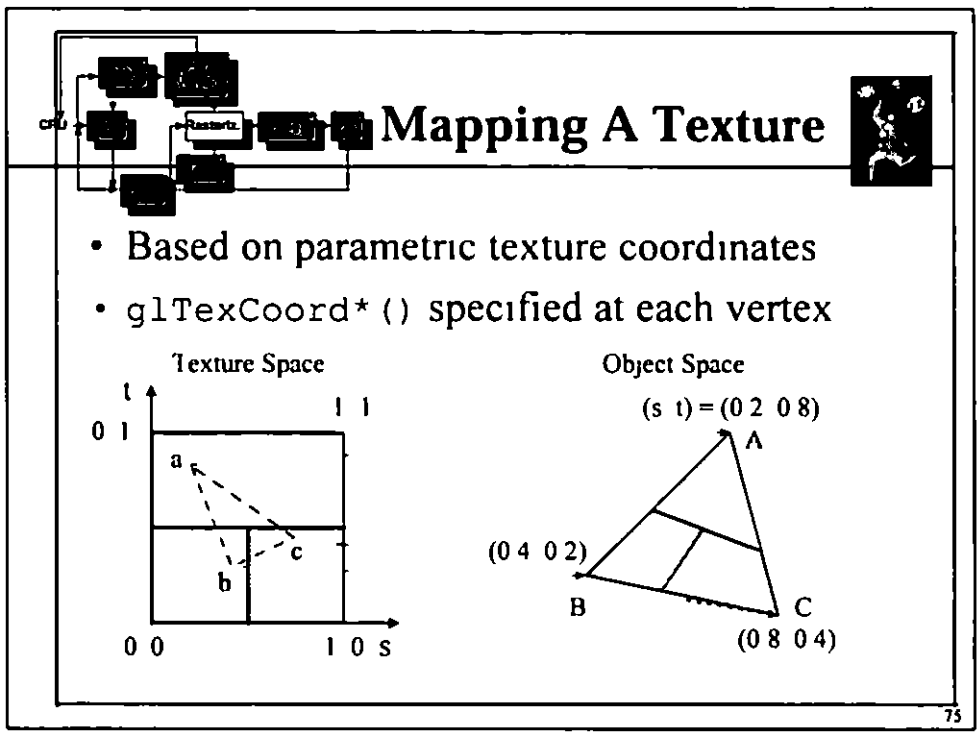
Do both with `glCopyTexSubImage2D( )`, ***etc***

74

`glCopyTexImage*()` allows textures to be defined by in any of OpenGL's buffers The source buffer is selected using the `glReadBuffer()` command

Using `glTexSubImage*()` to replace all or part of an existing texture often outperforms using `glTexImage*()` to allocate and define a new one This can be useful for creating a "texture movie" (sequence of textures which changes appearance on an object's surface)

There are some advanced techniques using `glTexSubImage*()` which include loading an image which doesn't meet the power of two requirement Additionally, several small images can be "packed" into one larger image (which was originally created with `glTexImage*()`), and loaded individually with `glTexSubImage*()` Both of these techniques require the manipulation of the texture transform matrix, which is outside the scope of this course



When you want to map a texture onto a geometric primitive, you need to provide texture coordinates The `glTexCoord*()` call sets the current texture coordinates Valid texture coordinates are between 0 and 1, for each texture dimension, and the default texture coordinate is (0, 0, 0, 1) If you pass fewer texture coordinates than the currently active texture mode (for example, using `glTexCoord1d()` while `GL_TEXTURE_2D` is enabled), the additionally required texture coordinates take on default values

Generating Texture Coordinates

Automatically generate texture coordinates

`glTexGen{ifd}[v]()`

specify a plane Ax+By+Cz+D = 0

- generate texture coordinates based upon distance from plane

generation modes

- `GL_OBJECT_LINEAR`
- `GL_EYE_LINEAR`
- `GL_SPHERE_MAP`

76

You can have OpenGL automatically generate texture coordinates for vertices by using the `glTexGen()` and `glEnable( GL_TEXTURE_GEN_{STRQ} )` The coordinates are computed by determining the vertex's distance from each of the enabled generation planes

As with lighting positions, texture generation planes are transformed by the ModelView matrix, which allows different results based upon when the `glTexGen()` is issued

There are three ways in which texture coordinates are generated

`GL_OBJECT_LINEAR` - textures are fixed to the object (like wall paper)

`GL_EYE_LINEAR` - texture fixed in space, and object move through texture (like underwater light shining on a swimming fish)

`GL_SPHERE_MAP` - object reflects environment, as if it were made of mirrors (like the shiny guy in *Terminator 2*)

Texture Application Methods

Filter Modes

- minification or magnification
- special *mipmap* minification filters

Wrap Modes

- clamping or repeating

Texture Functions

- how to mix primitive s color with texture s color
  - *blend modulate or replace texels*

77

Textures and the objects being textured are rarely the same size (in pixels) *Filter modes* determine the methods used by how texels should be expanded (*magnification*), or shrunk (*minification*) to match a pixel's size An additional technique, called *mipmapping* is a special instance of a minfication filter

Wrap modes determine how to process texture coordinates outside of the [0,1] range The available modes are

`GL_CLAMP` - clamp any values outside the range to closest valid value, causing the edges of the texture to be "smeared" across the primitive

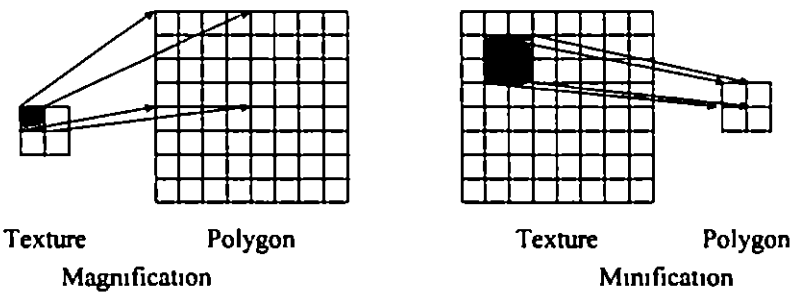
`GL_REPEAT` - use only the fractional part of the texture coordinate, causing the texture to repeat across an object

Finally, the texture environment describes how a primitives fragment colors and texel colors should be combined to produce the final framebuffer color Depending upon the type of texture (i e intensity texture vs RGBA texture) and the mode, pixels and texels may be simply multiplied, linearly combined, or the texel may replace the fragment's color altogether

Filter Modes

Example

```
glTexParameteri( target, type, mode ),
```



78

Filter modes control how pixels are minified or magnified Generally a color is computed using the nearest texel or by a linear average of several texels

The filter *type*, above is one of `GL_TEXTURE_MIN_FILTER` or `GL_TEXTURE_MAG_FILTER`

The *mode* is one of `GL_NEAREST`, `GL_LINEAR`, or special modes for mipmapping Mipmapping modes are used for minification only, and have values of

- `GL_NEAREST_MIPMAP_NEAREST`
- `GL_NEAREST_MIPMAP_LINEAR`
- `GL_LINEAR_MIPMAP_NEAREST`
- `GL_LINEAR_MIPMAP_LINEAR`

Full coverage of mipmap texture filters is outside the scope of this course

Mipmapped Textures

Mipmap allows for prefiltered texture maps of decreasing resolutions

Lessens interpolation errors for smaller textured objects

Declare mipmap level during texture definition

```
glTexImage*D(GL_TEXTURE_*D, level,    )
```

GLU mipmap builder routines

```
gluBuild1DMipmaps(    )
gluBuild2DMipmaps(    )
```

79

As primitives become smaller in screen space, a texture may appear to shimmer as the minification filters creates rougher approximations Mipmapping is an attempt to reduce the shimmer effect by creating several approximations to the original image at lower resolutions

Each mipmap level should have an image which is one-half the height and width of the previous level, to an minimum of one texel in either dimension For example, level 0 could be 32 x 8 texels Then level 1 would be 16 x 4, level 2 would be 8 x 2, level 3, 4 x 1, level 4, 2 x 1, finally, level 5, 1 x 1

The `gluBuild*Dmipmaps()` routines will automatically generate each mipmap image, and call `glTexImage*D()` with the appropriate *level* value

Wrapping Mode

Example

```
glTexParameteri( GL_TEXTURE_2D,
   GL_TEXTURE_WRAP_S, GL_CLAMP )
glTexParameteri( GL_TEXTURE_2D,
   GL_TEXTURE_WRAP_T, GL_REPEAT )
```

80

Wrap mode determines what should happen if a texture coordinate lies outside of the [0,1] range If the GL_REPEAT wrap mode is used, for texture coordinate values less than zero or greater than one, the integer is ignored and only the fractional value is used

If the GL_CLAMP wrap mode is used, the texture value at the extreme (either 0 or 1) is used

Texture Functions

Controls how texture is applied

`glTexEnv{f i}[v]( GL_TEXTURE_ENV, prop, param )`

GL_TEXTURE_ENV_MODE ***modes***

- GL_MODULATE
- GL_BLEND
- GL_REPLACE

Set blend color with GL_TEXTURE_ENV_COLOR

81

The texture mode determines how texels and fragment colors are combined The most common modes are

GL_MODULATE - multiply texel and fragment color

GL_BLEND - linearly blend texel, fragment, env color

GL_REPLACE - replace fragment's color with texel

If *prop* is GL_TEXTURE_ENV_COLOR, *param* is an array of four floating point values representing the color to be used with the GL_BLEND texture function

Perspective Correction Hint

Texture coordinate and color interpolation

either linearly in screen space

or using depth/perspective values (slower)

Noticeable for polygons "on edge"

`glHint( GL_PERSPECTIVE_CORRECTION_HINT, hint )`

where `hint` is one of

- `GL_DONT_CARE`
- `GL_NICEST`
- `GL_FASTEST`

82

An OpenGL implementation may chose to ignore hints

Is There Room for a Texture?

Query largest dimension of texture image

typical largest square texture

doesn t consider internal format size

```
glGetIntegerv( GL_MAX_TEXTURE_SIZE, &size )
```

Texture proxy

- will memory accommodate requested texture size?
- no image specified placeholder
- if it won t fit texture state variables set to 0
  - *doesn t know about other textures*
  - *only considers whether this one texture will fit all of memory*


```
GLint proxyComponents
glTexImage2D(GL_PROXY_TEXTURE_2D, 0, GL_RGBA8, 64, 64, 0,
GL_RGBA, GL_UNSIGNED_BYTE  NULL)
glGetTexLevelParameteriv(GL_PROXY_TEXTURE_2D, 0,
GL_TEXTURE_COMPONENTS, &proxyComponents)
```

Texture Residency

Working set of textures

- high performance usually hardware accelerated
- textures must be in texture objects
- a texture in the *working set* is *resident*
- for residency of current texture check GL_TEXTURE_RESIDENT state

If too many textures, not all are resident

- can set priority to have some kicked out first
- establish 0.0 to 1.0 priorities for texture objects

84

Query for residency of an array of texture objects

```
GLboolean glAreTexturesResident(GLsizei n, GLuint *texNums,
GLboolean *residences)
```

Set priority numbers for an array of texture objects

```
glPrioritizeTextures(GLsizei n, GLuint *texNums, GLclampf
*priorities)
```

Lower priority numbers mean that, in a crunch, these texture objects will be more likely to be moved out of the working set

One common strategy is avoid prioritization, because many implementations will automatically implement an LRU (least recently used) scheme, when removing textures from the working set

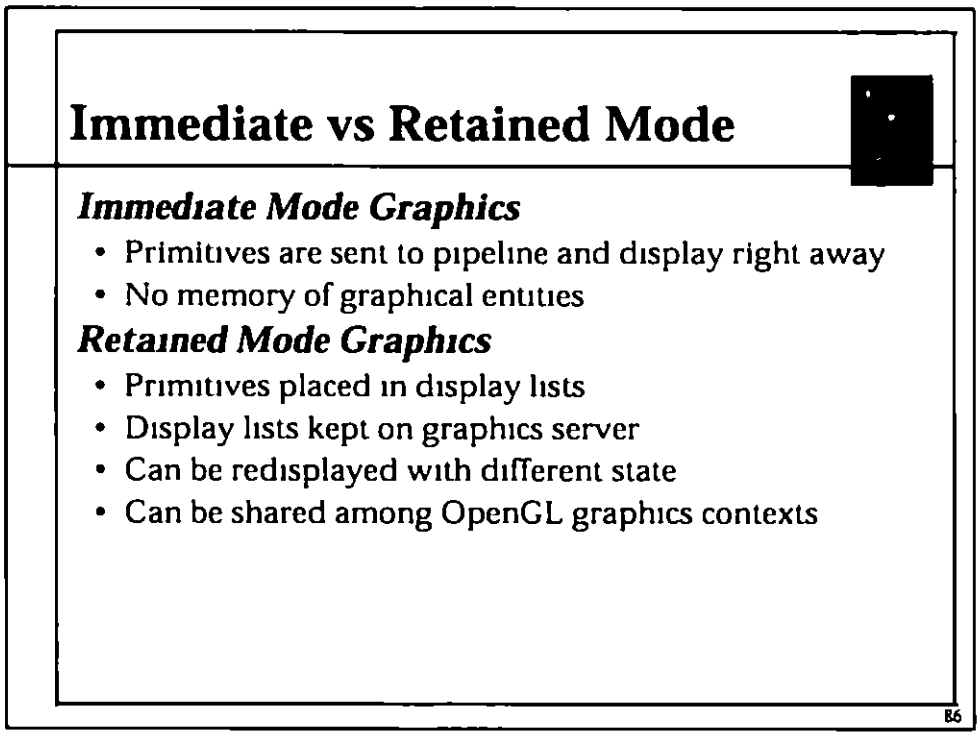
If there is no high-performance working set, then all texture objects are considered to be resident

Overview of Other Topics

Display Lists
Feedback
- Picking/Selection

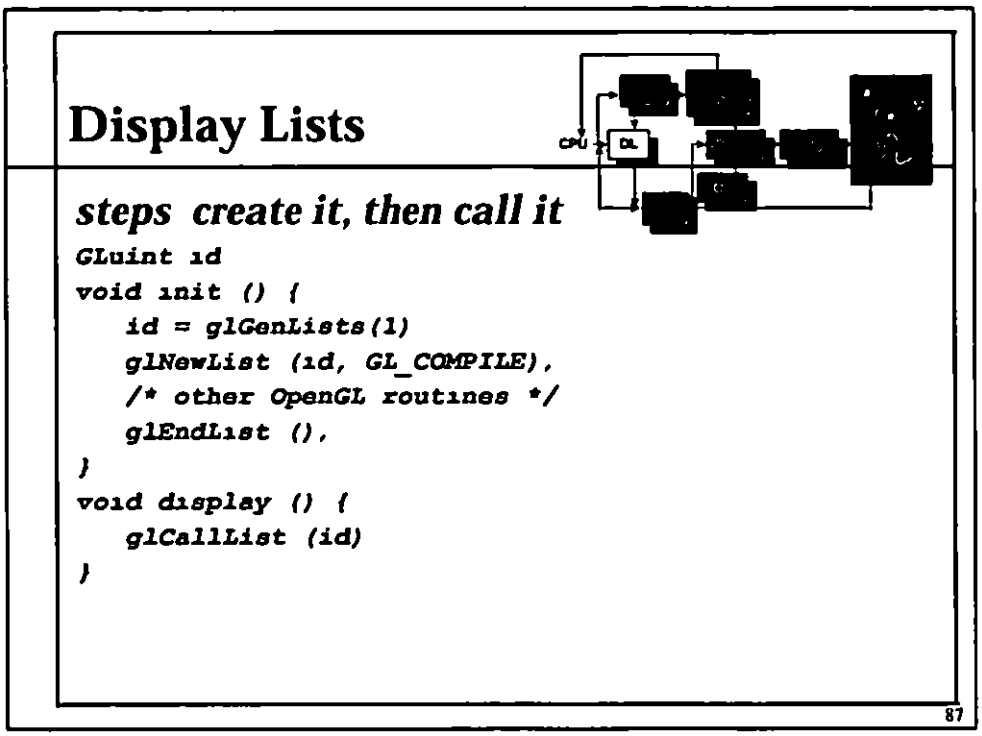
Image Primitives
Fog
Per Fragment Operations
Blending
Antialiasing

85



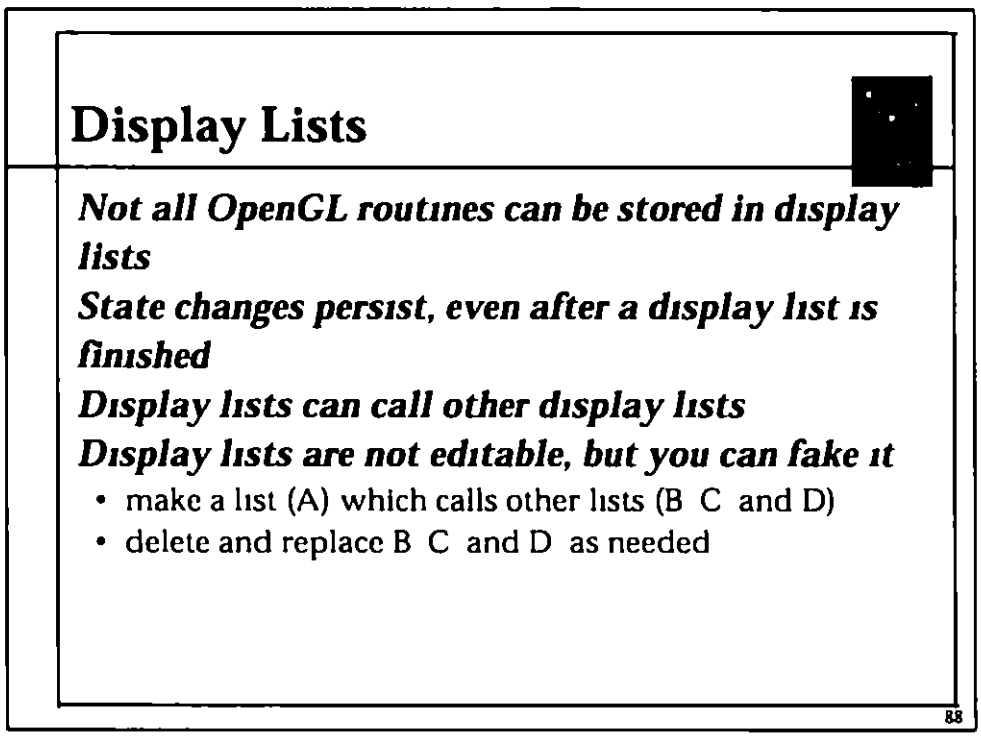
If display lists are shared, texture objects are also shared

To share display lists among graphics contexts in the X Window System, use the `glXCreateContext` routine



Instead of GL_COMPILE, `glNewList` also accepts the constant GL_COMPILE_AND_EXECUTE, which both creates and executes a display list

If a new list is created with the same identifying number as an existing display list, the old list is deleted No error occurs



Some routines cannot be stored in a display list Here is a list of them

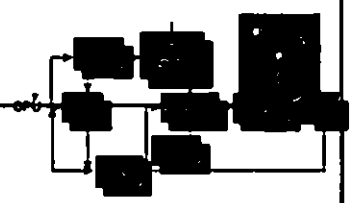
all `glGet*` routines

`glIs*` routines (e g , `glIsEnabled, glIsList, glIsTexture`)

`glGenLists`	`glDeleteLists`	`glFeedbackBuffer`
`glSelectBuffer`	`glRenderMode`	`glVertexPointer`
`glNormalPointer`	`glColorPointer`	`glIndexPointer`
`glTexCoordPointer`		`glEdgeFlagPointer`
`glEnableClientState`		`glDisableClientState`
`glReadPixels`	`glPixelStore`	`glGenTextures`
`glDeleteTextures`		`glAreTexturesResident`
`glFlush`	`glFinish`	

If there is an attempt to store any of these routines in a display list, the routine is executed in immediate mode No error occurs

Feedback & Selection



Usually, transformed vertices and colors generate an image in the frame buffer

Feedback mode transformed values are returned to the application

Selection mode if primitives are drawn within viewing volume, names are returned to the application

- In both cases drawing stopped no pixels are produced

89

Picking

Picking is a special case of selection

Programming steps

- restrict drawing to small region near cursor
  use `gluPickMatrix()` on projection matrix
- enter selection mode rerender scene
- primitives drawn near cursor cause hits
- exit selection, analyze hit records

90

The picking region is usually specified in a piece of code like this

```
glMatrixMode (GL_PROJECTION)
glLoadIdentity()
gluPickMatrix(x, y, width, height, viewport)
gluPerspective(   ) or glOrtho(   )
```

The picking matrix is the rare situation where the standard projection matrix (perspective or ortho) is multiplied onto a non-identity matrix

Each *hit record* contains

number of names per hit

smallest and largest depth values

all the names

Picking Pseudocode

```
glutMouseFunc (pickMe)

void pickMe (int button, int state  int x  int y) {
   GLuint nameBuffer[256]
   GLint hits
   GLint myViewport[4]
   if (button != GLUT_LEFT_BUTTON ||
       state != GLUT_DOWN) return
   glGetIntegerv (GL_VIEWPORT, myViewport)
   glSelectBuffer (256, nameBuffer)
   (void) glRenderMode (GL_SELECT)
   glInitNames()
```

91

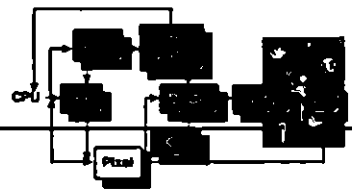
Picking Pseudocode (continued)

```
   glMatrixMode (GL_PROJECTION)
   glPushMatrix ()
   glLoadIdentity ()
   gluPickMatrix ((GLdouble) x, (GLdouble)
      (myViewport[3]-y)  5 0, 5 0, myViewport)
 gluPerspective or glOrtho or other projection
   glPushName (1)
 draw something
   glLoadName (2)
 draw something else  continue
   glMatrixMode (GL_PROJECTION)
   glPopMatrix ()
   hits = glRenderMode (GL_RENDER)
 process nameBuffer
}
```

92

Be sure and push the first name you use onto the name stack After the first name is pushed, it can be replaced by `glLoadName`

Bitmaps and Images



Pixel-based primitives

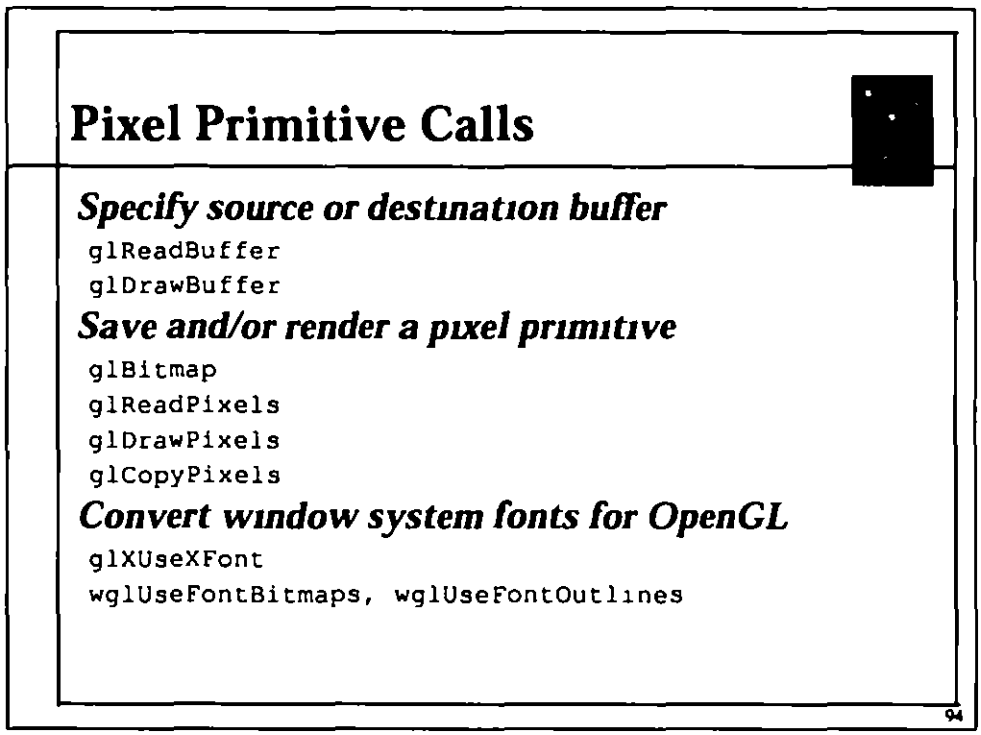
- bitmaps (one bit per pixel)
- pixmaps or pixel rectangles (many bits per pixel)
- texture images treated similarly

Use `glRasterPos*()` to position pixel primitive

- raster position transformed like geometry

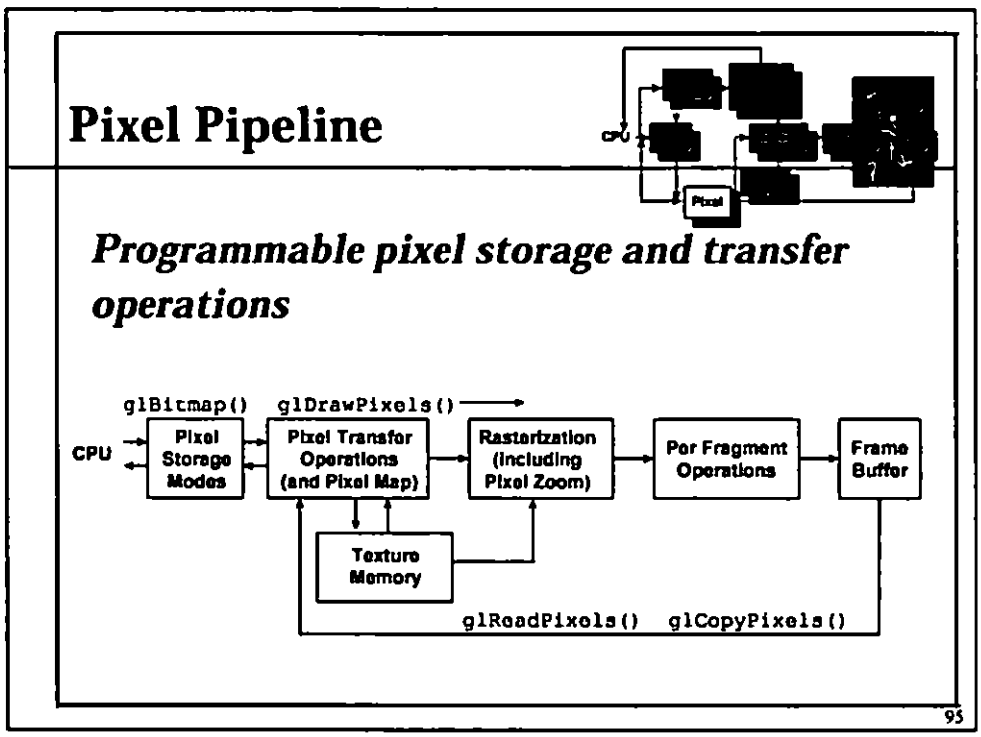
OpenGL does <u>not</u> encode/decode file formats (such as, GIF, JPEG, TIFF)

93



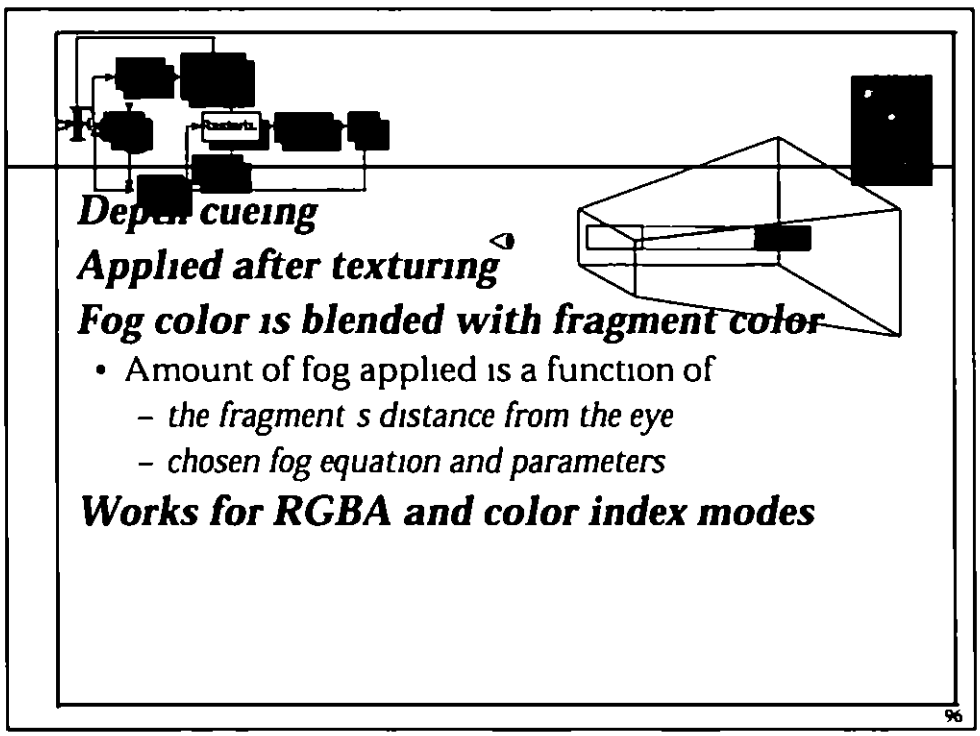
Source = incoming fragments

Destination = values (color, depth, etc) which are already in the bitplanes



Warning non-default values for pixel storage and transfer can be very slow

For best performance, the internal representation of a pixel array should match the hardware For example, for a 24 bit frame buffer, 8-8-8 RGB would probably be a good match, but 10-10-10 RGB could be bad



Programming steps to perform fog

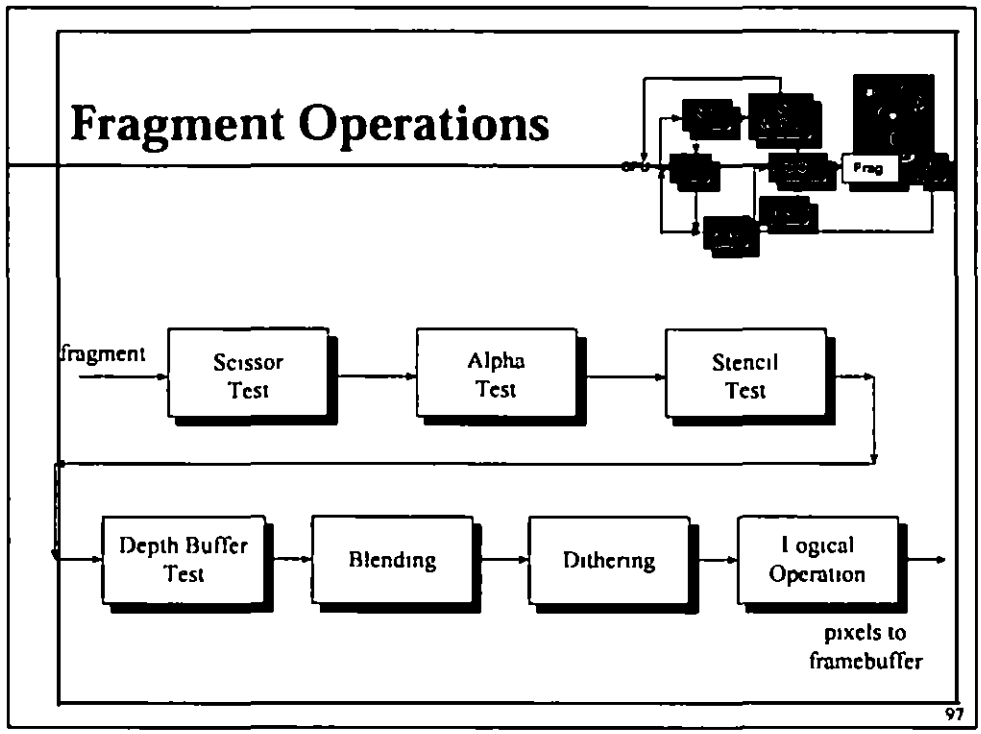
1) clear screen to fog color

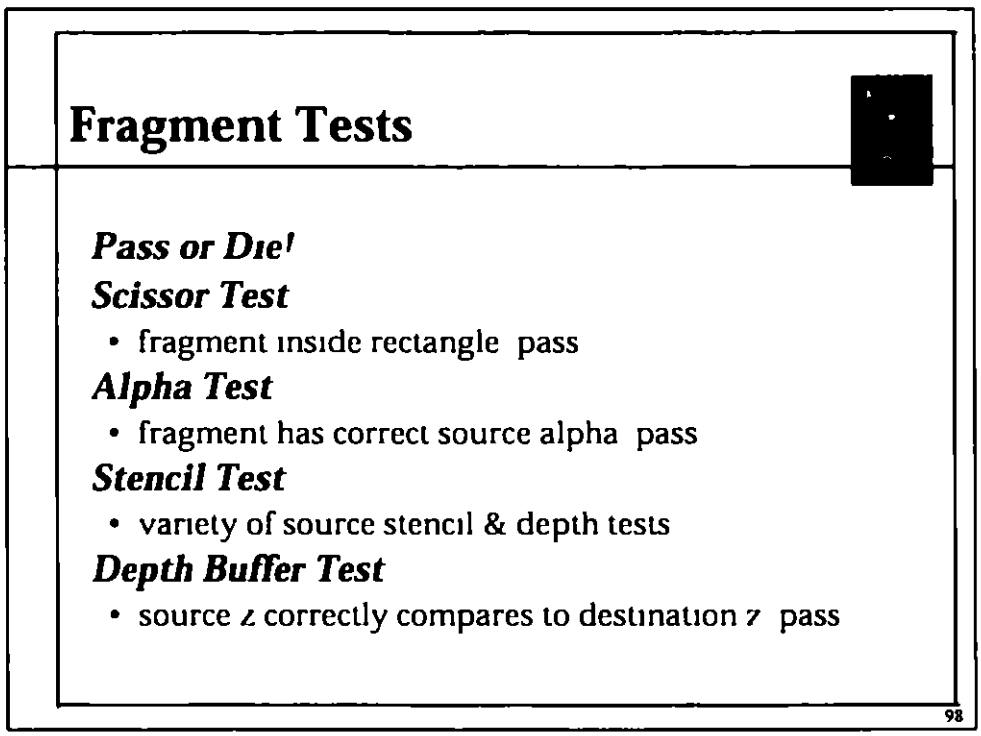
2) turn on fog with `glEnable(GL_FOG)`

3) use `glFog` to specify the fog equation to use (linear, exponential or exponential squared)

4) use `glFog` to specify parameters which affect fog density and color

5) use `glHint` to specify quality/performance tradeoffs


Fragment Operations
Frag
fragment
Scissor Test
Alpha Test
Stencil Test
Depth Buffer Test
Blending
Dithering
Logical Operation
pixels to framebuffer
97



OpenGL routines which control these fragment operations

```
glEnable (GL_SCISSOR_TEST)
glScissor (GLint x, GLint y, GLsizei width, GLsizei height)

glEnable (GL_ALPHA_TEST)
glAlphaFunc (GLenum func, GLclampf ref)

glEnable (GL_STENCIL_TEST)
glStencilFunc (GLenum func, GLint ref, GLuint mask)
glStencilOp (GLenum fail, GLenum zfail, GLenum zpass)
glStencilMask (GLuint mask)

glEnable (GL_DEPTH_TEST)
glDepthFunc (GLenum func)
glDepthRange (GLclampd near, GLclampd far)
glDepthMask (GLboolean flag)
```

Blending

RGBA mode only, not color index

- alpha represents 0% to 100% opacity

Translucency effects

- combination of source and destination colors
- $C_d = C_sS + C_dD$

  *dest color = src color * src factor + dest color * dest factor*

Rendering order matters, need to sort polygons

- depth buffer doesn t work well with alpha blending

Alpha buffer (bitplanes) rarely needed

- source alpha usually enough

99

Blending routines

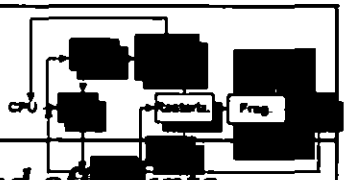
```
glEnable(GL_BLEND)
glBlendFunc(sfactor, dfactor)
```

Some typical choices for sfactor and dfactor are

```
glBlendFunc(GL_ONE, GL_ZERO)  /* default--no blending */
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)
glBlendFunc(GL_SRC_ALPHA, GL_ONE)
```

There are very few uses for having actual bitplanes (destination alpha) to save alpha values One use for destination alpha is to save a chromakey value for manipulating video images

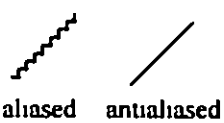
Antialiasing



Smooth jagged lines and round off points

2 steps

- during rasterization
  - *subpixel coverage values calculated*
  - *in RGBA coverage = alpha value*
  - *in color index coverage = last 4 bits of color index*
- during fragment operations
  - *in RGBA blend with source alpha*



100

Routines for antialiasing lines

```
glEnable (GL_LINE_SMOOTH)
glHint (GL_LINE_SMOOTH_HINT, GL_DONT_CARE)
```

Routines for antialiasing points

```
glEnable (GL_POINT_SMOOTH)
glHint (GL_POINT_SMOOTH_HINT, GL_DONT_CARE)
```

Routines for RGBA mode, only

```
glEnable (GL_BLEND)
glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)
```

There is also `glEnable(GL_POLYGON_SMOOTH)` for antialiasing the edges of filled polygons, but it only works in RGBA mode Also hidden surface removal with alpha blending raises additional issues You might use the accumulation buffer as an alternate way to achieve full scene antialiasing

Last Fragment Operations

Dithering

- to compensate for low color resolution
- combine *close* colors to approximate the color you really want
- on high-resolution systems, dithering is a no-op

Logical Operations

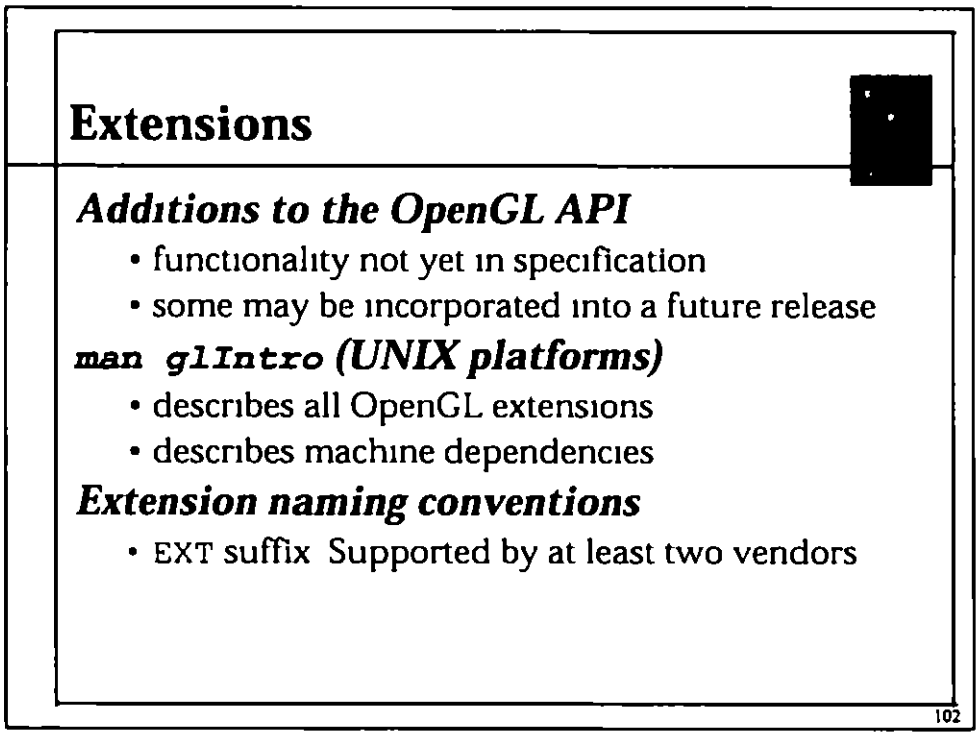
- logical operations on incoming fragment (source) and current color buffer (destination)

101

OpenGL routines

```
glEnable (GL_DITHER)
```

```
glEnable (GL_INDEX_LOGIC_OP) or glEnable (GL_COLOR_LOGIC_OP)
glLogicOp (GLenum opcode)
```



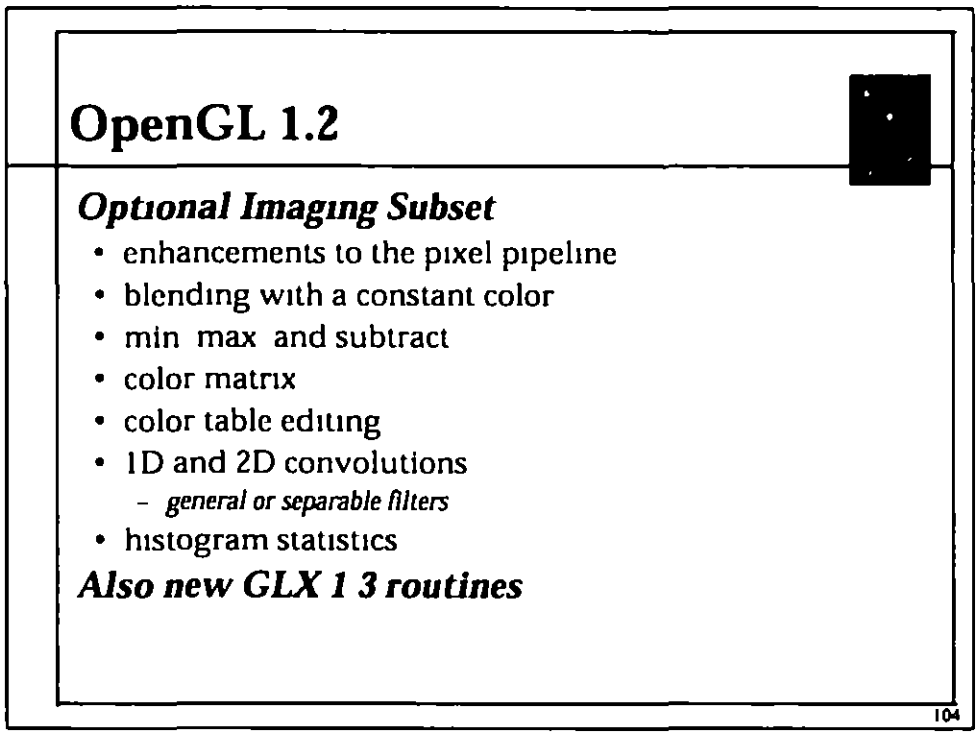
The availability of an extension can be queried by using `glGetString(GL_EXTENSIONS)` and looking for a string which is specified by the vendor An alternate way of identifying the availability of an extension is to look for a constant at compile time (`#ifdef XXX`)

OpenGL 1.2

Mandatory New Core Capabilities

- vertex normals rescaled
- 3D textures
- texture coordinate edge clamping
- level of detail for mipmap textures
- specular highlights after texturing
- BGRA and packed pixel formats
- vertex array subrange operations

103



The OpenGL 1 2 imaging subset initiates a new concept for OpenGL functionality The imaging subset is not mandatory for all OpenGL 1 2 implementations However, if a vendor chooses to support it, they must support all functionality in the imaging subset

Support for the imaging subset can be detected by using `glGetString(GL_EXTENSIONS)` and looking for the string `ARB_imaging` However, it isn't an extension, and the functions and enumerants do not contain EXT suffixes

The GLX 1 3 features include pixel buffers, more flexible frame buffer configuration, and support for read-only drawables (preparing for support for video)

Final Review: Typical Steps

open a window with specific visual/pixel format

- establish depth buffer and double buffer

to initialize

- read images from disk load color maps
- tessellate polygons load vertex arrays
- create display lists texture objects light sources

in resize()

- re-establish viewport projection, and viewing transformations
- careful with the aspect ratio

105

Final Review (2)

in display()

- clear screen to background color
- initially push modelview matrix (usually)
- change states and render geometry & images
- finally restore modelview matrix (usually)
- swap buffers
- check for errors

optional routines for input devices or idle function

- input device may initiate picking

enter event processing infinite loop

On-Line Resources

http //www opengl org

- start here up to date specification

news comp graphics api opengl

http //reality sgi com/opengl/opengl-links html

http //www microsoft com/hwdev/devdes/opengl htm

ftp //sgigate sgi com/pub/opengl

http //www ssec wisc edu/~brianp/Mesa html

- Brian Paul s Mesa 3D

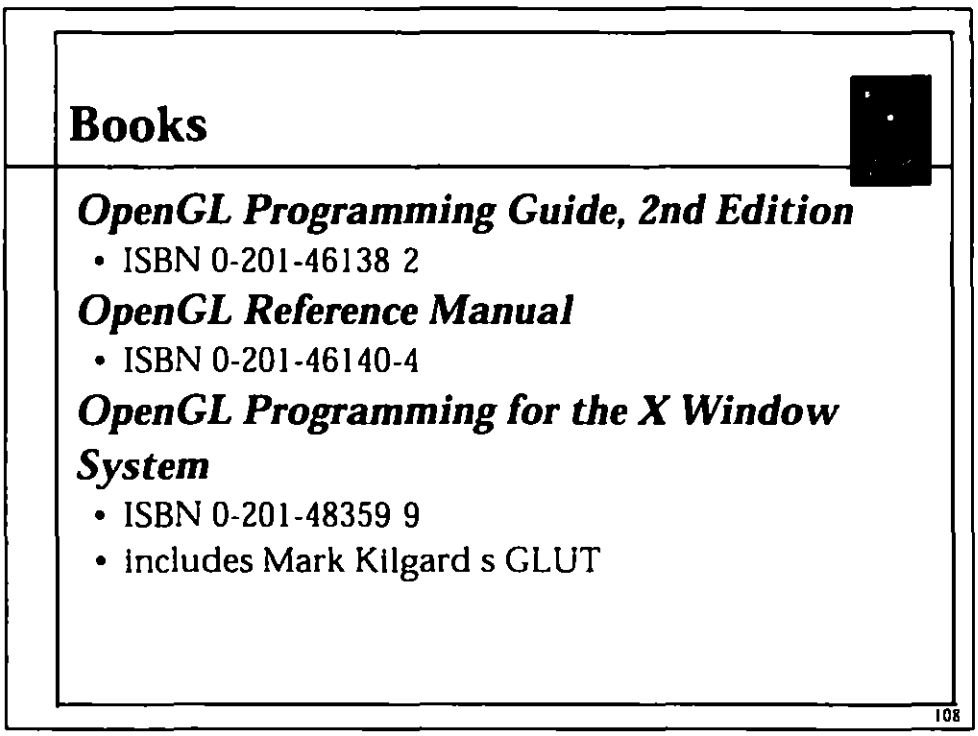
http //www cs utah edu/~narobins/opengl html

- very special thanks to Nate Robins for the OpenGL Tutors
- source code for tutors available here!

http //www specbench org/gpc/opc static

- benchmarks

107



Other Books

OpenGL Programming for Windows 95 and Windows NT
by Ron Fosner, ISBN 0-201-40709-4

OpenGL SuperBible
by Richard S Wright, Jr and Michael Sweet, ISBN 1-57169-073-5

Interactive Computer Graphics A Top-Down Approach with OpenGL
by Edward Angel, ISBN 0-201-85571-2

Thanks for Coming

Questions and Answers

mason@woo com
shreiner@sgi com

The Design of the OpenGL Graphics Interface

Mark Segal
Kurt Akeley
Silicon Graphics Computer Systems
2011 N Shoreline Blvd Mountain View CA 94039

Abstract

OpenGL is an emerging graphics standard that provides advanced rendering features while maintaining a simple programming model Because OpenGL is rendering only it can be incorporated into any window system (and has been into the X Window System and a soon to be released version of Windows) or can be used without a window system An OpenGL implementation can efficiently accommodate almost any level of graphics hardware from a basic framebuffer to the most sophisticated graphics subsystems It is therefore a good choice for use in interactive 3D and 2D graphics applications

We describe how these and other considerations have governed the selection and presentation of graphical operators in OpenGL Complex operations have been eschewed in favor of simple direct control over the fundamental operations of 3D and 2D graphics Higher level graphical functions may however be built from OpenGL s low level operators as the operators have been designed with such layering in mind

CR Categories and Subject Descriptors I 3 3 [Computer Graphics] Picture/Image Generation I 3 7 [Computer Graphics] Three Dimensional Graphics and Realism

1 Introduction

Computer graphics (especially 3D graphics and interactive 3D graphics in particular) is finding its way into an increasing number of applications from simple graphing programs for personal computers to sophisticated modeling and visualization software on workstations and supercomputers As the interest in computer graphics has grown so has the desire to be able to write applications that run on a variety of platforms with a range of graphical capabilities A graphics standard eases this task by eliminating the need to write a distinct graphics driver for each platform on which the application is to run

To be viable a graphics standard intended for interactive 3D applications must satisfy several criteria It must be implementable on platforms with varying graphics capabilities without compromising the graphics performance of the underlying hardware and without sacrificing control over the hardware s operation It must provide a natural interface that allows a programmer to describe rendering operations tersely Finally the interface must be flexible enough to accommodate extensions so that as new graphics operations become significant or available in new graphics subsystems these operations can be provided without disrupting the original interface

OpenGL meets these criteria by providing a simple direct interface to the fundamental operations of 3D graphics rendering It supports basic graphics primitives such as points line segments polygons and images as well as basic rendering operations such as affine and projective transformations and lighting calculations It also supports advanced rendering features such as texture mapping and antialiasing

There are several other systems that provide an API (Application Programmer s Interface) for effecting graphical rendering In the case of 2D graphics the PostScript page description language[5] has become widely accepted making it relatively easy to electronically exchange and to a limited degree manipulate static documents containing both text and 2D graphics Besides providing graphical rendering operators PostScript is also a stack based programming language

The X window system[9] has become standard for UNIX workstations A programmer uses X to obtain a window on a graphics display into which either text or 2D graphics may be drawn X also provides a means for obtaining user input from such devices as keyboards and mice The adoption of X by most workstation manufacturers means that a single program can produce 2D graphics or obtain user input on a variety of workstations by simply recompiling the program This integration even works across a network the program may run on one workstation but display on and obtain user input from another even if the workstations on either end of the network are made by different companies

For 3D graphics several systems are in use One relatively well known system is PHIGS (Programmer s Hierarchical Interactive Graphics System) Based on GKS[6] (Graphics Kernel System) PHIGS is an ANSI (American National Standards Institute) standard PHIGS (and its descendant PHIGS+[11]) provides a means to manipulate and draw 3D objects by encapsulating object descriptions and attributes into a *display list* that is then referenced when the object is displayed or manipulated One advantage of the display list is that a complex object need be described only once even if it is to be displayed many times This is especially important if the object to be displayed must be transmitted across a low bandwidth channel (such as a network) One disadvantage of a display list is that it can require considerable effort to re specify the object if it is being continually modified as a result of user interaction Another difficulty with PHIGS and PHIGS+ (and with GKS) is lack of support for advanced rendering features such as texture mapping

PEX[10] extends X to include the ability to manipulate and draw

3D objects (PEXlib[7] is an API employing the PEX protocol) Originally based on PHIGS PEX allows *immediate mode* rendering meaning that objects can be displayed as they are described rather than having to first complete a display list PEX currently lacks advanced rendering features (although a compatible version that provides such features is under design) and is available only to users of X Broadly speaking however the methods by which graphical objects are described for rendering using PEX (or rather PEXlib) are similar to those provided by OpenGL

Like both OpenGL and PEXlib Renderman[16] is an API that provides a means to render geometric objects Unlike these interfaces however Renderman provides a programming language (called a shading language) for describing how these objects are to appear when drawn This programmability allows for generating very realistic looking images but it is impractical to implement on most graphics accelerators making Renderman a poor choice for interactive 3D graphics

Finally there are APIs that provide access to 3D rendering as a result of methods for describing higher level graphical objects Chief among these are HOOPS[17] and IRIS Inventor[15] The objects provided by these interfaces are typically more complex than the simple geometry describable with APIs like PEXlib or OpenGL they may comprise not only geometry but also information about how they are drawn and how they react to user input HOOPS and Inventor free the programmer from tedious descriptions of individual drawing operations but simple access to complex objects generally means losing fine control over rendering (or at least making such control difficult) In any case OpenGL can provide a good base on which to build such higher level APIs

2 OpenGL

In this section we present a brief overview of OpenGL For a more comprehensive description the reader is referred to [8] or [13]

OpenGL draws *primitives* into a framebuffer subject to a number of selectable modes Each primitive is a point, line segment polygon pixel rectangle or bitmap Each mode may be changed independently the setting of one does not affect the settings of others (although many modes may interact to determine what eventually ends up in the framebuffer) Modes are set, primitives specified and other OpenGL operations described by issuing *commands* in the form of function or procedure calls

Figure 1 shows a schematic diagram of OpenGL Commands enter OpenGL on the left. Most commands may be accumulated in a *display list* for processing at a later time Otherwise commands are effectively sent through a processing pipeline

The first stage provides an efficient means for approximating curve and surface geometry by evaluating polynomial functions of input values The next stage operates on geometric primitives described by vertices points line segments and polygons In this stage vertices are transformed and lit, and primitives are clipped to a viewing volume in preparation for the next stage rasterization The rasterizer produces a series of framebuffer addresses and values using a two-dimensional description of a point, line segment, or polygon Each *fragment* so produced is fed to the next stage that performs operations on individual fragments before they finally alter the framebuffer These operations include conditional updates into the framebuffer based on incoming and previously stored depth values (to effect depth buffering) blending of incoming fragment colors with stored colors as well as masking and other logical operations on fragment values

Finally pixel rectangles and bitmaps bypass the vertex processing portion of the pipeline to send a block of fragments directly through rasterization to the individual fragment operations eventually causing a block of pixels to be written to the framebuffer Values may also be read back from the framebuffer or copied from one portion of the framebuffer to another These transfers may include some type of decoding or encoding

3 Design Considerations

Designing any API requires tradeoffs between a number of general factors like simplicity in accomplishing common operations vs generality or many commands with few arguments vs few commands with many arguments In this section we describe considerations peculiar to 3D API design that have influenced the development of OpenGL

3.1 Performance

A fundamental consideration in interactive 3D graphics is performance Numerous calculations are required to render a 3D scene of even modest complexity and in an interactive application a scene must generally be redrawn several times per second An API for use in interactive 3D applications must therefore provide efficient access to the capabilities of the graphics hardware of which it makes use But different graphics subsystems provide different capabilities so a common interface must be found

The interface must also provide a means to switch on and off various rendering features This is required both because some hardware may not provide support for some features and so cannot provide those features with acceptable performance and also because even with hardware support, enabling certain features or combinations of features may decrease performance significantly Slow rendering may be acceptable for instance when producing a final image of a scene but interactive rates are normally required when manipulating objects within the scene or adjusting the viewpoint In such cases the performance-degrading features may be desirable for the final image but undesirable during scene manipulation

3.2 Orthogonality

Since it is desirable to be able to turn features on and off it should be the case that doing so has few or no side effects on other features If for instance it is desired that each polygon be drawn with a single color rather than interpolating colors across its face doing so should not affect how lighting or texturing is applied Similarly enabling or disabling any single feature should not engender an inconsistent state in which rendering results would be undefined These kinds of feature independence are necessary to allow a programmer to easily manipulate features without having to generate tests for particular illegal or undesirable feature combinations that may require changing the state of apparently unrelated features Another benefit of feature independence is that features may be combined in useful ways that may have been unforseen when the interface was designed

3.3 Completeness

A 3D graphics API running on a system with a graphics subsystem should provide some means to access all the significant functionality of the subsystem If some functionality is available but not provided then the programmer is forced to use a different API to get at the

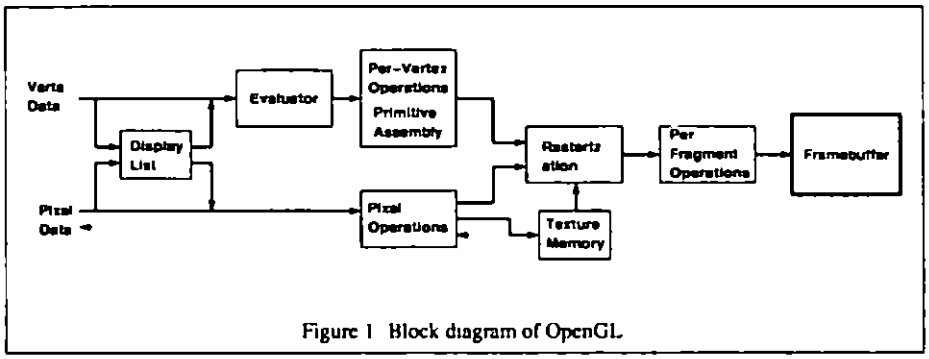


Figure 1 Block diagram of OpenGL

missing features This may complicate the application because of interaction between the two APIs

On the other hand if an implementation of the API provides certain features on one hardware platform then generally speaking those features should be present on any platform on which the API is provided If this rule is broken it is difficult to use the API in a program that is certain to run on diverse hardware platforms without remembering exactly which features are supported on which machines In platforms without appropriate acceleration some features may be poor performers (because they may have to be implemented in software) but at least the intended image will eventually appear

3 4 Interoperability

Many computing environments consist of a number of computers (often made by different companies) connected together by a network In such an environment it is useful to be able to issue graphics commands on one machine and have them execute on another (this ability is one of the factors responsible for the success of X) Such an ability (called *interoperability*) requires that the model of execution of API commands be *client server* the client issues commands and the server executes them (Interoperability also requires that the client and the server share the same notion of how API commands are encoded for transmission across the network the client server model is just a prerequisite) Of course the client and the server may be the same machine

Since API commands may be issued across a network it is impractical to require a tight coupling between client and server A client may have to wait for some time for an answer to a request presented to the server (a *roundtrip*) because of network delays whereas simple server requests not requiring acknowledgement can be buffered up into a large group for efficient transmission to and execution by the server

3.5 Extensibility

As was discussed in the introduction a 3D graphics API should at least in principle be extendable to incorporate new graphics hardware features or algorithms that may become popular in the future Although attainment of this goal may be difficult to gauge until long after the API is first in use steps can be taken to help to achieve it Orthogonality of the API is one element that helps achieve this goal Another is to consider how the API would have been affected if features that were consciously omitted were added to the API

3 6 Acceptance

It might seem that design of a clean consistent 3D graphics API would be a sufficient goal in itself But unless programmers decide to use the API in a variety of applications designing the API will have served no purpose It is therefore worthwhile to consider the effect of design decisions on programmer acceptance of the API

4 Design Features

In this section we highlight the general features of OpenGL s design and provide illustrations and justifications of each using specific examples

4 1 Based on IRIS GL

OpenGL is based on Silicon Graphics IRIS GL While it would have been possible to have designed a completely new API experience with IRIS GL provided insight into what programmers want and don t want in a 3D graphics API Further making OpenGL similar to IRIS GL where possible makes OpenGL much more likely to be accepted there are many successful IRIS GL applications and programmers of IRIS GL will have an easy time switching to OpenGL

4.2 Low Level

An essential goal of OpenGL is to provide device independence while still allowing complete access to hardware functionality The API therefore provides access to graphics operations at the lowest possible level that still provides device independence As a result, OpenGL does not provide a means for describing or modeling complex geometric objects Another way to describe this situation is to say that OpenGL provides mechanisms to describe how complex geometric objects are to be rendered rather than mechanisms to describe the complex object themselves

The OpenGl Utility Library

One benefit of a low level API is that there are no requirements on how an application must represent or describe higher level objects (since there is no notion of such objects in the API) Adherence to this principle means that the basic OpenGL API does not support some geometric objects that are traditionally associated with graphics APIs For instance an OpenGL implementation need not render concave polygons One reason for this omission is that concave

polygon rendering algorithms are of necessity more complex than those for rendering convex polygons and different concave polygon algorithms may be appropriate in different domains In particular if a concave polygon is to be drawn more than once it is more efficient to first decompose it into convex polygons (or triangles) once and then draw the convex polygons Another reason for the omission is that to render a general concave polygon all of its vertices must first be known Graphics subsystems do not generally provide the storage necessary for a concave polygon with a (nearly) arbitrary number of vertices Convex polygons on the other hand can be reduced to triangles as they are specified so no more than three vertices need be stored

Another example of the distinction between low level and high level in OpenGL is the difference between OpenGL evaluators and NURBS The evaluator interface provides a basis for building a general polynomial curve and surface package on top of OpenGL One advantage of providing the evaluators in OpenGL instead of a more complex NURBS interface is that applications that represent curves and surfaces as other than NURBS or that make use of special surface properties still have access to efficient polynomial evaluators (that may be implemented in graphics hardware) without incurring the costs of converting to a NURBS representation

Concave polygons and NURBS are however common and useful operators and they were familiar (at least in some form) to users of IRIS GL Therefore a general concave polygon decomposer is provided as part of the OpenGL Utility Library which is provided with every OpenGL implementation The Utility Library also provides an interface built on OpenGL s polynomial evaluators to describe and display NURBS curves and surfaces (with domain space trimming) as well as a means of rendering spheres cones and cylinders The Utility Library serves both as a means to render useful geometric objects and as a model for building other libraries that use OpenGL for rendering

In the client server environment, a utility library raises an issue utility library commands are converted into OpenGL commands on the client, if the server computer is more powerful than the client, the client side conversion might have been more effectively carried out on the server This dilemma arises not just with OpenGL but with any library in which the client and server may be distinct computers In OpenGL the base functionality reflects the functions efficiently performed by advanced graphics subsystems because no matter what the power of the server computer relative to the client the server s graphics subsystem is assumed to efficiently perform the functions it provides If in the future for instance graphics subsystems commonly provide full trimmed NURBS support, then such functionality should likely migrate from the Utility Library to OpenGL itself Such a change would not cause any disruption to the rest of the OpenGL API another block would simply be added to the left side in Figure 1

4.3 Fine-Grained Control

In order to minimize the requirements on how an application using the API must store and present its data the API must provide a means to specify individual components of geometric objects and operations on them This fine grained control is required so that these components and operations may be specified in any order and so that control of rendering operations is flexible enough to accommodate the requirements of diverse applications

Vertices and Associated Data

In OpenGL most geometric objects are drawn by enclosing a se ries of coordinate sets that specify vertices and optionally normals texture coordinates and colors between **glBegin/glEnd** command pairs For example to specify a triangle with vertices at (0 0 0) (0 1 0) and (1 0 1) one could write

```
glBegin(GL_POLYGON)
  glVertex3i(0 0 0)
  glVertex3i(0 1 0)
  glVertex3i(1 0 1)
glEnd()
```

Each vertex may be specified with two three or four coordinates (four coordinates indicate a homogeneous three-dimensional loca tion) In addition a *current normal current texture coordinates* and *current color* may be used in processing each vertex OpenGL uses normals in lighting calculations the current normal is a three dimensional vector that may be set by sending three coordinates that specify it Color may consist of either red green blue and alpha values (when OpenGL has been initialized to RGBA mode) or a single color index value (when initialization specified color index mode) One two three or four texture coordinates determine how a texture image maps onto a primitive

Each of the commands that specify vertex coordinates normals colors or texture coordinates comes in several flavors to accomodate differing application s data formats and numbers of coordinates Data may also be passed to these commands either as an argument list or as a pointer to a block of storage containing the data The variants are distinguished by mnemonic suffixes

Using a procedure call to specify each individual group of data that together define a primitive means that an application may store data in any format and order that it chooses data need not be stored in a form convenient for presentation to the graphics API because OpenGL accomodates almost any data type and format using the appropriate combination of data specification procedures Another advantage of this scheme is that by simply combining calls in the appropriate order different effects may be achieved Figure 2 shows an example of a uniformly colored triangle obtained by specifying a single color that is inherited by all vertices of the triangle a smooth shaded triangle is obtained by respecifying a color before each vertex Not every possible data format is supported (byte values may not be given for vertex coordinates for instance) only because it was found from experience with IRIS GL that not all formats are used Adding the missing formats in the future however would be a trivial undertaking

One disadvantage of using procedure calls on such a fine grain is that it may result in poor performance if procedure calls are costly In such a situation an interface that specifies a format for a block of data that is sent all at once may have a performance advantage The difficulty with specifying a block of data however is that it either constrains the application to store its data in one of the supported formats or it requires the application to copy its data into a block structured in one of those formats resulting in inefficiency (Allowing any format arising from an arbitrary combination of individual data types is impractical because there are so many combinations)

In OpenGL, the maximum flexibility provided by individual pro cedure calls was deemed more important than any inefficiency in duced by using those calls This decision is partly driven by the consideration that modern compilers and computer hardware have improved to the point where procedure calls are usually relatively

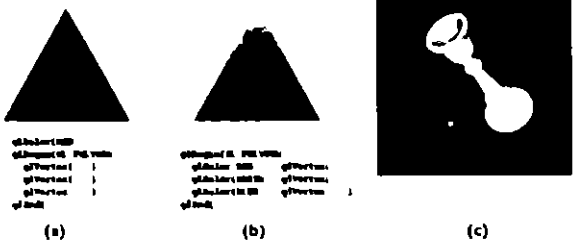


Figure 2 (a) A uniformly colored triangle (b) A Gouraud shaded triangle (c) A scene consisting of many lit, shaded polygons

inexpensive especially when compared with the work necessary to process the geometric data contained in the call This is one area in which OpenGL differs significantly from PEX with PEX a prim itive s vertices (and associated data) are generally presented all at once in a single array If it turns out that fine grained procedure calls are too expensive then it may be necessary to add a few popular block formats to the OpenGL API or to provide a mechanism for defining such formats

4 4 Modal

As a consequence of fine grained control OpenGL maintains con siderable state or modes that determines how primitives are ren dered This state is present in lieu of having to present a large amount of information with each primitive that would describe the settings for all the operations to which the primitive would be subjected Presenting so much information with each primitive is tedious and would result in excessive data being transmitted from client to server Therefore essentially no information is presented with a primitive except what is required to define it Instead a considerable propor tion of OpenGL commands are devoted to controlling the settings of rendering operations

One difficulty with a modal API arises in implementations in which separate processors (or processes) operate in parallel on dis tinct primitives In such cases a mode change must be broadcast to all processors so that each receives the new parameters before it pro cesses its next primitive A mode change is thus processed serially halting primitive processing until all processors have received the change and reducing performance accordingly One way to lessen the impact of mode changes in such a system is to insert a processor that distributes work among the parallel processors This processor can buffer up a string of mode changes transmitting the changes all at once only when another primitive finally arrives[11]

Another way to handle state changes relies on defining groups of named state settings which can then be invoked simply by pro viding the appropriate name (this is the approach taken by X and PEX) With this approach a single command naming the state set ting changes the server s settings This approach was rejected for OpenGL for several reasons Keeping track of a number of state vec tors (each of which may contain considerable information) may be impractical on a graphics subsystem with limited memory Named state settings also conflict with the emphasis on fine grained con trol there are cases as when changing the state of a single mode when transmitting the change directly is more convenient and effi cient than first setting up and then naming the desired state vector Finally the named state setting approach may still be used with OpenGL by encapsulating state changing commands in display lists (see below)

The Matrix Stack

Three kinds of transformation matrices are used in OpenGl the *model view* matrix which is applied to vertex coordinates the *tex ture* matrix which is applied to texture coordinates and the *pro jection* matrix which describes the viewing frustum and is applied to vertex coordinates after they are transformed by the model view matrix Each of these matrices is 1×1

Any of one these matrices may be loaded with or multiplied by a general transformation commands are provided to specify the special cases of rotation translation and scaling (since these cases take only a few parameters to specify rather than the 16 required for a general transformation) A separate command controls a mode indicating which matrix is currently affected by any of these manip ulations In addition each matrix type actually consists of a stack of matrices that can be pushed or popped The matrix on the top of the stack is the one that is applied to coordinates and that is affected by matrix manipulation commands

The retained state represented by these three matrix stacks simpli fies specifying the transformations found in hierarchical graphical data structures Other graphics APIs also employ matrix stacks but often only as a part of more general attribute structures But OpenGL is unique in providing three kinds of matrices which can be manipulated with the same commands The texture matrix for instance can be used to effectively rotate or scale a texture image applied to primitive and when combined with perspective viewing transformations can even be used to obtain projective texturing ef fects such as spotlight simulation and shadow effects using shadow maps[14]

State Queries and Attribute Stacks

The value of nearly any OpenGL parameter may be obtained by an appropriate *get* command There is also a stack of parameter values that may be pushed and popped For stacking purposes all parame ters are divided into 21 functional groups any combination of these groups may be pushed onto the attribute stack in one operation (a pop operation automatically restores only those values that were last pushed) The get commands and parameter stacks are required so that various libraries may make use of OpenGL efficiently without interfering with one another

4.5 Framebuffer

Most of OpenGL requires that the graphics hardware contain a framebuffer This is a reasonable requirement since nearly all inter active graphics applications (as well as many non interactive ones) run on systems with framebuffers Some operations in OpenGL are achieved only through exposing their implementation using a framebuffer (transparency using alpha blending and hidden sur face removal using depth buffering are two examples) Although OpenGL may be used to provide information for driving such de vices as pen plotters and vector displays such use is secondary

Multipass Algorithms

One useful effect of making the framebuffer explicit is that it en ables the use of multipass algorithms in which the same primitives are rendered several times One example of a multipass algorithm

employs an *accumulation buffer*[3] a scene is rendered several times each time with a slightly different view and the results averaged in the framebuffer Depending on how the view is altered on each pass this algorithm can be used to achieve full window antialiasing depth of field effects motion blur or combinations of these Multipass algorithms are simple to implement in OpenGL because only a small number of parameters must be manipulated between passes and changing the values of these parameters is both efficient and without side effects on other parameters that must remain constant

Invariance

Consideration of multipass algorithms brings up the issue of how what is drawn in the framebuffer is or is not affected by changing parameter values If for instance changing the viewpoint affected the way in which colors were assigned to primitives the accumulation buffer algorithm would not work For a more plausible example if some OpenGL feature is not available in hardware then an OpenGL implementation must switch from hardware to software when that feature is switched on Such a switch may significantly affect what eventually reaches the framebuffer because of slight differences in the hardware and software implementations

The OpenGL specification is not pixel exact, it does not indicate the exact values to which certain pixels must be set given a certain input. The reason is that such specification besides being difficult, would be too restrictive Different implementations of OpenGL run on different hardware with different floating point formats rasterization algorithms and framebuffer configurations It should be possible nonetheless to implement a variety of multipass algorithms and expect to get reasonable results

For this reason the OpenGL specification gives certain invariance rules that dictate under what circumstances one may expect identical results from one particular implementation given certain inputs (implementations on different systems are never required to produce identical results given identical inputs) These rules typically indicate that changing parameters that control an operation cannot affect the results due to any other operation but that such invariance is not required when an operation is turned on or off This makes it possible for an implementation to switch from hardware to software when a mode is invoked without breaking invariance On the other hand a programmer may still want invariance even when toggling some mode. To accommodate this case any operation covered by the invariance rules admits a setting of its controlling parameters that cause the operation to act as if it were turned off even when it is on A comparison for instance may be turned on or off but when on the comparison that is performed can be set to always (or never) pass

4 6 Not Programmable

OpenGL does not provide a programming language Its function may be controlled by turning operations on or off or specifying parameters to operations but the rendering algorithms are essentially fixed One reason for this decision is that, for performance reasons graphics hardware is usually designed to apply certain operations in a specific order replacing these operations with arbitrary algorithms is usually infeasible Programmability would conflict with keeping the API close to the hardware and thus with the goal of maximum performance

The Graphics Pipeline and Per Fragment Operations

The model of command execution in OpenGL is that of a pipeline with a fixed topology (although stages may be switched in or out) The pipeline is meant to mimic the organization of graphics subsystems The final stages of the pipeline for example consist of a series of tests on and modifications to fragments before they are eventually placed in the framebuffer To draw a complex scene in a short amount of time many fragments must pass through these final stages on their way to the framebuffer leaving little time to process each fragment Such high *fill rates* demand special purpose hardware that can only perform fixed operations with minimum access to external data

Even though fragment operations are limited many interesting and useful effects may be obtained by combining the operations appropriately Per fragment operations provided by OpenGL include

- alpha blending blend a fragment s color with that of the corresponding pixel in the framebuffer based on an alpha value
- depth test compare a depth value associated with a fragment with the corresponding value already present in the framebuffer and discard or keep the fragment based on the outcome of the comparison
- stencil test compare a reference value with a corresponding value stored in the framebuffer and update the value or discard the fragment based on the outcome of the comparison

Alpha blending is useful to achieve transparency or to blend a fragment s color with that of the background when antialiasing the depth test can effect depth buffering (and thus hidden surface removal) the stencil test can be used for a number of effects[12] including highlighting interference regions and simple CSG (Constructive Solid Geometry) operations These (and other) operations may be combined to achieve for instance transparent interference regions with hidden surfaces removed or any number of other effects

The OpenGL graphics pipeline also induces a kind of orthogonality among primitives Each vertex whether it belongs to a point, line segment or polygon primitive is treated in the same way its coordinates are transformed and lighting (if enabled) assigns it a color The primitive defined by these vertices is then rasterized and converted to fragments as is a bitmap or image rectangle primitive All fragments no matter what their origin are treated identically This homogeneity among operations removes unneeded special cases (for each primitive type) from the pipeline It also makes natural the combination of diverse primitives in the same scene without having to set special modes for each primitive type

4 7 Geometry and Images

OpenGL provides support for handling both 3D (and 2D) geometry and 2D images An API for use with geometry should also provide support for writing reading and copying images because geometry and images are often combined as when a 3D scene is laid over a background image Many of the per fragment operations that are applied to fragments arising from geometric primitives apply equally well to fragments corresponding to pixels in an image making it easy to mix images with geometry For example a triangle may be blended with an image using alpha blending OpenGL supports a number of image formats and operations on image components (such as lookup tables) to provide flexibility in image handling

Texture Mapping

Texture mapping provides an important link between geometry and images by effectively applying an image to geometry OpenGL makes this coupling explicit by providing the same formats for specifying texture images as for images destined for the framebuffer

Besides being useful for adding realism to a scene (Figure 3a) texture mapping can be used to achieve a number of other useful effects[4] Figures 3b and 3c show two examples in which the texture coordinates that index a texture image are generated from vertex coordinates OpenGL s orthogonality makes achieving such effects with texture mapping simply a matter of enabling the appropriate modes and loading the appropriate texture image without affecting the underlying specification of the scene

4 8 Immediate Mode and Display Lists

The basic model for OpenGL command interpretation is immediate mode in which a command is executed as soon as the server receives it vertex processing for example may begin even before specification of the primitive of which it is a part has been completed Immediate mode execution is well suited to interactive applications in which primitives and modes are constantly altered In OpenGL the fine grained control provided by immediate mode is taken as far as possible even individual lighting parameters (the diffuse reflectance color of a material for instance) and texture images are set with individual commands that have immediate effect

While immediate mode provides flexibility its use can be inefficient if unchanging parameters or objects must be respecified To accommodate such situations OpenGL provides display lists A display list encapsulates a sequence of OpenGL commands (all but a handful of OpenGL commands may be placed in a display list) and is stored on the server The display list is given a numeric name by the application when it is specified the application need only name the display list to cause the server to effectively execute all the commands contained within the list This mechanism provides a straightforward effective means for an application to transmit a group of commands to the server just once even when those same commands must be executed many times

Display List Optimization

Accumulating commands into a group for repeated execution presents possibilities for optimization Consider for example specifying a texture image Texture images are often large requiring a large and therefore possibly slow data transfer from client to server (or from the server to its graphics subsystem) whenever the image is respecified For this reason some graphics subsystems are equipped with sufficient storage to hold several texture images simultaneously If the texture image definition is placed in a display list then the server may be able to load that image just once when it is specified When the display list is invoked (or re invoked) the server simply indicates to the graphics subsystem that it should use the texture image already present in its memory thus avoiding the overhead of respecifying the entire image

Examples like this one indicate that display list optimization is required to achieve the best performance In the case of texture image loading the server is expected to recognize that a display list contains texture image information and to use that information appropriately This expectation places a burden on the OpenGL implementor to make sure that special display list cases are treated as efficiently as possible It also places a burden on the application writer to know to use display lists in cases where doing so could improve performance Another possibility would have been to introduce special commands for functions that can be poor performers in immediate mode But such specialization would clutter the API and blur the clear distinction between immediate mode and display lists

Display List Hierarchies

Display lists may be redefined in OpenGL but not edited The lack of editing simplifies display list memory management on the server eliminating the penalty that such management would incur One display list may however invoke others An effect similar to display list editing may thus be obtained by (1) building a list that invokes a number of subordinate lists (2) redefining the subordinate lists This redefinition is possible on a fine grain a subordinate display list may contain anything (even nothing) including just a single vertex or color command

There is no automatic saving or restoring of modes associated with display list execution (If desired such saving and restoring may be performed explicitly by encapsulating the appropriate commands in the display list) This allows the highest possible performance in executing a display list since there is almost no overhead associated with its execution It also simplifies controlling the modal behavior of display list hierarchies only modes explicitly set are affected

Lack of automatic modal behavior in display lists also has a disadvantage it is difficult to execute display lists in parallel since the modes set in one display list must be in effect before a following display list is executed In OpenGL display lists are generally not used for defining whole scenes or complex portions of scenes but rather for encapsulating groups of frequently repeated mode setting commands (describing a texture image for instance) or commands describing simple geometry (the polygons approximating a torus for instance)

4 9 Depth buffer

The only hidden surface removal method directly provided by OpenGL is the depth (or) buffer This assumption is in line with that of the graphics hardware containing a framebuffer Other hidden surface removal methods may be used with OpenGL (a BSP tree[2] coupled with the painter s algorithm for instance) but it is assumed that such methods are never supported in hardware and thus need not be supported explicitly by OpenGL

4 10 Local Shading

The only shading methods provided by OpenGL are local That is methods for determining surface color such as ray tracing or radiosity that require obtaining information from other parts of the scene are not directly supported The reason is that such methods require knowledge of the global scene database but so far specialized graphics hardware is structured as a pipeline of localized operations and does not provide facilities to store and traverse the large amount of data necessary to represent a complex scene Global shading methods may be used with OpenGL only if the shading can be precomputed and the results associated with graphical objects before they are transmitted to OpenGL

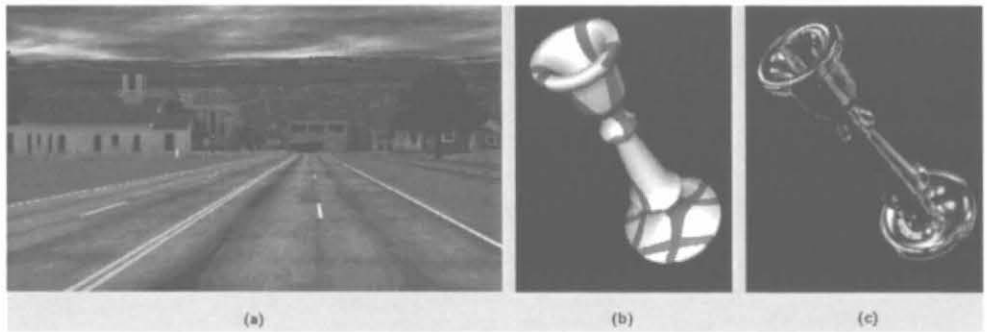


Figure 3. (a) A scene with a number of textures mapped onto primitives. (b) Contouring achieved with texture mapping and a texture coordinate generation function. (c) Reflectance mapping with a texture coordinate generation function.

4.11 Rendering Only

OpenGL provides access to rendering operations only. There are no facilities for obtaining user input from such devices as keyboards and mice, since it is expected that any system (in particular, a window system) under which OpenGL runs must already provide such facilities. Further, the effects of OpenGL commands on the framebuffer are ultimately controlled by the window system (if there is one) that allocates framebuffer resources. The window system determines which portions of the framebuffer OpenGL may access and communicates to OpenGL how those portions are structured. These considerations make OpenGL window system independent.

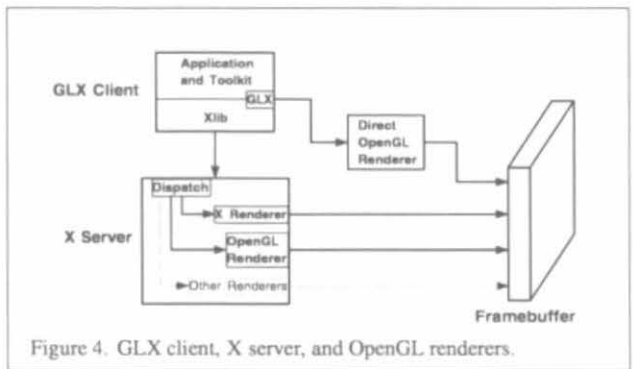


Figure 4. GLX client, X server, and OpenGL renderers.

Integration in X

X provides both a procedural interface and a network protocol for creating and manipulating framebuffer windows and drawing certain 2D objects into those windows. OpenGL is integrated into X by making it a formal X extension called *GLX*. GLX consists of about a dozen calls (with corresponding network encodings) that provide a compact, general embedding of OpenGL in X. As with other X extensions (two examples are Display PostScript and PEX), there is a specific network protocol for OpenGL rendering commands encapsulated in the X byte stream.

OpenGL requires a region of a framebuffer into which primitives may be rendered. In X, such a region is called a *drawable*. A *window*, one type of drawable, has associated with it a *visual* that describes the window's framebuffer configuration. In GLX, the visual is extended to include information about OpenGL buffers that are not present in unadorned X (depth, stencil, accumulation, front, back, etc.).

X also provides a second type of drawable, the *pixmap*, which is an off-screen framebuffer. GLX provides a *GLX pixmap* that corresponds to an X pixmap, but with additional buffers as indicated by some visual. The GLX pixmap provides a means for OpenGL applications to render off-screen into a software buffer.

To make use of an OpenGL-capable drawable, the programmer creates an OpenGL *context* targeted to that drawable. When the context is created, a copy of an OpenGL renderer is initialized with the visual information about the drawable. This OpenGL renderer is conceptually (if not actually) part of the X server, so that, once created, an X client may *connect* to the OpenGL context and issue OpenGL commands (Figure 4). Multiple OpenGL contexts may be created that are targeted to distinct or shared drawables. Any OpenGL-capable drawable may also be used for standard X drawing (those buffers of the drawable that are unused by X are ignored by X).

A GLX client that is running on a computer of which the graphics subsystem is a part may avoid passing OpenGL tokens through the X server. Such direct rendering may result in increased graphics performance since the overhead of token encoding, decoding, and dispatching is eliminated. Direct rendering is supported but not required by GLX. Direct rendering is feasible because sequentiality need not be maintained between X commands and OpenGL commands except where commands are explicitly synchronized.

Because OpenGL comprises rendering operations only, it fits well into already existing window systems (integration into Windows is similar to that described for X) without duplicating operations already present in the window system (like window control or mouse event generation). It can also make use of window system features such as off-screen rendering, which, among other uses, can send the results of OpenGL commands to a printer. Rendering operations provided by the window system may even be interspersed with those of OpenGL.

4.12 API not Protocol

PEX is primarily specified as a network protocol; PEXlib is a presentation of that protocol through an API. OpenGL, on the other

hand is primarily specified as an API the API is encoded in a specified network protocol when OpenGL is embedded in a system (like X) that requires a protocol One reason for this preference is that an applications programmer works with the API and not with a protocol Another is that different platforms may admit different protocols (X places certain constraints on the protocol employed by an X extension while other window systems may impose different constraints) This means that the API is constant across platforms even when the protocol cannot be thereby making it possible to use the same source code (at least for the OpenGL portion) without re gard for any particular protocol Further when the client and server are the same computer OpenGL commands may be transmitted directly to a graphics subsystem without conversion to a common encoding

Interoperability between diverse systems is not compromised by preferring an API specification over one for a protocol Tests in which an OpenGL client running under one manufacturer s imple mentation was connected to another manufacturer s OpenGL server have provided excellent results

5 Example Three Kinds of Text

To illustrate the flexibility of OpenGL in performing different types of rendering tasks we outline three methods for the particular task of displaying text The three methods are using bitmaps using line segments to generate outlined text and using a texture to generate antialiased text

The first method defines a font as a series of display lists each of which contains a single bitmap

```
for i = start + a   to  start + z  {
  glBeginList(i)
    glBitmap(      )
  glEndList()
}
```

`glBitmap` specifies both a pointer to an encoding of the bitmap and offsets that indicate how the bitmap is positioned relative to previous and subsequent bitmaps In GLX the effect of defining a number of display lists in this way may also be achieved by calling `glXUseXFont` `glXUseXFont` generates a number of display lists each of which contains the bitmap (and associated offsets) of a single character from the specified X font In either case the string Bitmapped Text whose origin is the projection of a location in 3D is produced by

```
glRasterPos3i(x  y  z)
glListBase(start)
glCallLists( Bitmapped Text  14 GL_BYTE)
```

See Figure 5a `glListBase` sets the display list base so that the subsequent `glCallLists` references the characters just defined `glCallLists` invokes a series of display lists specified in an array each value in the array is added to the display list base to obtain the number of the display list to use In this case the array is an array of bytes representing a string The second argument to `glCallLists` indicates the length of the string the third argument indicates that the string is an array of 8 bit bytes (16 and 32 bit integers may be used to access fonts with more than 256 characters)

The second method is similar to the first but uses line segments to outline each character Each display list contains a series of line segments

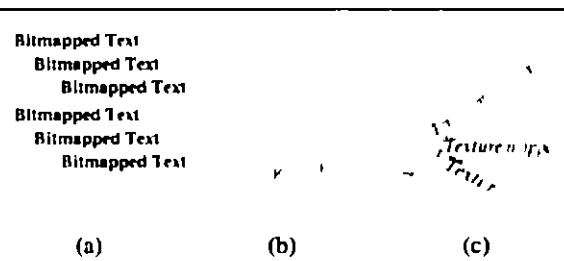


Figure 5 (a) Bitmap example (b) Stroke font example (c) Texture mapped font example

```
glTranslate(ox  oy  0)
glBegin(GL_LINES)
  glVertex(    )

glEnd()
glTranslate(dx-ox  dy-oy  0)
```

The initial `glTranslate` updates the transformation matrix to position the character with respect to a character origin The final `glTranslate` updates that character origin in preparation for the following character A string is displayed with this method just as in the previous example but since line segments have 3D position the text may be oriented as well as positioned in 3D (Figure 5b) More generally the display lists could contain both polygons and line segments and these could be antialiased

Finally a different approach may be taken by creating a texture image containing an array of characters A certain range of texture coordinates thus corresponds to each character in the texture image Each character may be drawn in any size and in any 3D orientation by drawing a rectangle with the appropriate texture coordinates at its vertices

```
glTranslate(ox  oy  0)
glBegin(GL_QUADS)
  glTexCoord(      )
  glVertex(      )

glEnd()
glTranslate(dx-ox  dy-oy  0)
```

If each group of commands for each character is enclosed in a display list, and the commands for describing the texture image itself (along with the setting of the list base) are enclosed in another display list called TEX then the string "Texture mapped text!!" may be displayed by

```
glCallList(TEX)
glCallLists( Texture mapped text!!  21
                                  GL_BYTE)
```

One advantage of this method is that by simply using appropriate texture filtering the resulting characters are antialiased (Figure 5c)

6 Conclusion

OpenGL is a 3D graphics API intended for use in interactive applica tions It has been designed to provide maximum access to hardware graphics capabilities no matter at what level such capabilities are

available This efficiency stems from a flexible interface that pro vides direct control over fundamental operations OpenGL does not enforce a particular method of describing 3D objects and how they should appear but instead provides the basic means by which those objects no matter how described may be rendered Because OpenGL imposes minimum structure on 3D rendering it provides an excellent base on which to build libraries for handling structured geometric objects no matter what the particular structures may be

The goals of high performance feature orthogonality interoper ability implementability on a variety of systems and extensibility have driven the design of OpenGL s API We have shown the effects of these and other considerations on the presentation of rendering operations in OpenGL The result has been a straightforward API with few special cases that should be easy to use in a variety of applications

Future work on OpenGL is likely to center on improving im plementations through optimization and extending the API to han dle new techniques and capabilities provided by graphics hardware Likely candidates for inclusion are image processing operators new texture mapping capabilities and other basic geometric primitives such as spheres and cylinders We believe that the care taken in the design of the OpenGL API will make these as well as other extensions simple and will result in OpenGL s remaining a useful 3D graphics API for many years to come

References

[1] Kurt Akeley RealityEngine graphics In *SIGGRAPH 93 Con ference Proceedings* pages 109–116 August 1993

[2] H Fuchs Z M Kedem and B F Naylor On visible surface generation by a priori tree structures *Computer Graphics (SIGGRAPH 80 Proceedings)* 14(3) 124–133 July 1980

[3] Paul Haeberli and Kurt Akeley The accumulation buffer Hardware support for high-quality rendering *Computer Graphics (SIGGRAPH 90 Proceedings)* 24(2) 309–318 July 1990

[4] Paul Haeberli and Mark Segal Texture mapping as a fun damental drawing primitive In *Proceedings of the Fourth Eurographics Workshop on Rendering* pages 259–266 June 1993

[5] Adobe Systems Incorporated *PostScript Language Reference Manual* Addison Wesley Reading Mass 1986

[6] International Standards Organization International standard information processing systems — computer graphics — graphical kernel system for three dimensions (GKS 3D) func tional description Technical Report ISO Document Number 9905 1988(E) American National Standards Institute New York 1988

[7] Jeff Stevenson PEXlib specification and C language binding version 5 1P *The X Resource* Special Issue B September 1992

[8] Jackie Neider Mason Woo and Tom Davis *OpenGL Pro gramming Guide* Addison Wesley Reading Ma 1993

[9] Adrian Nye *X Window System User s Guide* volume 3 of *The Definitive Guides to the X Window System* O Reilly and Associates Sebastapol Ca 1987

[10] Paula Womack ed PEX protocol specification and encoding version 5 1P *The X Resource* Special Issue A May 1992

[11] PHIGS+ Committee Andries van Dam chair PHIGS+ functional description revision 3 0 *Computer Graphics* 22(3) 125–218 July 1988

[12] Jarek Rossignac Abe Megahed and Bengt Olaf Schneider Interactive inspection of solids Cross sections and interfer ences *Computer Graphics (SIGGRAPH 92 Proceedings)* 26(2) 353–360 July 1992

[13] Mark Segal and Kurt Akeley The OpenGL graphics system A specification Technical report, Silicon Graphics Computer Systems Mountain View Ca 1992

[14] Mark Segal Carl Korobkin Rolf van Widenfelt, Jim Foran and Paul Haeberli Fast shadows and lighting effects using texture mapping *Computer Graphics (SIGGRAPH 92 Pro ceedings)* 26(2) 249–252 July 1992

[15] Paul S Strauss and Rikk Carey An object-oriented 3D graph ics toolkit. *Computer Graphics (SIGGRAPH 92 Proceed ings)* 26(2) 341–349 July 1992

[16] Steve Upstill *The RenderMan Companion* Addison Wesley Reading Mass 1990

[17] Garry Wiegand and Bob Covey *HOOPS Reference Manual, Version 3 0* Ithaca Software 1991

The OpenGL Graphics Interface

Mark Segal
Kurt Akeley
Silicon Graphics Computer Systems
2011 N Shoreline Blvd Mountain View, CA 94039
USA

Abstract

Graphics standards are receiving increased attention in the computer graphics community as more people write programs that use 3D graphics and as those already possessing 3D graphical programs want those programs to run on a variety of computers

OpenGL is an emerging graphics standard that provides advanced rendering features while maintaining a simple programming model Its procedural interface allows a graphics programmer to describe rendering tasks whether simple or complex easily and efficiently Because OpenGL is rendering only it can be incorporated into any window system (and has been into the X Window System and the soon to be released Windows NT) or can be used without a window system Finally OpenGL is designed so that it can be implemented to take advantage of a wide range of graphics hardware capabilities from a basic framebuffer to the most sophisticated graphics subsystems

1 Introduction

Computer graphics (especially 3D graphics and interactive 3D graphics in particular) is finding its way into an increasing number of applications from simple graphing programs for personal computers to sophisticated modeling and visualization software on workstations and supercomputers As the interest in computer graphics has grown so has the desire to be able to write an application so that it runs on a variety platforms with a range of graphical capabilites A graphics standard eases this task by eliminating the need to write a distinct graphics driver for each platform on which the application is to run

Several standards have succeeded in integrating specific domains of 2D graphics The PostScript page description language[4] has become widely accepted mak

ing it relatively easy to electronically exchange and to a limited degree manipulate static documents containing both text and 2D graphics The X window system[7] has become standard for UNIX workstations A programmer uses X to obtain a window on a graphics display into which either text or 2D graphics may be drawn X also provides a standard means for obtaining user input from such devices as key boards and mice The adoption of X by most workstation manufacturers means that a single program can produce 2D graphics or obtain user input on a variety of work stations by simply recompiling the program This integration even works across a network the program may run on one workstation but display on and obtain user input from another even if the workstations on either end of the network are made by different companies

For 3D graphics several standards have been proposed but none has (yet) gained wide acceptance One relatively well known system is PHIGS (Programmer s Hierarchical Interactive Graphics System) Based on GKS[5] (Graphics Kernel System) PHIGS is an ANSI (American National Standards Institute) standard PHIGS (and its descendant PHIGS+[9]) provides a means to manipulate and draw 3D objects by encapsulating object descriptions and attributes into a *display list* that is then referenced when the object is displayed or manipulated One advantage of the display list is that a complex object need be described only once even if it is to be displayed many times This is especially important if the object to be displayed must be transmitted across a low bandwidth channel (such as a network) One disadvantage of a display list is that it can require considerable effort to re specify the object if it is being continually modified as a result of user interaction Another difficulty with PHIGS and PHIGS+ (and with GKS) is that they lack support for advanced rendering features such as texture mapping

PEX[8] which is often said to be an acronym for PHIGS Extension to X extends X to include the ability to manipulate and draw 3D objects (PEXlib[6] is the programmer s interface to the PEX protocol) Among other extensions PEX adds *immediate mode* rendering to PHIGS meaning that objects can be displayed as they are described rather than having to first complete a display list One difficulty with PEX has been that different suppliers of the PEX interface have chosen to support different features making program portability problematic PEX also lacks advanced rendering feaures and is available only to users of X

2 OpenGL

OpenGL (GL for 'Graphics Library) provides advanced rendering features in either immediate mode or display list mode While OpenGL is a relatively new stan

dard it is very similar in both its functionality and its interface to Silicon Graphics IRIS GL and there are many succesful 3D applications that currently use IRIS GL for their 3D rendering

Like the graphics systems already discussed OpenGL is a software interface to graphics hardware The interface consists of a set of several hundred procedures and functions that allow a programmer to specify the objects and operations involved in producing high quality graphical images specifically color images of three dimensional objects Like PEX OpenGL integrates 3D drawing into X but can also be integrated into other window systems (e g Windows/NT) or can be used without a window system

OpenGL draws *primitives* into a framebuffer subject to a number of selectable modes Each primitive is a point line segment polygon pixel rectangle or bitmap Each mode may be changed independently the setting of one does not affect the settings of others (although many modes may interact to determine what eventually ends up in the framebuffer) Modes are set primitives specified and other OpenGL operations described by sending *commands* in the form of function or procedure calls

Geometric primitives (points line segments and polygons) are defined by a group of one or more *vertices* A vertex defines a point an endpoint of an edge or a corner of a polygon where two edges meet Data (consisting of positional coordinates colors normals and texture coordinates) are associated with a vertex and each vertex is processed independently in order and in the same way The only exception to this rule is if the group of vertices must be *clipped* so that the indicated primitive fits within a specified region in this case vertex data may be modified and new vertices created The type of clipping depends on which primitive the group of vertices represents

OpenGL provides direct control over the fundamental operations of 3D and 2D graphics This includes specification of such parameters as transformation matrices lighting equation coefficients antialiasing methods and pixel update operators It does not provide a means for describing or modeling complex geometric objects Another way to describe this situation is to say that OpenGL provides mechanisms to describe how complex geometric objects are to be rendered rather than mechanisms to describe the complex objects themselves

The model for interpretation of OpenGL commands is client server That is a program (the client) issues commands and these commands are interpreted and processed by OpenGL (the server) The server may or may not operate on the same computer as the client

The effects of OpenGL commands on the framebuffer are ultimately controlled by the window system that allocates framebuffer resources It is the window sys

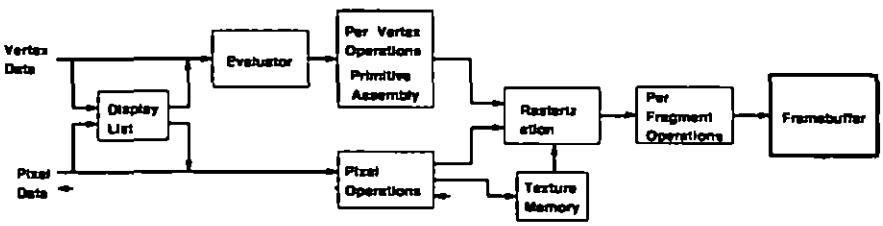


Figure 1 Block diagram of OpenGL

tem that determines which portions of the framebuffer that OpenGL may access at any given time and that communicates to OpenGL how those portions are struc tured Similarly display of framebuffer contents on a CRT monitor (including the transformation of individual framebuffer values by such techniques as gamma cor rection) is not addressed by OpenGL Framebuffer configuration occurs outside of OpenGL in conjunction with the window system the initialization of an OpenGL context occurs when the window system allocates a window for OpenGL rendering Additionally OpenGL has no facilities for obtaining user input since it is expected that any window system under which OpenGL runs must already provide such fa cilities These considerations make OpenGL independent of any particular window system

3 Basic OpenGL Operation

Figure 1 shows a schematic diagram of OpenGL Commands enter OpenGL on the left Most commands may be accumulated in a *display list* for processing at a later time Otherwise commands are effectively sent through a processing pipeline

The first stage provides an efficient means for approximating curve and surface geometry by evaluating polynomial functions of input values The next stage oper ates on geometric primitives described by vertices points line segments and poly gons In this stage vertices are transformed and lit and primitives are clipped to a viewing volume in preparation for the next stage rasterization The rasterizer pro duces a series of framebuffer addresses and values using a two dimensional descrip- tion of a point line segment or polygon Each *fragment* so produced is fed to the next stage that performs operations on individual fragments before they finally alter the framebuffer These operations include conditional updates into the framebuffer

based on incoming and previously stored depth values (to effect depth buffering) blending of incoming fragment colors with stored colors as well as masking and other logical operations on fragment values

Finally pixel rectangles and bitmaps bypass the vertex processing portion of the pipeline to send a block of fragments directly through rasterization to the individual fragment operations eventually causing a block of pixels to be written to the framebuffer Values may also be read back from the framebuffer or copied from one portion of the framebuffer to another These transfers may include some type of decoding or encoding

3 1 The OpenGL Utility Library

A guiding principle in the design of OpenGL has been to provide program portability without mandating how higher level graphical objects must be described As a result the basic OpenGL interface does not support some geometric objects that are traditionally associated with graphics standards For instance an OpenGL implementation need not render concave polygons One reason for this omission is that concave polygon rendering algorithms are of necessity more complex than those for rendering convex polygons and different concave polygon algorithms may be appropriate in different domains In particular if a concave polygon is to be drawn more than once it is more efficient to first decompose it into convex polygons (or triangles) once and then draw the convex polygons

A general concave polygon decomposer is provided as part of the OpenGL Utility Library which is provided with every OpenGL implementation The Utility Library also provides an interface built on OpenGL s polynomial evaluators to describe and display NURBS curves and surfaces (with domain space trimming) as well as a means for rendering spheres cones and cylinders The Utility Library serves both as a means to render useful geometric objects and as a model for building other libraries that use OpenGL for rendering

4 The OpenGL Pipeline

4 1 Vertices and Primitives

In OpenGL most geometric objects are drawn by enclosing a series of coordinate sets that specify vertices and optionally normals texture coordinates and colors between **glBegin/glEnd** command pairs For example, to specify a triangle with vertices at (0 0 0) (0 1 0) and (1 0 1) one could write

Object	Interpretation of Vertices
points	each vertex describes the location of a point
line strip	series of connected line segments each vertex after first describes the endpoint of next segment
line loop	same as line strip but final segment added from final vertex to first vertex
separate line	each pair of vertex describes a line segment
polygon	line loop formed by vertices describes the boundary of a convex polygon
triangle strip	each vertex after the first two describes a triangle given by that vertex and the previous two
triangle fan	each vertex after the first two describes a triangle given by that vertex, the previous vertex, and the first vertex
separate triangle	each consecutive triad of vertices describes a triangle
quadrilateral strip	each pair of vertices after the first two describes a quadrilateral given by that pair and the previous pair
independent quad	each consecutive group of four vertices describes a quadrilateral

Table 1 **glBegin/glEnd** objects

```
glBegin(GL_POLYGON)
  glVertex3i(0 0 0)
  glVertex3i(0 1 0)
  glVertex3i(1 0 1)
glEnd()
```

The ten geometric objects that are drawn this way are summarized in Table 4 1 This particular group of objects was selected because each object s geometry is specified by a simple list of vertices because each admits an efficient rendering algorithm and because it was determined that taken together these objects satisfy the needs of nearly all graphics applications

Each vertex may be specified with two three or four coordinates (four coor dinates indicate a homogeneous three dimensional location) In addition a *current normal current texture coordinates* and *current color* may be used in processing each vertex OpenGL uses normals in lighting calculations the current normal is a three dimensional vector that may be set by sending three coordinates that specify it Color may consist of either red green blue and alpha values (when OpenGL has been initialized to RGBA mode) or a single color index value (when initialization specified color index mode) One two three or four texture coordinates determine how a texture image maps onto a primitive

Each of the commands that specify vertex coordinates normals colors or tex ture coordinates comes in several flavors to accomodate differing application s data formats and numbers of coordinates Data may also be passed to these commands either as an argument list or as a pointer to a block of storage containing the data The variants are distinguished (in the C language) by mnemonic suffixes

Most OpenGL commands that do not specify vertices and associated informa tion may not appear between **glBegin** and **glEnd** This restriction allows imple mentations to run in an optimized mode while processing primitive specifications

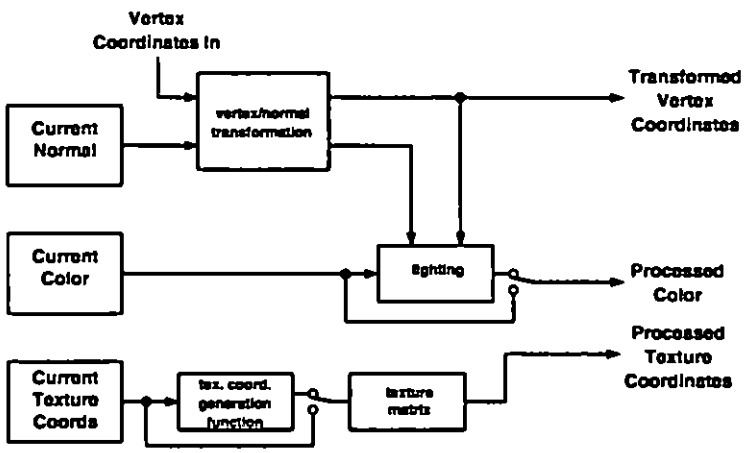


Figure 2 Association of current values with a vertex

so that primitives may be processed as efficiently as possible

When a vertex is specified the current color, normal, and texture coordinates are used to obtain values that are then associated with the vertex (Figure 2) The vertex itself is transformed by the *model view matrix*, a 4×4 matrix which can represent both linear and translational transformations The color is obtained from either computing a color from lighting or, if lighting is disabled, from the current color Texture coordinates are similarly passed through a *texture coordinate generation function* (which may be the identity) The resulting texture coordinates are transformed by the *texture matrix* (this matrix may be used to effectively scale or rotate a texture that is applied to a primitive) Figure 4 shows some results of using texture coordinate generation functions

A number of commands control the values of parameters used in processing a vertex One group of commands manipulates transformation matrices these commands are designed to form an efficient means for generating and manipulating the transformations that occur in hierarchical 3D graphics scenes A matrix may be loaded or multiplied by a scaling, rotation, translation, or general matrix Another command controls which matrix is affected by a manipulation the model-view matrix, the texture matrix, or the *projection* matrix (to be described presently) Each of these three matrix types also has an associated stack onto which matrices may be pushed or popped

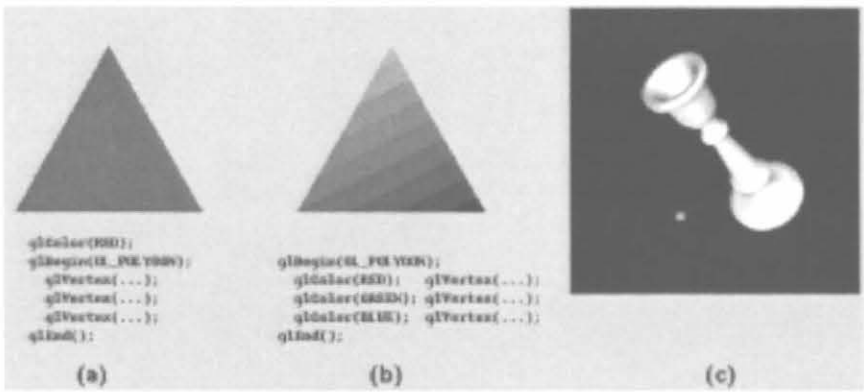


Figure 3. (a) A uniformly colored triangle. (b) A Gouraud shaded triangle. (c) A scene consisting of many lit, shaded polygons.

Lighting parameters are grouped into three categories: material parameters, that describe the reflectance characteristics of the surface being lit, light source parameters, that describe the emission properties of each light source, and lighting model parameters, that describe global properties of the lighting model. Lighting is performed on a per-vertex basis; lighting results are eventually interpolated across a line segment or polygon. The general form of the lighting equation includes terms for constant, diffuse, and specular illumination, each of which may be attenuated by the distance of the vertex from the light source. A programmer may sacrifice realism in favor of faster lighting calculations by indicating that the viewer, the light sources, or both should be assumed to be infinitely far from the scene. Figure 3 shows some results with lighting disabled and enabled.

4.2 Clipping and Projection

Once a primitive has been assembled from a group of vertices, it is subjected to clipping by *clip planes*. The positions of these planes (every OpenGL implementation must provide at least six) is specifiable using the **glClipPlane** command. Each plane may be enabled or disabled individually.

In the case of a point, the clip planes either have no effect on the point or annihilate it depending as the point lies inside or outside the intersection of the half-spaces determined by the clip planes. In the case of a line segment or polygon, the clip

planes may have no effect on annihilate or alter the original primitive In the later case new vertices may be created between edges described by original vertices color and texture coordinate values for these new vertices are found by appropri ately interpolating the values assigned to the original vertices

After the clip planes (if any) have been applied the vertex coordinates of the resulting primitive are transformed by the projection matrix Then *view frustum* clipping occurs View frustum clipping is like clip plane application but with fixed planes if coordinates after transformation are given by $(x\ y\ z\ w)$ then the six half spaces defined by these planes are $-w \leq x\ x \leq w\ -w \leq y\ y \leq w\ -w \leq z\ z \leq w$

With view frustum clipping completed each group of vertex coordinates is pro jected by computing x/w y/w and z/w The resulting values (which must each lie in $[-1\ 1]$) are multiplied and offset by parameters that control the size of the view port into which primitives are to be drawn The **glViewport** (for x/w and y/w) and **glDepthRange** (for z/w) commands control these parameters

4 3 Rasterization

Rasterization converts a projected viewport scaled primitive into a series of *frag ments* Each fragment comprises a location of a pixel in the framebuffer along with color texture coordinates and depth (z) When a line segment or polygon is ras terized these associated data are interpolated across the primitive to obtain a value for each fragment

The rasterization of each kind of primitive is controlled by a corresponding group of parameters One width affects point rasterization and another affects line seg ment rasterization Additionally a stipple sequence may be specified for line seg ments and a stipple pattern may be speicifed for polygons

Antialiasing may be enabled or disabled individually for each primitive type When enabled a coverage value is computed for each fragment describing the por tion of that fragment that is covered by the projected primitive This coverage value is used after texturing has been completed to modify the fragment s alpha value (in RGBA mode) or color index value (in color index mode)

4 3 1 Pixel Rectangles and Bitmaps

Pixel rectangles and *bitmaps* are the two primitives that are unaffected by the geo metric operations that occur in the pipeline prior to rasterization A pixel rectangle is a group of values destined for the framebuffer (typically the values represent col ors although provision is made for other types of data such as depth values) The

values stored as a block of data in host memory are sent using **glDrawPixels** Arguments to **glDrawPixels** indicate the memory address of the data, the type of data, and the width and height of the rectangle that the data values form In addition two groups of parameters are maintained that control the decoding of the stored values The first group describes how the values are packed in memory and provides a means for selecting a subrectangle from a larger containing rectangle The second group controls conversions that may be applied to the values after they obtained values may be scaled offset and mapped by means of look up tables These various parameters form a flexible means for specifying rectangular images stored in a variety of formats

Once obtained the resulting values produce a rectangle of fragments The location of this rectangle is controlled by the *current raster position* which is treated very much like a point (including associating a color and texture coordinates with it) except that it is set with a separate command (**glRasterPos**) that does not occur between **glBegin** and **glEnd** The rectangle s size is determined by its specified width and height as well as the setting of pixel rectangle zoom parameters (set with **glPixelZoom**)

A bitmap is similar to a pixel rectangle except that it specifies a rectangle of zeros and ones and is designed for describing characters that can be placed at a projected 3D location (through the current raster position) Each one in the bitmap produces a fragment whose associated values are those of the current raster position while each zero produces no fragment The **glBitmap** command also specifies offsets that control how the bitmap is placed with respect to the current raster position and how the current raster position is advanced after the bitmap is drawn (thus determining the relative positions of sequential bitmaps)

4 4 Texturing and Fog

OpenGL provides a general means for generating texture-mapped primitives (Figure 4) When texturing is enabled each fragment s texture coordinates index a texture image generating a *texel* This texel may have between one and four components so that a texture image may represent for example intensity only (one component) RGB color (three components) or RGBA color (four components) Once the texel is obtained it modifies the fragment s color according to a specifiable texture *environment*

A texture image is specified using **glTexImage** which takes arguments similar to those of **glDrawpixels**, so that the same image format may be used whether that image is destined for the framebuffer or texture memory In addition **glTexImage** may be used to specify mipmaps[3] so that a texture may be filtered as it is applied

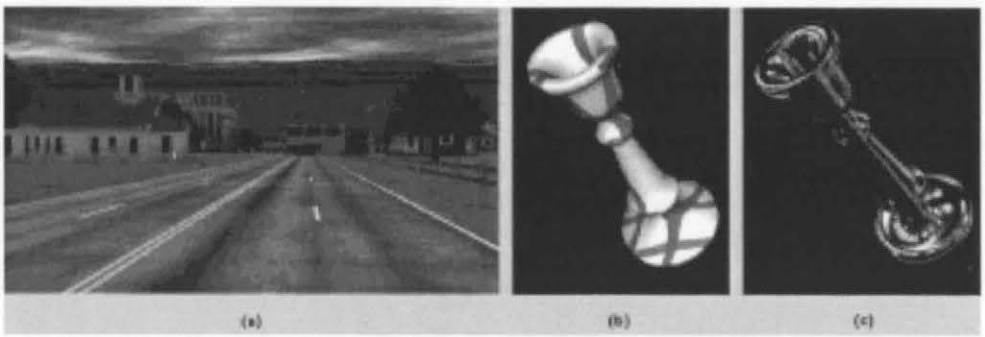


Figure 4. (a) A scene with a number of textures mapped onto primitives. (b) Contouring achieved with texture mapping and a texture coordinate generation function. (c) Reflectance mapping with a texture coordinate generation function.

to a primitive. The filter function (and whether or not it implies mipmaps) is controlled by a number of specifiable parameters using **glTexParameter**. The texture environment is selected with **glTexEnv**.

Finally, after texturing, a fog function (if enabled) is applied to each fragment. The fog function blends the incoming color with a constant (specifiable) fog color according to a computed weighting factor. This factor is a function of the distance (or an approximation to the distance) from the viewer to the 3D point that corresponds to the fragment. Exponential fog simulates atmospheric fog and haze, while linear fog may be used to produce depth-cueing.

4.5 The Framebuffer

The destination of rasterized fragments is the framebuffer, where the results of OpenGL rendering may be displayed. In OpenGL, the framebuffer consists of a rectangular array of pixels corresponding to the window allocated for OpenGL rendering. Each pixel is simply a set of some number of bits. Corresponding bits from each pixel in the framebuffer are grouped together into a *bitplane*; each bitplane contains a single bit from each pixel.

The bitplanes are grouped into several logical buffers: the *color*, *depth*, *stencil*, and *accumulation* buffers. The color buffer is where fragment color information is placed. The depth buffer is where fragment depth information is placed, and is typically used to effect hidden surface removal through z-buffering. The stencil buffer

contains values each of which may be updated whenever a corresponding fragment reaches the framebuffer Stencil values are useful in multi pass algorithms in which a scene is rendered several times to achieve such effects as CSG (union intersec tion and difference) operations on a number of objects and capping of objects sliced by clip planes

The accumulation buffer is also useful in multipass algorithms it can be ma nipulated so that it averages values stored in the color buffer This can effect such effects as full screen anti aliasing (by jittering the viewpoint for each pass) depth of field (by jittering the angle of view) and motion blur (by stepping the scene in time)[2] Multi pass algorithms are simple to implement in OpenGL because only a small number of parameters must be manipulated before each pass and changing the values of these parameters is both efficient and without side effects on the values of other parameters that must remain constant

OpenGL supports both double buffering and stereo so the color buffer is fur ther subdivided into four buffers the front left & right buffers and the back left & right buffers The front buffers are those that are typically displayed while the back buffers (in a double buffered application) are being used to compose the next frame A monoscopic application would use only the left buffers In addition there may be some number of auxiliary buffers (these are never displayed) into which fragments may be rendered Any of the buffers may be individually enabled or disabled for fragment writing

A particular copy of OpenGL may not provide depth stencil accumulation, or auxiliary buffers Further only some subset of the left & right front and left & right back buffers may be present Different buffers may be available (each with varying numbers of bits) depending on the platform and window system on which OpenGL is running Every window system must however provide at least one window type with a front (left) color buffer and depth stencil and accumulation buffers This guarantees a minimum configuration that a programmer may assume is present no matter where an OpenGL program is run

4 6 Per-Fragment Operations

Before being placed into its corresponding frame buffer location a fragment is sub jected to a series of tests and modifications each of which may be individually en abled disabled and controlled The tests and modifications include the stencil test the depth buffer test (typically used to achieve hidden surface removal) and blend ing We briefly describe only a subset of the tests for specifics the reader should consult [10]

The stencil test when enabled compares the value in the stencil buffer corre

sponding to the fragment with a reference value If the comparison succeeds then the stored stencil value may be modified by a function such as increment decrement or clear and the fragment proceeds to the next test If the test fails the stored value may be updated using a different function and the fragment is discarded Similarly the depth buffer test compares the fragment s depth value with the corresponding value stored in the depth buffer If the comparison succeeds the fragment is passed to the next stage and the fragment s depth value replaces the value stored in the depth buffer (if the depth buffer has been enabled for writing) If the comparison fails the fragment is discarded and no depth buffer modification occurs

Blending mixes a fragment s color with the corresponding color already stored in the framebuffer (blending occurs once for each color buffer enabled for writing) The exact blending function may be specified with **glBlendFunction**

Blending is the operation that actually achieves antialiasing for RGBA colors Recall that the coverage computation only modifies a fragment s alpha value this alpha value must be used to blend the fragment color with the already stored background color to obtain the antialiasing effect Blending is also used to achieve transparency

In addition to modifying individual framebuffer values with a series of fragments a whole buffer or buffers may be cleared to some specifiable constant value Clear values are maintained for the color buffers (all color buffers share a single value) the stencil buffer the depth buffer and the accumulation buffer

4 7 Miscellaneous Functions

4 7 1 Evaluators

Evaluators allow the specification of polynomial functions of one or two variables whose values determine primitives vertex coordinates normal coordinates color or texture coordinates A polynomial map specified in terms of the Bezier basis[1] may be given for any of these groups of values Once defined and enabled the maps are invoked in one of two ways The first way is to cause a single evaluation of each enabled map by specifying a point in the maps domain using **glEvalCoord** This command is meant to be placed between **glBegin** and **glEnd** so that individual primitives may be built each of which approximates a portion of a curve or surface The second method is to specify a grid in domain space using **glEvalMesh** Each vertex of the evaluated grid is a function of the defined polynomials **glEvalMesh** generates its own primitives and thus cannot be placed between **glBegin** and **glEnd**

The evaluator interface provides a basis for building a more general curve and surface package on top of OpenGL One advantage of providing the evaluators in

OpenGL instead of a more complex NURBS interface is that applications that rep resent curves and surfaces as other than NURBS or that make use of special surface properties still have access to efficient polynomial evaluators (that may be imple mented in graphics hardware) without incurring the costs of converting to a NURBS representation

4 7 2 Display Lists

A display list encapsulates a group of OpenGL commands so that they may be later issued (in the order originally specified) by simply naming the display list This is accomplished by surrounding the commands to be encapsulated with **glBeginList** and **glEndList** **glBeginList** takes an integer argument that is the numeric name of the display list

Display lists may be redefined but not edited The lack of editing simplifies dis-play list memory management in the OpenGL server eliminating the performance penalty such management would incur Display lists may however be nested (one display list may invoke another) An effect similar to display list editing may thus be obtained by (1) building a list containing a number of subordinate lists (2) re defining the subordinate lists

A single display list is invoked with **glCallList** **glCallLists** calls a series of display lists in succession Arguments to **glCallLists** specify an array of integers that are added to a *list base* to form the series of display list numbers **glCallLists** is useful to display a string of characters when the commands that generate each char acter have been encapsulated in their own display list Section 6 gives an example using **glCallLists**

4 7.3 Feedback and Selection

As described so far OpenGL renders primitives into the framebuffer OpenGL has two additional modes *Feedback* mode returns information about primitives (ver tex coordinates color and texture coordinates) after they have been processed but before they are rasterized This mode is useful for instance if OpenGL output is to be fed to a pen plotter instead of a framebuffer

In *selection* mode OpenGL returns a *hit* whenever a (clipped) primitive lies within the view frustum This mode is used for instance to determine which por tions of a scene lie within a region of the window centered around the mouse po sition (this is often termed *picking*) The Utility Library provides routines to ma-nipulate the transformations so that when the scene is redrawn only those portions that lie within a specified region about a specified position will return hits Each hit

returns the contents of the *selection stack* which may be manipulated as the scene is drawn By appropriately manipulating the stack the application can identify the scene features that intersected the selection region

4 7 4 OpenGL State

Finally the value of nearly any OpenGL parameter may be obtained by an appropriate *get* command There is also a stack of parameter values that may be pushed and popped For stacking purposes all parameters are divided into 21 functional groups any combination of these groups may be pushed onto the attribute stack in one operation (a pop operation automatically restores only those values that were last pushed) The get commands and parameter stacks make it possible to implement various libraries each without interfering with another s OpenGL usage

5 Integration in a Window System

OpenGL draws 3D and 2D scenes into a framebuffer but to be useful in a heterogeneous environment OpenGL must be made subordinate to a window system that allocates and controls framebuffer resources We describe how OpenGL is integrated into the X Window System but integration into other window systems (Windows NT for instance) is similar

X provides both a procedural interface and a network protocol for creating and manipulating framebuffer windows and drawing certain 2D objects into those windows OpenGL is integrated into X by making it a formal X extension called *GLX* GLX consists of about a dozen calls (with corresponding network encodings) that provide a compact general embedding of OpenGL in X As with other X extensions (two examples are Display PostScript and PEX) there is a specific network protocol for OpenGL rendering commands encapsulated in the X byte stream

OpenGL requires a region of a framebuffer into which primitives may be rendered In X such a region is called a *drawable* A *window* one type of drawable has associated with it a *visual* that describes the window s framebuffer configuration In GLX the visual is extended to include information about OpenGL buffers that are not present in unadorned X (depth stencil accumulation front, back etc)

X also provides a second type of drawable the *pixmap* which is an off screen framebuffer GLX provides a *GLX pixmap* that corresponds to an X pixmap but with additional buffers as indicated by some visual The GLX pixmap provides a means for OpenGL applications to render off-screen into a software buffer

To make use of an OpenGL capable drawable the programmer creates an OpenGL context targeted to that drawable When the context is created a copy of an OpenGL

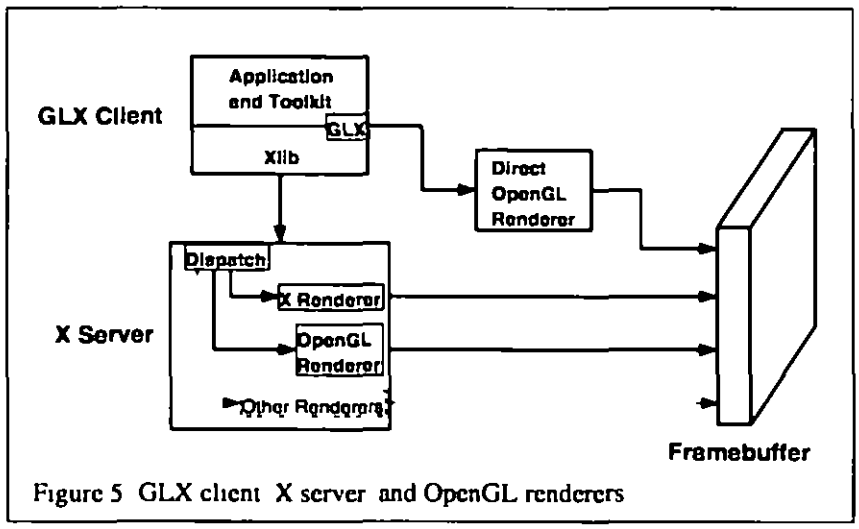


Figure 5 GLX client X server and OpenGL renderers

renderer is initialized with the visual information about the drawable This OpenGL renderer is conceptually (if not actually) part of the X server so that once created an X client may *connect* to the OpenGL context and issue OpenGL commands (Figure 5) Multiple OpenGL contexts may be created that are targeted to distinct or shared drawables Any OpenGL-capable drawable may also be used for standard X drawing (those buffers of the drawable that are unused by X are ignored by it) Calls are provided to synchronize drawing between OpenGL and X it is the client s responsibility to carry out this synchronization if required

A GLX client that is running on a computer of which the graphics subsystem is a part may avoid passing OpenGL tokens through the X server Such direct rendering may result in increased graphics performance since the overhead of token encoding decoding and dispatching is eliminated Direct rendering is supported but not required by GLX (a client may determine whether or not a server provides direct rendering) Direct rendering is feasible because sequentiality need not be maintained between X commands and OpenGL commands except where commands are explicitly synchronized

6 Example Three Kinds of Text

To illustrate the flexibility of OpenGL in performing different types of rendering tasks we outline three methods for the particular task of displaying text The three

methods are using bitmaps using line segments to generate outlined text and using a texture to generate antialiased text

The first method defines a font as a series of display lists each of which contains a single bitmap

```
for i = start + a   to  start + z  {
  glBeginList(i)
    glBitmap(      )
  glEndList()
}
```

Recall that `glBitmap` specifies both a pointer to an encoding of the bitmap and offsets that indicate how the bitmap is positioned relative to previous and subsequent bitmaps In GLX the effect of defining a number of display lists in this way may also be achieved by calling `glXUseXFont` `glXUseXFont` generates a number of display lists each of which contains the bitmap (and associated offsets) of a single character from the specified X font In either case the string Bitmapped Text whose origin is the projection of a location in 3D is produced by

```
glRasterPos3i(x  y  z)
glListBase(start)
glCallLists( Bitmapped Text   14  GL_BYTE)
```

See Figure 6a `glListBase` sets the display list base so that the subsequent `glCallLists` references the characters just defined The second argument to `glCallLists` indicates the length of the string the third argument indicates that the string is an array of 8 bit bytes (16 and 32 bit integers may be used to access fonts with more than 256 characters)

The second method is similar to the first but uses line segments to outline each character Each display list contains a series of line segments

```
glTranslate(ox  oy  0)
glBegin(GL_LINES)
  glVertex(    )

glEnd()
glTranslate(dx-ox  dy-oy  0)
```

The initial `glTranslate` updates the transformation matrix to position the character with respect to a character origin The final `glTranslate` updates that character origin in preparation for the following character A string is displayed with this

method just as in the previous example but since line segments have 3D position the text may be oriented as well as positioned in 3D (Figure 6b) More generally the display lists could contain both polygons and line segments and these could be antialiased

Finally a different approach may be taken by creating a texture image containing an array of characters A certain range of texture coordinates thus corresponds to each character in the texture image Each character may be drawn in any size and in any 3D orientation by drawing a rectangle with the appropriate texture coordinates at its vertices

```
glTranslate(ox  oy  0)
glBegin(GL_QUADS)
  glTexCoord(     ),
  glVertex(     )

glEnd()
glTranslate(dx-ox  dy-oy  0)
```

If each group of commands for each character is enclosed in a display list and the commands for describing the texture image itself are enclosed in another display list called TEX then the string "Texture mapped text" may be displayed by

```
glCallList(TEX)
glCallLists("Texture mapped text"   22  GL_BYTE)
```

One advantage of this method is that by simply using appropriate texture filtering the resulting characters are antialiased (Figure 6c)

7 Conclusion

OpenGL is a flexible procedural interface that allows a programmer to describe a variety of 3D rendering tasks It does not enforce a particular method of describing 3D objects but instead provides the basic means by which those objects no matter how described may be rendered This mechanistic view of rendering provides for efficient use of graphics hardware whether that hardware is a simple framebuffer or a graphics subsystem capable of directly manipulating 3D data OpenGL is rendering only so it is independent of the methods by which user input and other window system functions are achieved making the rendering portions of a graphical program that uses OpenGL platform independent

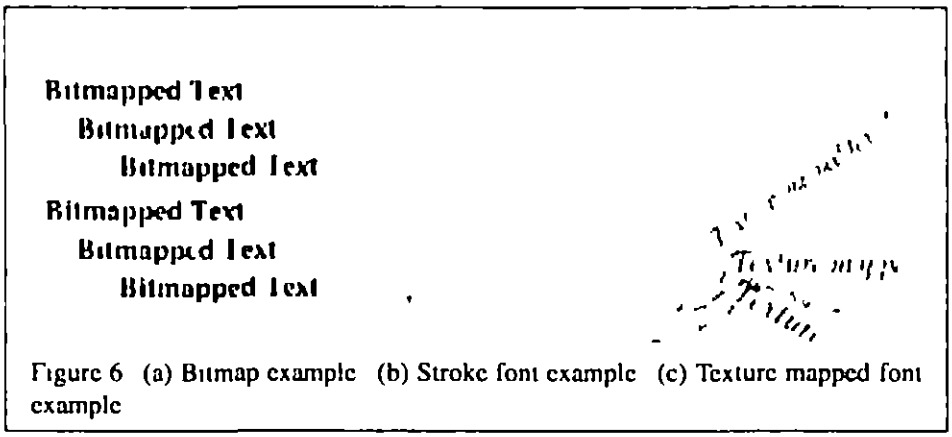


Figure 6 (a) Bitmap example (b) Stroke font example (c) Texture mapped font example

Because OpenGL imposes minimum structure on 3D rendering it is an excellent base on which to build libraries for handling structured geometric objects no matter what the particular structures may be Examples of such libraries include object oriented graphics toolkits that provide methods to display and manipulate complex objects endowed with a variety of attributes[11][12] A library that uses OpenGL for its rendering inherits OpenGL s platform independence making such a library available to a wide programming audience

References

[1] Gerald Farin *Curves and Surfaces for Computer Aided Geometric Design* Academic Press Boston Ma second edition 1990

[2] Paul Haeberli and Kurt Akeley The accumulation buffer Hardware support for high quality rendering In *Proceedings of SIGGRAPH 90* pages 309–318 1990

[3] Paul S Heckbert A survey of texture mapping *IEEE CG & A* pages 56–67 November 1986

[4] Adobe Systems Incorporated *PostScript Language Reference Manual* Addison Wesley Reading Mass 1986

[5] International Standards Organization International standard information processing systems — computer graphics — graphical kernel system for three

dimensions (GKS 3D) functional description Technical Report ISO Document Number 9905 1988(E) American National Standards Institute New York 1988

[6] Jeff Stevenson PEXlib specification and C language binding version 5 1P *The X Resource* Special Issue B September 1992

[7] Adrian Nye *X Window System User s Guide* volume 3 of *The Definitive Guides to the X Window System* O Reilly and Associates Sebastapol Ca. 1987

[8] Paula Womack ed PEX protocol specification and encoding version 5 1P *The X Resource* Special Issue A May 1992

[9] PHIGS+ Committee Andries van Dam chair PHIGS+ functional description revision 3 0 *Computer Graphics* 22(3) 125–218 July 1988

[10] Mark Segal and Kurt Akeley The OpenGL graphics system A specification Technical report, Silicon Graphics Computer Systems Mountain View Ca 1992

[11] Paul S Strauss and Rikk Carey An object oriented 3D graphics toolkit In *Proceedings of SIGGRAPH 92* pages 341–349 1992

[12] Garry Wiegand and Bob Covey *HOOPS Reference Manual Version 3 0* Ithaca Software 1991